

Journal

SECURITY ANALYSIS

AUGUST
15, 2005 [HTTP://WWW.CODEBREAKERS-JOURNAL.COM](http://www.codebreakers-journal.com)

VOL. 2, NO. 3
2005

This tutorial aims to collate information from a variety of sources and present it in a way which is accessible to beginners. Although detailed in parts, it is oriented towards reverse code engineering and superfluous information has been omitted.

Portable Executable File Format – A Reverse Engineer View

Goppit

Legal Information

The information contained herein is not a license, either expressly or impliedly, to any intellectual property owned or controlled by any of the authors or developers of Dissection Labs. The information contained herein is provided on an "AS IS" basis and to the maximum extent permitted by applicable law, this information is provided AS IS AND WITH ALL FAULTS, and the authors and developers of Dissection Labs hereby disclaim all other warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties, duties or conditions of merchantability, of fitness for a particular purpose, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence, all with regard to the contribution.

ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO DISSECTION LABS PUBLISHED WORKS.

IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF DISSECTION LABS BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR PUNITIVE OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO THIS DOCUMENT, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright 2004/2005 and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited.

This article can be found at <http://www.CodeBreakers-Journal.com>.

1. Table of Contents

- 2. Introduction 5
- 3. Basic Structure 7
- 4. The DOS Header 11
- 5. The PE Header 13
- 6. The Data Directory 24
- 7. The Section Table 26
- 8. The PE File Sections 28
- 9. The Export Section 32
- 10. The Import Section 38
- 11. The Loader 44
- 12. Navigating Imports on Disk 47
- 13. Adding Code to a PE File 54
- 14. Adding Import to an Executable 61
- 15. Introduction to Packers 71
- 16. Infection of PE Files by Viruses 83
- 17. Conclusion 85
- 18. Relative Virtual Addressing Explained 87
- 19. References & Bibliography 91
- 20. Tools Used 93
- 21. Appendix: Complete PE Offset Reference 95

--	--

2.Introduction

This tutorial aims to collate information from a variety of sources and present it in a way which is accessible to beginners. Although detailed in parts, it is oriented towards reverse code engineering and superfluous information has been omitted. You will see I have borrowed heavily from various published works and all authors are remembered with gratitude in the reference section at the end.

PE is the native Win32 file format. Every win32 executable (except VxDs and 16-bit DLLs) uses PE file format. 32bit DLLs, COM files, OCX controls, Control Panel Applets (.CPL files) and .NET executables are all PE format. Even NT's kernel mode drivers use PE file format.

Why do we need to know about it? 2 main reasons. Adding code to executables (e.g. keygen injection or adding functionality) and manually unpacking executables. With respect to the latter, most shareware nowadays comes "packed" in order to reduce size and to provide an added layer of protection.

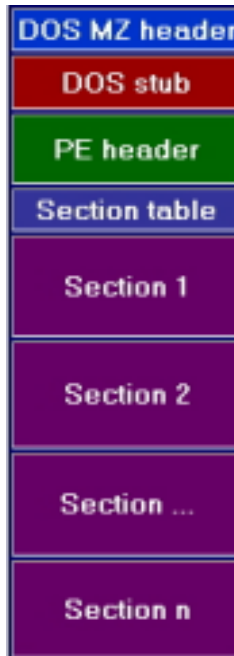
In a packed executable, the import tables are usually destroyed and data is often encrypted. The packer inserts code to unpack the file in memory upon execution, and then jumps to the original entry point of the file (where the original program actually starts executing). If we manage to dump this memory region after the packer finished unpacking the executable, we still need to fix the sections and import tables before our app will run. How will we do that if we don't even know what the PE format is?

The example executable I have used throughout this text is BASECALC.exe, a very useful app from fravia's site for calculating and converting decimal, hex, binary and octal. It is coded in Borland Delphi 2.0 which makes it ideal as an example to illustrate how Borland compilers leave the OriginalFirstThunks null (more on this later).

--	--

3.Basic Structure

The picture shows the basic structure of a PE file.



At a minimum, a PE file will have 2 sections; one for code and the other for data. An application for Windows NT has 9 predefined sections named .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug. Some applications do not need all of these sections, while others may define still more sections to suit their specific needs.

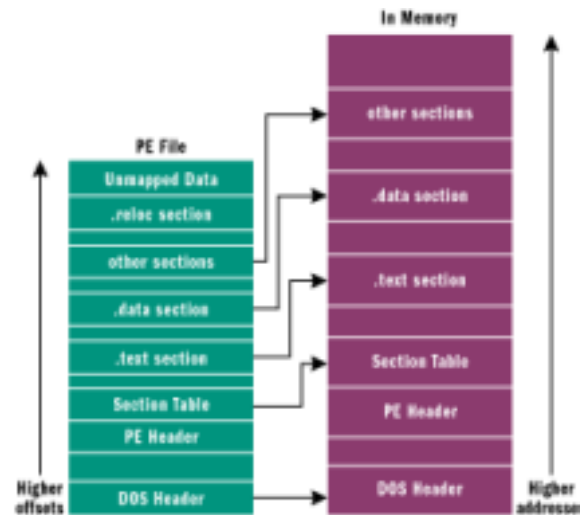
The sections that are most commonly present in an executable are:

- Executable Code Section, named .text (Micro\$oft) or CODE (Borland)
- Data Sections, named .data, .rdata, or .bss (Micro\$oft) or DATA (Borland)
- Resources Section, named .rsrc
- Export Data Section, named .edata
- Import Data Section, named .idata
- Debug Information Section, named .debug

The names are actually irrelevant as they are ignored by the OS and are present only for the convenience of the programmer. Another important point is that the structure of a PE file on disk is exactly the same as when it is loaded into memory so if you can locate info in the file on disk you will be able to find it when the file is loaded into memory.

However it is not copied exactly into memory. The windows loader decides which parts need mapping in and omits any others. Data that is not mapped in is placed at the end of the file past any parts that will be mapped in e.g. Debug information.

Also the location of an item in the file on disk will often differ from its location once loaded into memory because of the page-based virtual memory management that windows uses. When the sections are loaded into RAM they are aligned to fit to 4Kb memory pages, each section starting on a new page. Virtual memory is explained below.



The concept of virtual memory is that instead of letting software directly access physical memory, the processor and OS create an invisible layer between the two. Every time an attempt is made to access memory, the processor consults a "page table" that tells the process which physical memory address to actually use. It wouldn't be practical to have a table entry for each byte of memory (the page table would be larger than the total physical memory), so instead processors divide memory into pages. This has several advantages:

- 1) It enables the creation of multiple address spaces. An address space is an isolated page table that only allows access to memory that is pertinent to the current program or process. It ensures that programs are completely isolated from one another and that an error causing one program to crash is not able to poison another program's address space.
- 2) It enables the processor to enforce certain rules on how memory is accessed. Sections are needed in PE files because different areas in the file are treated differently by the memory manager when a module is loaded. At load time, the memory manager sets the access rights on memory pages for the different sections based on their settings in the section header. This determines whether a given section is readable, writable, or executable. This means each section must typically start on a fresh page.

However, the default page size for Windows is 4096 bytes (1000h) and it would be wasteful to align executables to a 4Kb page boundary on disk as that would make them significantly bigger than necessary. Because of this, the PE header has two different alignment fields; Section alignment and file alignment. Section alignment is how sections are aligned in memory as above. File alignment (usually 512 bytes or 200h) is how sections are aligned in the file on disk and is a multiple of disk sector size in order to optimize the loading process.

- 3) It enables a paging file to be used on the harddrive to temporarily store pages from the physical memory whilst they are not in use. For instance if an app has been loaded

	but becomes idle, its address space can be paged out to disk to make room for another app
--	---

which needs to be loaded into RAM. If the situation reverses, the OS can simply load the first app back into RAM and resume execution where it left off. An app can also use more memory than is physically available because the system can use the hard drive for secondary storage whenever there is not enough physical memory.

When PE files are loaded into memory by the windows loader, the in-memory version is known as a **module**. The starting address where file mapping begins is called an **HMODULE**. A module in memory represents all the code, data and resources from an executable file that is needed for execution whilst the term **process** basically refers to an isolated address space which can be used for running such a module.

--	--

4.The DOS Header

All PE files start with the DOS header which occupies the first 64 bytes of the file. It's there in case the program is run from DOS, so DOS can recognize it as a valid executable and run the DOS stub which is stored immediately after the header. The DOS stub usually just prints a string something like "This program must be run under Microsoft Windows" but it can be a full-blown DOS program. When building an application for Windows, the linker links a default stub program called WINSTUB.EXE into your executable. You can override the default linker behavior by substituting your own valid MS-DOS-based program in place of WINSTUB and using the -STUB: linker option when linking the executable file.

The DOS header is a structure defined in the windows.inc or winnt.h files. (If you have an assembler or compiler installed you will find them in the \include\ directory). It has 19 members of which magic and lfanew are of interest:

```
IMAGE_DOS_HEADER STRUCT
    e_magic      WORD      ?
    e_cblp       WORD      ?
    e_cp         WORD      ?
    e_crlc       WORD      ?
    e_cparhdr    WORD      ?
    e_minalloc   WORD      ?
    e_maxalloc   WORD      ?
    e_ss         WORD      ?
    e_sp         WORD      ?
    e_csum       WORD      ?
    e_ip         WORD      ?
    e_cs         WORD      ?
    e_lfanlc     WORD      ?
    e_ovno       WORD      ?
    e_res        WORD      4 dup(?)
    e_oemid      WORD      ?
    e_oeminfo    WORD      ?
    e_res2       WORD      10 dup(?)
    e_lfanew     DWORD     ?
IMAGE_DOS_HEADER ENDS
```

In the PE file, the magic part of the DOS header contains the value 4Dh, 5Ah (The letters "MZ" for Mark Zbikowsky one of the original architects of MS-DOS) which signifies a valid DOS header. MZ are the first 2 bytes you will see in any PE file opened in a hex editor (See example below.)

As we can see from its definition above, lfanew is a DWORD which sits at the end of the DOS header directly before the DOS stub begins. It contains the offset of the PE header, relative to the file beginning. The windows loader looks for this offset so it can skip the DOS stub and go directly to the PE header.

[NOTE: DWORD ("double word") = 4 bytes or 32bit value, WORD = 2 bytes or 16bit

value, sometimes you will also see dd for DWORD, dw for WORD and db for byte]

The definitions are helpful as they tell us the size of each member. This allows us to locate information of interest by counting the number of bytes from the start of the section or any other identifiable point.

As we said above, the DOS header occupies the first 64 bytes of the file - ie the first 4 rows seen in the hexeditor in the picture below. The last DWORD before the DOS stub begins contains 00h 01h 00h 00h. Allowing for reverse byte order this gives us 00 00 01 00h which is the offset where the PE header begins. The PE header begins with its signature 50h, 45h, 00h, 00h (the letters "PE" followed by two terminating zeroes).

If in the Signature field of the PE header, you find an NE signature here rather than a PE, you're working with a 16-bit Windows New Executable file. Likewise, an LE in the signature field would indicate a Windows 3.x virtual device driver (VxD). An LX here would be the mark of a file for OS/2 2.0.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	4D	5A	50	00	02	00	00	00	04	00	0F	00	FF	FF	00	00	; MZ.....yy..
00000010h:	B8	00	00	00	00	00	00	00	40	00	1A	00	00	00	00	00	;@.....
00000020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	;
00000040h:	BA	10	00	0E	1F	B4	09	CD	21	B8	01	4C	CD	21	90	90	; *....'.i!..Li!DD
00000050h:	54	68	69	73	20	70	72	6F	67	72	61	6D	20	6D	75	73	; This program mus
00000060h:	74	20	62	65	20	72	75	6E	20	75	6E	64	65	72	20	57	; t be run under W
00000070h:	69	6E	33	32	0D	0A	24	37	00	00	00	00	00	00	00	00	; in32..\$7.....
00000080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000090h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000100h:	50	45	00	00	4C	01	08	00	19	5E	42	2A	00	00	00	00	; PE.....^B*....
00000110h:	00	00	00	00	EO	00	8E	81	0B	01	02	19	00	A0	02	00	;â.ZD.....

We will discuss this in the next section.

5.The PE Header

The PE header is the general term for a structure named IMAGE_NT_HEADERS. This structure contains essential info used by the loader. IMAGE_NT_HEADERS has 3 members and is defined in windows.inc thus:

```
IMAGE_NT_HEADERS STRUCT
    Signature      DWORD      ?
    FileHeader     IMAGE_FILE_HEADER <>
    OptionalHeader IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

Signature is a DWORD containing the value 50h, 45h, 00h, 00h ("PE" followed by two terminating zeroes).

FileHeader is the next 20 bytes of the PE file and contains info about the physical layout & properties of the file e.g. number of sections. OptionalHeader is always present and forms the next 224 bytes. It contains info about the logical layout inside the PE file e.g. AddressOfEntryPoint. Its size is given by a member of FileHeader. The structures of these members are also defined in windows.inc

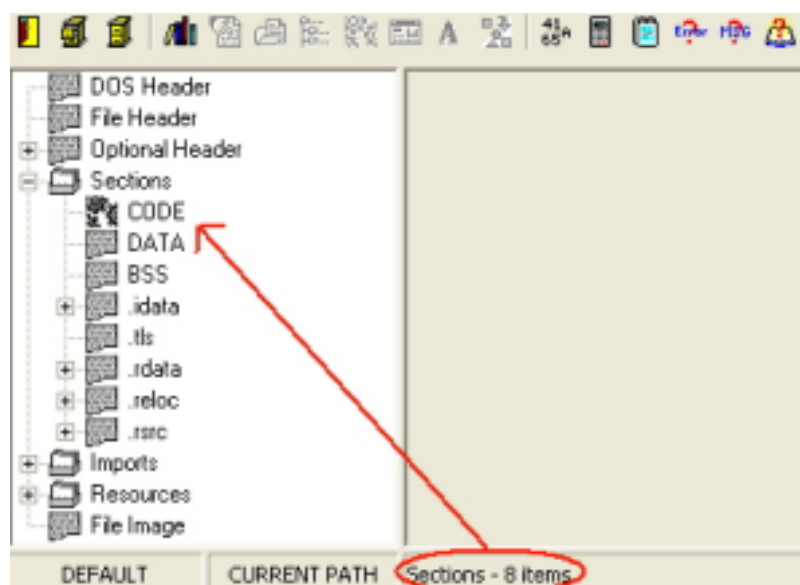
FileHeader is defined as follows:

```
IMAGE_FILE_HEADER STRUCT
    Machine          WORD      ?
    NumberOfSections WORD      ?
    TimeDateStamp    DWORD     ?
    PointerToSymbolTable DWORD   ?
    NumberOfSymbols   DWORD     ?
    SizeOfOptionalHeader WORD    ?
    Characteristics  WORD      ?
IMAGE_FILE_HEADER ENDS
```

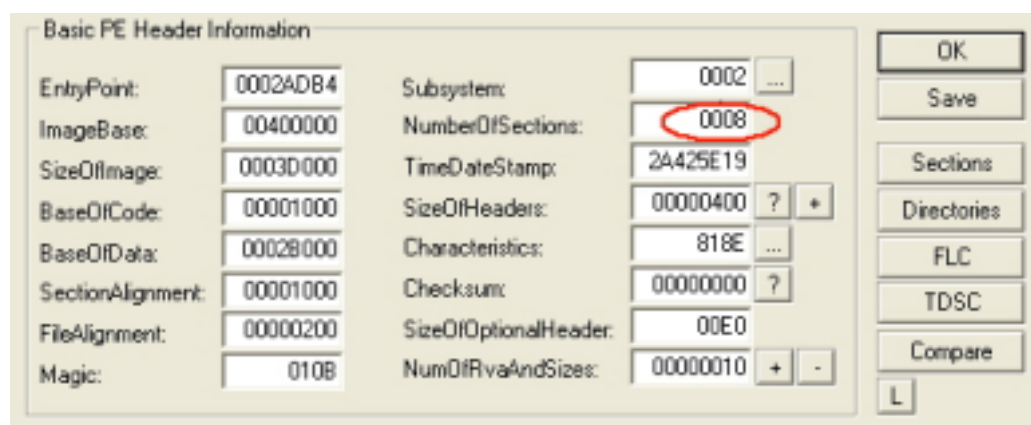
Most of these members are not of use to us but we must modify NumberOfSections if we add or delete any sections in the PE file. Characteristics contains flags which dictate for instance whether this PE file is an executable or a DLL. Back to our example in the Hexeditor, we can find NumberOfSections by counting a DWORD and a WORD (6 bytes) from the start of the PE header (to allow for the Signature and Machine members):

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	4D	5A	50	00	02	00	00	00	04	00	0F	00	FF	FF	00	00	; MZP.....yy..
00000010h:	B8	00	00	00	00	00	00	00	40	00	1A	00	00	00	00	00	;@.....
00000020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	;
00000040h:	B&	10	00	0E	1F	B4	09	CD	21	B8	01	4C	CD	21	90	90	; °....'.í!..Lí![]
00000050h:	54	68	69	73	20	70	72	6F	67	72	61	6D	20	6D	75	73	; This program mus
00000060h:	74	20	62	65	20	72	75	6E	20	75	6E	64	65	72	20	57	; t be run under W
00000070h:	69	6E	33	32	0D	0A	24	37	00	00	00	00	00	00	00	00	; in32..\$7.....
00000080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000090h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000100h:	50	45	00	00	4C	01	08	00	19	5E	42	2A	00	00	00	00	; PE..L....^B*....
00000110h:	00	00	00	00	EO	00	8E	81	0B	01	02	19	00	A0	02	00	;â.ž[].....

This can be verified by using any number of different (freeware) PE tools. For instance in PEBrowsePro:



Or in LordPE:



Or even from the "Subsystem" button of PEID:

Basic Information			
EntryPoint:	0002ADB4	SubSystem:	0002
ImageBase:	00400000	NumberOfSections:	0008
SizeOfImage:	0003D000	TimeDateStamp:	2A425E19
BaseOfCode:	00001000	SizeOfHeaders:	00000400
BaseOfData:	0002B000	Characteristics:	818E
SectionAlignment:	00001000	Checksum:	00000000
FileAlignment:	00000200	SizeOfOptionalHeader:	00E0
Magic:	010B	NumOfRvaAndSizes:	00000010

NOTE: PEID is an extremely useful tool - its main function is to scan executables and reveal the packer which has been used to compress/protect them. It also has the Krypto ANALyser plugin for detecting the use of cryptography in the executable e.g. CRC, MD5, etc. It can also utilise a user-defined list of packer signatures. This is the first tool to be used when embarking on any unpacking session.

Moving on to OptionalHeader, this takes up 224 bytes, the last 128 of which contain the Data Directory. Its definition is as follows:

```
IMAGE_OPTIONAL_HEADER32 STRUCT
    Magic                WORD        ?
    MajorLinkerVersion    BYTE        ?
    MinorLinkerVersion    BYTE        ?
    SizeOfCode            DWORD       ?
    SizeOfInitializedData DWORD       ?
    SizeOfUninitializedData DWORD     ?
    AddressOfEntryPoint    DWORD      ?
    BaseOfCode            DWORD       ?
    BaseOfData            DWORD       ?
    ImageBase            DWORD       ?
    SectionAlignment      DWORD       ?
    FileAlignment         DWORD       ?
    MajorOperatingSystemVersion WORD     ?
    MinorOperatingSystemVersion WORD     ?
    MajorImageVersion     WORD       ?
    MinorImageVersion     WORD       ?
    MajorSubsystemVersion WORD       ?
    MinorSubsystemVersion WORD       ?
    Win32VersionValue     DWORD      ?
    SizeOfImage           DWORD       ?
    SizeOfHeaders         DWORD       ?
    CheckSum              DWORD       ?
    Subsystem             WORD        ?
    DllCharacteristics     WORD        ?
    SizeOfStackReserve    DWORD       ?
    SizeOfStackCommit     DWORD       ?
    SizeOfHeapReserve     DWORD       ?
    SizeOfHeapCommit      DWORD       ?
    LoaderFlags           DWORD       ?
    NumberOfRvaAndSizes   DWORD       ?
    DataDirectory         IMAGE_DATA_DIRECTORY
IMAGE_OPTIONAL_HEADER32 ENDS
```


AddressOfEntryPoint -- The RVA of the first instruction that will be executed when the PE loader is ready to run the PE file. If you want to divert the flow of execution right from the start, you need to change the value in this field to a new RVA and the instruction at the new RVA will be executed first. Executable packers usually redirect this value to their decompression stub, after which execution jumps back to the original entry point of the app - the OEP. Of further note is the Starforce protection in which the CODE section is not present in the file on disk but is written into virtual memory on execution. The value in this field is therefore a VA (see [appendix](#) for further explanation).

ImageBase -- The preferred load address for the PE file. For example, if the value in this field is 400000h, the PE loader will try to load the file into the virtual address space starting at 400000h. The word "preferred" means that the PE loader may not load the file at that address if some other module already occupied that address range. In 99% of cases it is 400000h.

SectionAlignment -- The granularity of the alignment of the sections in memory. For example, if the value in this field is 4096 (1000h), each section must start at multiples of 4096 bytes. If the first section is at 401000h and its size is 10 bytes, the next section must be at 402000h even if the address space between 401000h and 402000h will be mostly unused.

FileAlignment -- The granularity of the alignment of the sections in the file. For example, if the value in this field is 512 (200h), each section must start at multiples of 512 bytes. If the first section is at file offset 200h and the size is 10 bytes, the next section must be located at file offset 400h: the space between file offsets 522 and 1024 is unused/undefined.

SizeOfImage -- The overall size of the PE image in memory. It's the sum of all headers and sections aligned to SectionAlignment.

SizeOfHeaders -- The size of all headers + section table. In short, this value is equal to the file size minus the combined size of all sections in the file. You can also use this value as the file offset of the first section in the PE file.

DataDirectory -- An array of 16 IMAGE_DATA_DIRECTORY structures, each relating to an important data structure in the PE file such as the import address table. This important structure will be discussed in the next section.

The overall layout of the PE Header can be seen from the following picture in the hexeditor. Note the DOS header and the parts of the PE header are always the same size (and shape) when viewed in the hexeditor, the DOS STUB can vary in size:

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	4D	5A	50	00	02	00	00	00	04	00	0F	00	FF	FF	00	00	; MZP.....ýý..
00000010h:	B8	00	00	00	00	00	00	00	40	00	1A	00	00	00	00	00	;0.....
00000020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	;
00000040h:	BA	10	00	0E	1F	B4	09	CD	21	B8	01	4C	CD	21	90	90	; °....'.í!.,.Lí!00
00000050h:	54	68	69	73	20	70	72	6F	67	72	61	6D	20	6D	75	73	; This program mus
00000060h:	74	20	62	65	20	72	75	6E	20	75	6E	64	65	72	20	57	; t be run under W
00000070h:	69	6E	33	32	0D	0A	24	37	00	00	00	00	00	00	00	00	; in32..\$7.....
00000080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000090h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000000f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
00000100h:	50	45	00	00	4C	01	08	00	19	5E	42	2A	00	00	00	00	; PE..L.....^B*....
00000110h:	00	00	00	00	E0	00	8E	81	0B	01	02	19	00	40	02	00	;â.ž0.....
00000120h:	00	DE	00	00	00	00	00	00	B4	AD	02	00	00	10	00	00	; .p.....'-.....
00000130h:	00	B0	02	00	00	00	40	00	00	10	00	00	00	02	00	00	; .°.....8.....
00000140h:	01	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	;
00000150h:	00	D0	03	00	00	04	00	00	00	00	00	00	02	00	00	00	; .D.....
00000160h:	00	00	10	00	00	40	00	00	00	10	00	00	10	00	00	00	;8.....
00000170h:	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00	;
00000180h:	00	D0	02	00	1E	18	00	00	00	40	03	00	00	8E	00	00	; .D.....0...ž..
00000190h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000001a0h:	00	10	03	00	04	2B	00	00	00	00	00	00	00	00	00	00	;+.....
000001b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000001c0h:	00	00	03	00	18	00	00	00	00	00	00	00	00	00	00	00	;
000001d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000001e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
000001f0h:	00	00	00	00	00	00	00	00	43	4F	44	45	00	00	00	00	;CODE....
00000200h:	88	9E	02	00	00	10	00	00	00	40	02	00	00	04	00	00	; ^ž.....
00000210h:	00	00	00	00	00	00	00	00	00	00	00	00	20	00	00	60	;
00000220h:	44	41	54	41	00	00	00	00	D4	06	00	00	00	B0	02	00	; DATA....Ŏ....°..

Besides the PE tools mentioned above, our favourite Ollydbg can also parse the PE headers into a meaningful display. Open our example in Olly and Press the M button or Alt+M to open the memory map - this shows how the sections of the PE file have been mapped into memory:

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00400000	00001000	BASECALC		PE header	Inag 01001002	R	RME	
00401000	00020000	BASECALC	CODE	code	Inag 01001002	R	RME	
00420000	00001000	BASECALC	DATA	data	Inag 01001002	R	RME	
0042C000	00001000	BASECALC	BSS		Inag 01001002	R	RME	
0042D000	00002000	BASECALC	.idata	imports	Inag 01001002	R	RME	
0042F000	00001000	BASECALC	.tls		Inag 01001002	R	RME	
00430000	00001000	BASECALC	.rdata		Inag 01001002	R	RME	
00431000	00003000	BASECALC	.reloc	relocations	Inag 01001002	R	RME	
00434000	00009000	BASECALC	.rsrc	resources	Inag 01001002	R	RME	

Now rightclick on PE header and select Dump in CPU. Next in the hex window, rightclick again and select special then PE header:

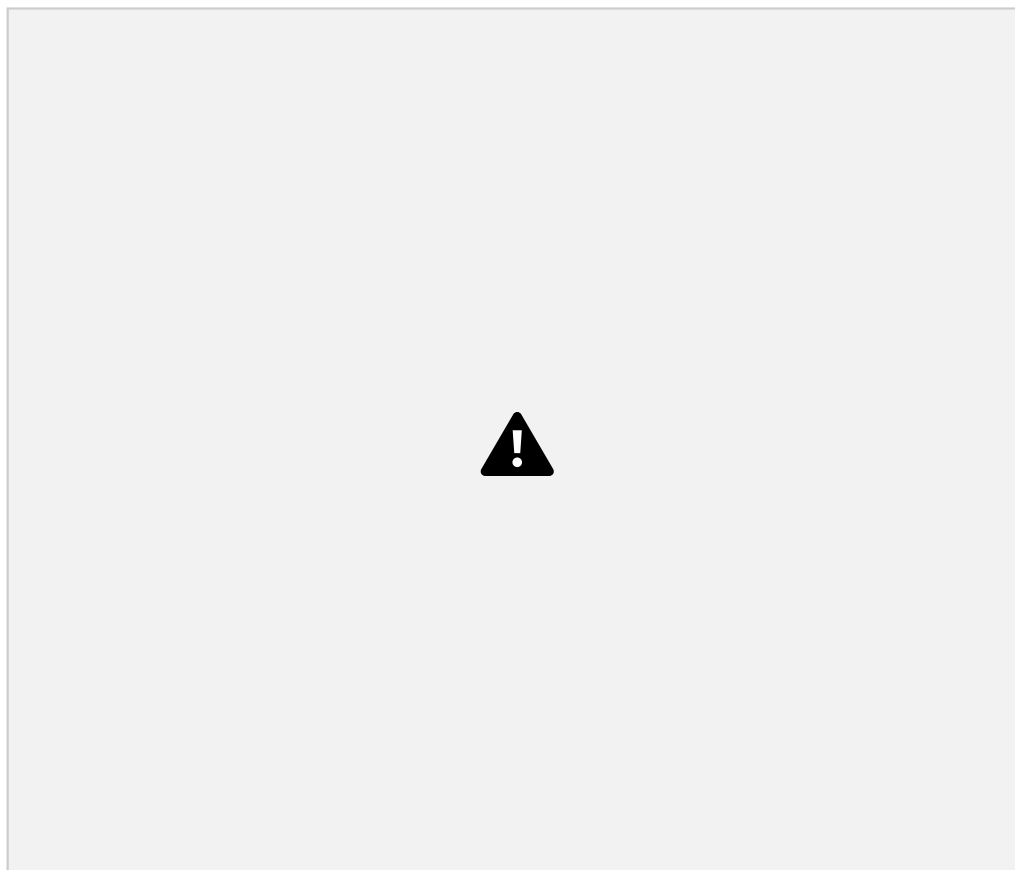
Address	Hex dump	ASCII
00400000	4D 5A 58 00 02 00 00 00 04 00 0F 00 FF FF 00 00	RZP.0...+.X. ..
00400010	B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00	0.....0.+.....
00400020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000..
00400030	00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 000..
00400040	BA 10 00 0E 1F 84 09 CD 21 88 01 4C CD 21 90 90	>.M? =180L=1st
00400050	54 60 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73	This program mus
00400060	74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57	t be run under M
00400070	69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00	in32..47.....
00400080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400100	50 45 00 00 4C 01 08 00 19 5E 42 2A 00 00 00 00	PE..LO.4^B....
00400110	00 00 00 00 E0 00 0E 01 00 01 02 19 00 A0 02 000.20004.50.
00400120	00 DE 00 00 00 00 00 00 B4 AD 02 00 00 10 00 00	.1.....140..>.
00400130	00 00 02 00 00 00 40 00 00 10 00 00 00 02 00 00	..0...0..>...0..
00400140	01 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	0.....>.....0..
00400150	00 CE 03 00 00 04 00 00 00 00 00 00 00 02 00 00	if?.....0....

Backup >
 Search for >
 Go to >
 ✓ Hex >
 Text >
 Short >
 Long >
 Float >
 Disassemble >
 Special > **PE header**
 Appearance >

Command

Analysing BASECALC: 1046 heuristical procedures, 847 calls to known, 233 calls to guessed functions

Now you should see this:



There are some specific points of interest in the optional header. If the last 2 members are both given bogus values, eg

LoaderFlags = ABDBFFDEh

NumberOfRvaAndSizes = DFFFDDDEh

Olly will determine the binary is a bad image and will eventually run the app without breaking at the entry point. If you were working with a virus then you would be infected. To avoid this when analyzing malware, open the app in the hexeditor and check the header first. If the NumberOfRvaAndSizes field alone is changed back to 10h the problem

is solved. A bogus value in this field can also cause some versions of Softice to reboot.

In addition the SizeOfRawData field in the section header can be given a very high value for one of the sections. This will then cause difficulties for many debugging and disassembling tools.

Another strange twist exists in the story of the PE header. Some of you may have noticed there is a section of garbage data between the DOS stub and the PE header in files linked by Micro\$ofts Linker. The origin of this data has been discussed in at least 3 forums and although it is not necessary to know about it, it is interesting so I will outline the details here.

PE files produced using M\$ development tools contain extra bytes in the DOS stub inserted by the linker Link.exe at compile time. In all cases, the penultimate DWORD is "Rich". This data is not present in files produced with other linkers (eg Borland, GCC, fasm, etc). This behavior is exhibited by all versions of M\$ Link.exe from v5.12.8078 which is part of the MASM32 package, up to v7.10.3077 which ships with the latest Visual C++ packages.

The data includes encrypted codes which identify the components used to compile the PE file. It is said to have led to the prosecution of a virus writer as it allowed M\$ to prove that the virus was compiled on his PC.

The dword after "Rich" is a key generated by the linker which repeats several times in the garbage data. When we compile a program the compiler puts the string "@comp.id" followed by a DWORD-sized compiler ID number in our obj file. When we link our obj file the linker extracts the comp.id number and XORs it with the key and writes it in the "garbage" as the 2nd DWORD before "Rich".

The "@comp.id" variables are hard coded:

ML.EXE Ver.6.14.8444 -> comp.id is 1220FC (You can search: FC2012)
ML.EXE Ver.7.00.9466 -> comp.id is 4024FA (search: FA2440)
ML.EXE Ver.7.10.2179 -> comp.id is 0F0883 (search: 83080F)
ML.EXE Ver.7.10.3077-> comp.id is 0F0C05 (search: 050C0F)
C++ Optimizing Compiler Version 12.00.8804 for 80x86 ->comp.id is 0B2306

The 1st DWORD before "Rich" is the key XORed with a hard coded constant 536E6144h. If we search "@comp.id" in our obj file and substitute the DWORD after it with zeroes we'll see that the second DWORD before "Rich" is equal to the key (DWORD after "Rich").

Here is an example of a simple "hello world" type program coded in MASM32 and open in the hexeditor. The extra bytes are highlighted:



Fortunately it is possible to patch the linker to stop this behaviour. There is a utility called SignFinder.exe by Asterix which allows you to find quickly the code which needs patching in any version of Link.exe. Using v5.12.8078 from MASM32 as an example:



So open Link.exe in Olly and press Ctrl+G. Enter 0044510C (the address from signfinder above + ImageBase of Link.exe which is 400000). Then highlight the add instruction as shown, rightclick and select binary>fill with NOPs:



It should look like this:



Finally rightclick again and select copy to executable>all modifications. Then click "copy all" and rightclick in the new window that pops up and select save file. The other versions of link.exe have the same code sequence at different locations which is patched in the same way.

If we use the patched linker to recompile the same example program we see the extra bytes have gone:



The only other differences between the 2 files are of course e_lfanew (the offset of the PE header), TimeDateStamp and SizeOfHeaders (which is effectively the offset of the first section).

--	--

6.The Data Directory

To recap, **DataDirectory** is the final 128 bytes of **OptionalHeader**, which in turn is the final member of the PE header **IMAGE_NT_HEADERS**.

As we have said, the **DataDirectory** is an array of 16 **IMAGE_DATA_DIRECTORY** structures, 8 bytes apiece, each relating to an important data structure in the PE file. Each array refers to a predefined item, such as the import table. The structure has 2 members which contain the location and size of the data structure in question:



VirtualAddress is the relative virtual address (RVA) of the data structure (see later section).

isize contains the size in bytes of the data structure.

The 16 directories to which these structures refer are themselves defined in



windows.inc:

For example, in LordPE the data directory for our example executable contains only 4 members (highlighted). The 12 unused ones are shown filled with zeros:



For example, in the above picture the "import table" fields contain the RVA and size of the **IMAGE_IMPORT_DESCRIPTOR** array - the Import Directory. In the hexeditor, the picture below shows the PE header with the data directory outlined in red. Each box represents one **IMAGE_DATA_DIRECTORY** structure, the first DWORD being **VirtualAddress** and the last being **isize**.

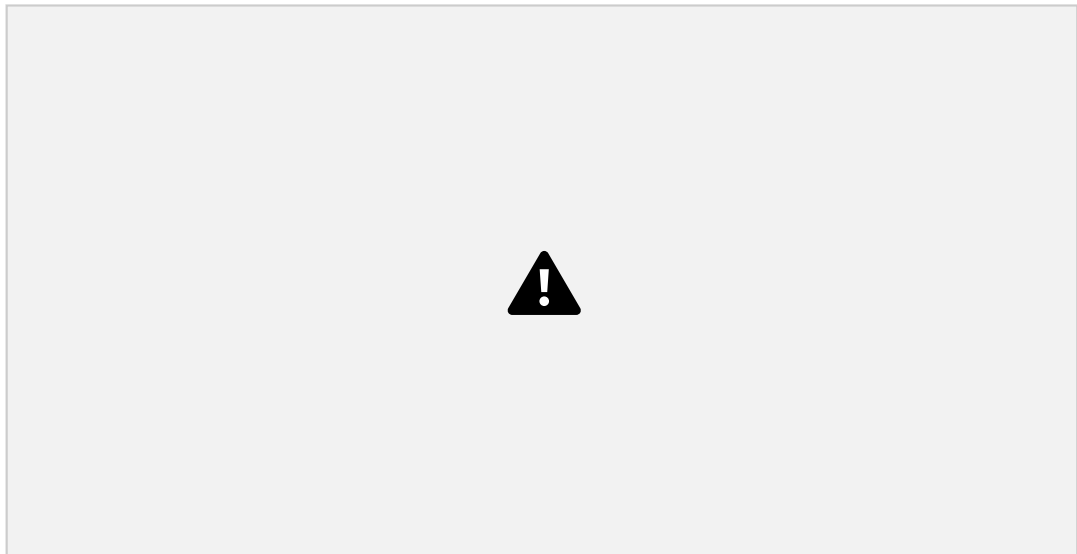


The Import Directory is highlighted in pink. The first 4 bytes are the RVA 2D000h (NB reverse order). The size of the Import Directory is 181Eh bytes. As we said above the position of these data directories from the beginning of the PE header is always the same i.e. the DWORD 80 bytes from the beginning of the PE header is always the RVA to the Import Directory.

To locate a particular directory, you determine the relative address from the data directory. Then use the virtual address to determine which section the directory is in. Once you determine which section contains the directory, the section header for that section is then used to find the exact offset.

7.The Section Table

This follows immediately after the PE header. It is an array of **IMAGE_SECTION_HEADER** structures, each containing the information about one section in the PE file such as its attribute and virtual offset. Remember the number of sections is the second member of FileHeader (6 bytes from the start of the PE header). If there are 8 sections in the PE file, there will be 8 duplicates of this structure in the table. Each header structure is 40 bytes apiece and there is no "padding" between them. The structure is defined in windows.inc thus:



Again, not all members are useful. I'll describe only the ones that are really important.

Name1 -- (NB this field is 8 bytes) The name is just a label and can even be left blank. Note this is not an ASCII string so it doesn't need a terminating zero.

VirtualSize -- (DWORD union) The actual size of the section's data in bytes. This may be less than the size of the section on disk (Size OfRawData) and will be what the loader allocates in memory for this section.

VirtualAddress -- The RVA of the section. The PE loader examines and uses the value in this field when it's mapping the section into memory. Thus if the value in this field is 1000h and the PE file is loaded at 400000h, the section will be loaded at 401000h.

SizeOfRawData -- The size of the section's data in the file on disk, rounded up to the next multiple of file alignment by the compiler.

PointerToRawData -- (Raw Offset) - incredibly useful because it is the offset from the file's beginning to the section's data. If it is 0, the section's data are not contained in the file and will be arbitrary at load time. The PE loader uses the value in this field to find where the data in the section is in the file.

Characteristics -- Contains flags such as whether this section contains executable code,

initialized data, uninitialized data, can it be written to or read from (see [appendix](#)).

NOTE: When searching for a specific section, it is possible to bypass the PE header entirely and start parsing the section headers by searching for the section name in the ASCII window of your hexeditor.

Back to our example in the hexeditor, our file has 8 sections as we saw in the PE header section.



After the section headers we find the sections themselves. In the file on disk, each section starts at an offset that is some multiple of the FileAlignment value found in OptionalHeader. Between each section's data there will be 00 byte padding.

When loaded into RAM, the sections always start on a page boundary so that the first byte of each section corresponds to a memory page. On x86 CPUs pages are 4kB aligned, whilst on IA-64, they are 8kB aligned. This alignment value is stored in SectionAlignment also in OptionalHeader.

For example, if the optional header ends at file offset 981 and FileAlignment is 512, the first section will start at byte 1024. Note that you can find the sections via the PointerToRawData or the VirtualAddress, so there is no need to bother with alignments.

In the picture above, the Import Data Section (.idata) will start at offset 0002AC00h (highlighted pink, NB reverse byte order) from the start of the file. Its size, given by the DWORD before, will be 1A00h bytes.

8.The PE File Sections

The sections contain the main content of the file, including code, data, resources, and other executable information. Each section has a header and a body (the raw data). The section headers are contained in the Section Table but section bodies lack a rigid file structure. They can be organized almost any way a linker wishes to organize them, as long as the header is filled with enough information to be able to decipher the data.

An application for Windows NT typically has the nine predefined sections named `.text`, `.bss`, `.rdata`, `.data`, `.rsrc`, `.edata`, `.idata`, `.pdata`, and `.debug`. Some applications do not need all of these sections, while others may define still more sections to suit their specific needs.

Executable Code

In Windows NT all code segments reside in a single section called **.text** or **CODE**. Since Windows NT uses a page-based virtual memory management system, having one large code section is easier to manage for both the operating system and the application developer. This section also contains the entry point mentioned earlier and the jump thunk table (where present) which points to the IAT (see [import theory](#)).

Data

The **.bss** section represents uninitialized data for the application, including all variables declared as static within a function or source module.

The **.rdata** section represents read-only data, such as literal strings, constants, and debug directory information.

All other variables (except automatic variables, which appear on the stack) are stored in the **.data** section. These are application or module global variables.

Resources

The **.rsrc** section contains resource information for a module. The first 16 bytes comprises a header like most other sections, but this section's data is further structured into a resource tree which is best viewed using a resource editor. A good one, ResHacker, is free and allows editing, adding, deleting, replacing and copying resources:



This is a powerful tool for cracking purposes as it will quickly display dialog boxes including those concerning incorrect registration details or nag screens. A shareware app can often be cracked just by deleting the nagscreen dialog resource in ResHacker.

Export data

The **.edata** section contains the Export Directory for an application or DLL. When present, this section contains information about the names and addresses of exported functions. We will discuss these in greater depth later.

Import data

The **.idata** section contains various information about imported functions including the Import Directory and Import Address Table. We will discuss these in greater depth later.

Debug information

Debug information is initially placed in the **.debug** section. The PE file format also supports separate debug files (normally identified with a .DBG extension) as a means of collecting debug information in a central location. The debug section contains the debug

information, but the debug directories live in the `.rdata` section mentioned earlier. Each of those directories references debug information in the `.debug` section.

Thread Local Storage

Windows supports multiple threads of execution per process. Each thread has its own private storage, Thread Local Storage or TLS, to keep data specific to that thread, such as pointers to data structures and resources that the thread is using. The linker can create a `.tls` section in a PE file that defines the layout for the TLS needed by routines in the executable and any DLLs to which it directly refers. Each time the process creates a thread, the new thread gets its own TLS, created using the `.tls` section as a template.

Base Relocations

When the linker creates an EXE file, it makes an assumption about where the file will be mapped into memory. Based on this, the linker puts the real addresses of code and data items into the executable file. If for whatever reason the executable ends up being loaded somewhere else in the virtual address space, the addresses the linker plugged into the image are wrong. The information stored in the `.reloc` section allows the PE loader to fix these addresses in the loaded image so that they're correct again. On the other hand, if the loader was able to load the file at the base address assumed by the linker, the `.reloc` section data isn't needed and is ignored.

The entries in the `.reloc` section are called base relocations since their use depends on the base address of the loaded image. Base relocations are simply a list of locations in the image that need a value added to them. The format of the base relocation data is somewhat quirky. The base relocation entries are packaged in a series of variable length chunks. Each chunk describes the relocations for one 4KB page in the image.

For example, if an executable file is linked assuming a base address of 0x10000. At offset 0x2134 within the image is a pointer containing the address of a string. The string starts at physical address 0x14002, so the pointer contains the value 0x14002. You then load the file, but the loader decides that it needs to map the image starting at physical address 0x60000. The difference between the linker-assumed base load address and the actual load address is called the delta. In this case, the delta is 0x50000. Since the entire image is 0x50000 bytes higher in memory, so is the string (now at address 0x64002). The pointer to the string is now incorrect. The executable file contains a base relocation for the memory location where the pointer to the string resides. To resolve a base relocation, the loader adds the delta value to the original value at the base relocation address. In this case, the loader would add 0x50000 to the original pointer value (0x14002), and store the result (0x64002) back into the pointer's memory. Since the string really is at 0x64002, everything is fine with the world.

--	--

9.The Export Section

This section is particularly relevant to DLLs. The following passage from Win32 Programmer's Reference explains why:



Functions can be exported by a DLL in two ways; "by name" or "by ordinal only". An ordinal is a 16-bit (WORD-sized) number that uniquely identifies a function in a particular DLL. This number is unique only within the DLL it refers to. We will discuss exporting by ordinal only later.

If a function is exported by name, when other DLLs or executables want to call the function, they use either its name or its ordinal in **GetProcAddress** which returns the address of the function in its DLL. The Win32 Programmer's Reference explains how GetProcAddress works (although in reality there is more to it, not documented by M\$, more on this later). Note the sections I have highlighted:



GetProcAddress can do this because the names and addresses of exported functions are stored in a well defined structure in the Export Directory. We can find the Export Directory because we know it is the first element in the data directory and the RVA to it is contained at offset 78h from the start of the PE header (see [appendix](#)).

The export structure is called **IMAGE_EXPORT_DIRECTORY**. There are 11 members in the structure but some are not important:



nName -- The internal name of the module. This field is necessary because the name of the file can be changed by the user. If that happens, the PE loader will use this internal name.

nBase -- Starting ordinal number (needed to get the indexes into the address-of function array - see below).

NumberOfFunctions -- Total number of functions (also referred to as symbols) that are exported by this module.

NumberOfNames -- Number of symbols that are exported by name. This value is **not** the number of **all** functions/symbols in the module. For that number, you need to check NumberOfFunctions. It can be 0. In that case, the module may export by ordinal only. If there is no function/symbol to be exported in the first case, the RVA of the export table in the data directory will be 0.

AddressOfFunctions -- An RVA that points to an array of pointers to (RVAs of) the functions in the module - the Export Address Table (EAT). To put it another way, the RVAs to all functions in the module are kept in an array and this field points to the head of that array.

AddressOfNames -- An RVA that points to an array of RVAs of the names of functions in the module - the Export Name Table (ENT).

AddressOfNameOrdinals -- An RVA that points to a 16-bit array that contains the ordinals of the named functions - the Export Ordinal Table (EOT).



Thus the IMAGE_EXPORT_DIRECTORY structures point to three arrays and a table of ASCII strings. The important array is the EAT, which is an array of function pointers that contain the addresses of exported functions. The other 2 arrays (EAT & EOT) run parallel in ascending order based on the name of the function so that a binary search for a function's name can be performed and will result in its ordinal being found in the other array. The ordinal is simply an index into the EAT for that function.

.....
.....
.....
.....

[illegible]

For example, if a DLL exports 40 functions, it must have 40 members in the array pointed to by `AddressOfFunctions` (the EAT) and the `NumberOfFunctions` field must contain the value 40.

To find the address of a function from its name, the OS first obtains the values of `NumberOfFunctions` and `NumberOfNames` in the Export Directory. Next it walks the arrays pointed to by `AddressOfNames` (the ENT) and `AddressOfNameOrdinals` (the EOT) in parallel, searching for the function name. If the name is found in the ENT, the value in the associated element in the EOT is extracted and used as the index into the EAT.

For example, in our 40-function-DLL we are looking for functionX. If we find the name functionX (indirectly via another pointer) in the 39th element in the ENT, we look in the 39th element of the EOT and see the value 5. We then look at the 5th element of the EAT to find the RVA of functionX.

If you already have the ordinal of a function, you can find its address by going directly to the EAT. Although obtaining the address of a function from an ordinal is much easier and faster than using the name of the function, the disadvantage is the difficulty in the maintaining the module. If the DLL is upgraded/updated and the ordinals of the functions are altered, other programs that depend on the DLL will break.

Exporting by Ordinal Only

`NumberOfFunctions` must be at least equal to `NumberOfNames`. However sometimes `NumberOfNames` is less than `NumberOfFunctions`. When a function is exported by ordinal only it doesn't have entries in both ENT and EOT arrays - it doesn't have a name. The functions that don't have names are exported by ordinal only.

For example, if there are 70 functions but only 40 entries in the ENT, it means there are 30 functions in the module that are exported by ordinal only. Now how can we find out which functions these are? It's not easy. You must find out by exclusion, i.e. the entries in the EAT that are **not** referenced by the EOT contain the RVAs of functions that are exported by ordinal only.

The programmer can specify the starting ordinal number in a .def file. For example, the tables in the picture above could start at 200. In order to prevent the need for 200 empty entries first in the array, the **nBase** member holds the starting value and the loader subtracts the ordinal numbers from it to obtain the true index into the EAT.

Export Forwarding

Sometimes functions which appear to exported from a particular DLL actually reside in a completely different DLL. This is called export forwarding. For example, in WinNT, Win2k and XP, the kernel32.dll function `HeapAlloc` is forwarded to the `RtlAllocHeap` function exported by `ntdll.dll`. `NTDLL.DLL` also contains the native API set which is the direct interface with the windows kernel. Forwarding is performed at link time by a special instruction in the .DEF file.

Forwarding is one technique Microsoft employs to expose a common Win32 API set and to hide the significant low-level differences between the Windows NT and Windows 9x internal API sets. Applications are not supposed to call functions in the native API set since this would break compatibility between win9x and 2k/XP. This probably explains

	why packed executables which have been unpacked and had their imports reconstructed
--	---

manually on one OS may not run on the other OS because the API forwarding system or some other detail has been altered.

When a symbol (function) is forwarded its RVA clearly can't be a code or data address in the current module. Instead the EAT table contains a pointer to an ASCII string of the DLL and function name to which it is forwarded. In the prior example it would be `NTDLL.RtlAllocHeap`

If therefore the EAT entry for a function points to an address inside the Exports Section (ie the ASCII string) rather than outside into another DLL, you know that function is forwarded.

10. The Import Section

The import section (usually .idata) contains information about all the functions imported by the executable from DLLs (see last section for explanation). This information is stored in several data structures. The most important of these are the Import Directory and the Import Address Table which we will discuss next. In some executables there may also be Bound_Import and Delay_Import directories. The Delay_Import directory is not so important to us but we will discuss the Bound_Import directory later.

The Windows loader is responsible for loading all of the DLLs that the application uses and mapping them into the process address space. It has to find the addresses of all the imported functions in their various DLLs and make them available for the executable being loaded.

The addresses of functions inside a DLL are not static but change when updated versions of the DLL are released, so applications cannot be built using hardcoded function addresses. Because of this a mechanism had to be developed that allowed for these changes without needing to make numerous alterations to an executable's code at runtime. This was accomplished through the use of an Import Address Table (IAT). This is a table of pointers to the function addresses which is filled in by the windows loader as the DLLs are loaded.

By using a pointer table, the loader does not need to change the addresses of imported functions everywhere in the code they are called. All it has to do is add the correct address to a single place in the import table and its work is done.

The Import Directory

The Import Directory is actually an array of **IMAGE_IMPORT_DESCRIPTOR** structures. Each structure is 20 bytes and contains information about a DLL which our PE file imports functions from. For example, if our PE file imports functions from 10 different DLLs, there will be 10 **IMAGE_IMPORT_DESCRIPTOR** structures in this array. There's no field indicating the number of structures in this array. Instead, the final structure has fields filled with zeros.

As with Export Directory, you can find where the Import Directory is by looking at the Data Directory (80 bytes from beginning of PE header). The first and last members are most important:



The first member **OriginalFirstThunk**, which is a DWORD union, may at one time have been a set of flags. However, Microsoft changed its meaning and never bothered to update WINNT.H. This field really contains the RVA of an array of **IMAGE_THUNK_DATA** structures.

[By the way, a union is just a redefinition of the same area of memory. The union above doesn't contain 2 DWORDS but only one which could contain either the OriginalFirstThunk data or the Characteristics data.]

The **TimeDateStamp** member is set to zero unless the executable is bound when it contains -1 (see below). The **ForwarderChain** member was used for old-style binding and will not be considered here.

Name1 contains the a pointer (RVA) to the ASCII name of the DLL.

The last member **FirstThunk**, also contains the RVA of an array of DWORD-sized **IMAGE_THUNK_DATA** structures - **a duplicate of the first array**. If the function described is a bound import (see below) then FirstThunk contains the actual address of the function instead of an RVA to an **IMAGE_THUNK_DATA**. These structures are defined thus:



Each **IMAGE_THUNK_DATA** is a DWORD union that effectively only has one of 2 values. In the file on disk it either contains the ordinal of the imported function (in which case it will begin with an 8 - see export by ordinal only below) or an RVA to an **IMAGE_IMPORT_BY_NAME** structure. Once loaded the ones pointed at by FirstThunk are overwritten with the addresses of imported functions - this becomes the Import Address Table.

Each **IMAGE_IMPORT_BY_NAME** structure is defined as follows:



Hint -- contains the index into the Export Address Table of the DLL the function resides in. This field is for use by the PE loader so it can look up the function in the DLL's Export Address Table quickly. The name at that index is tried, and if it doesn't match then a binary search is done to find the name. As such this value is not essential and some linkers set this field to 0.

Name1 -- contains the name of the imported function. The name is a null-terminated ASCII string. Note that Name1's size is defined as a byte but it's really a variable-sized field. It's just that there is no way to represent a variable-sized field in a structure. The structure is provided so that you can refer to it with descriptive names.

The most important parts are the imported DLL names and the arrays of IMAGE_THUNK_DATA structures. Each IMAGE_THUNK_DATA structure corresponds to one imported function from the DLL. The arrays pointed to by OriginalFirstThunk and FirstThunk run parallel and are terminated by a null DWORD. There are separate pairs of arrays of IMAGE_THUNK_DATA structures for each imported DLL.

Or to put it another way, there are several IMAGE_IMPORT_BY_NAME structures. You create two arrays, then fill them with the RVAs of those IMAGE_IMPORT_BY_NAME structures, so both arrays contain exactly the same values (i.e. exact duplicate). Now you assign the RVA of the first array to OriginalFirstThunk and the RVA of the second array to FirstThunk.

The number of elements in the OriginalFirstThunk and FirstThunk arrays depends on the number of functions imported from the DLL. For example, if the PE file imports 10 functions from user32.dll, Name1 in the IMAGE_IMPORT_DESCRIPTOR structure will contain the RVA of the string "user32.dll" and there will be 10 IMAGE_THUNK_DATAs in each array.

The 2 parallel arrays have been called by several different names but the commonest are **Import Address Table** (for the one pointed at by FirstThunk) and **Import Name Table** or **Import Lookup Table** (for the one pointed at by OriginalFirstThunk).

Why are there two parallel arrays of pointers to the IMAGE_IMPORT_BY_NAME structures? The Import Name Tables are left alone and never modified. The Import Address Tables are overwritten with the actual function addresses by the loader. The loader iterates through each pointer in the arrays and finds the address of the function that each structure refers to. The loader then overwrites the pointer to IMAGE_IMPORT_BY_NAME with the function's address. The arrays of RVAs in the Import Name Tables remain unchanged so that if the need arises to find the names of imported functions, the PE loader can still find them.

Although the IAT is pointed to by entry number 12 in the Data Directory, some linkers don't set this directory entry and the app will run nevertheless. The loader only uses this to temporarily mark the IATs as read-write during import resolution and can resolve the imports without it.)

This is how the windows loader is able to overwrite the IAT when it resides in a read-only section. At load time the system temporarily sets the attributes of the pages containing the imports data to read/write. Once the import table is initialized the pages are set back to their original protected attributes.



Calls to imported functions take place via a function pointer in the IAT and can take 2 forms, one more efficient than the other. For example imagine the address 00405030 refers to one of the entries in the FirstThunk array that's overwritten by the loader with the address of GetMessage in USER32.DLL.

The efficient way to call GetMessage looks like this:

```
0040100C CALL DWORD PTR [00405030 ]
```

The inefficient way looks like this:

```
0040100C CALL [00402200]
```

```
.....
```

```
.....
```

```
00402200 JMP DWORD PTR [00405030]
```

i.e. the second method achieves the same but uses 5 additional bytes of code and takes longer to execute because of the extra jump.

Why are calls to imported functions implemented in this way? The compiler can't distinguish between calls to ordinary functions within the same module and imported functions and emits the same output for both: `CALL [XXXXXXXX]`

where XXXXXXXX has to be an actual code address (not a pointer) to be filled in by the linker later. The linker does not know the address of the imported function and so has to supply a substitute chunk of code - the JMP stub seen above.

The optimised form is obtained by using the `_declspec(dllimport)` modifier to tell the compiler that the function resides in a DLL. It will then output `CALL DWORD PTR [XXXXXXXX]`.

If `_declspec(dllimport)` has not been used when compiling an executable there will be a

whole collection of jump stubs for imported functions located together somewhere in the code. This has been known by various name such as the "transfer area", "trampoline" or "jump thunk table".

Functions Exported by Ordinal Only

As we discussed in the export section, some functions are exported by ordinal only. In this case, there will be no IMAGE_IMPORT_BY_NAME structure for that function in the caller's module. Instead, the IMAGE_THUNK_DATA for that function contains the ordinal of the function.

Before the executable is loaded, you can tell if an IMAGE_THUNK_DATA structure contains an ordinal or an RVA by looking at the most significant bit (MSB) or high bit. If set then the lower 31 bits are treated as an ordinal value. If clear, the value is an RVA to an IMAGE_IMPORT_BY_NAME. Microsoft provides a handy constant for testing the MSB of a dword, **IMAGE_ORDINAL_FLAG32**. It has the value of 80000000h.

For example, if a function is exported by ordinal only and its ordinal is 1234h, the IMAGE_THUNK_DATA for that function will be 80001234h.

Bound Imports


When the loader loads a PE file into memory, it examines the import table and loads the required DLLs into the process address space. Then it walks the array pointed at by FirstThunk and replaces the IMAGE_THUNK_DATAs with the real addresses of the import functions. This step takes time. If somehow the programmer can predict the addresses of the functions correctly, the PE loader doesn't have to fix the IMAGE_THUNK_DATAs each time the PE file is run as the correct address is already there. Binding is the product of that idea.

There is a utility named **bind.exe** that comes with Microsoft compilers that examines the IAT (FirstThunk array) of a PE file and replaces the IMAGE_THUNK_DATA dwords with the addresses of the import functions. When the file is loaded, the PE loader must check if the addresses are valid. If the DLL versions do not match the ones in the PE files or if the DLLs need to be relocated, the PE loader knows that the bound addresses are stale and it walks the Import Name Table (OriginalFirstThunk array) to calculate the new addresses.

Therefore although the INT is not necessary for an executable to load, if not present the executable cannot be bound. For a long time Borland's linker TLINK did not create an INT therefore files created by Borland could not be bound. We will see another consequence of the missing INT in the next section.

The Bound Import Directory

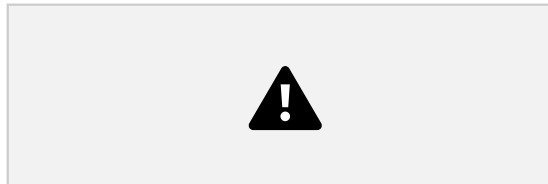
The information the loader uses to determine if bound addresses are valid is kept in a IMAGE_BOUND_IMPORT_DESCRIPTOR structure. A bound executable contains a list of these structures, one for each imported DLL that has been bound:

	<div></div>
--	--

The **TimeStamp** member must match the TimeDateStamp of the exporting DLL's FileHeader; if it doesn't match, the loader assumes that the binary is bound to a "wrong" DLL and will re-patch the import list. This can happen if the version of the exporting DLL doesn't match or if it has had to be relocated in memory.

The **OffsetModuleName** member contains the offset (not RVA) from the first IMAGE_BOUND_IMPORT_DESCRIPTOR to the name of the DLL in null-terminated ASCII.

The **NumberOfModuleForwarderRefs** member contains the number of IMAGE_BOUND_FORWARDER_REF structures that immediately follow this structure. These are defined thus:



As you can see they are identical to the previous structure apart from the final member which is reserved in any case. The reason there are 2 similar structures like this is that when binding against a function which is forwarded to another DLL, the validity of that forwarded DLL has to be checked at load time too. The IMAGE_BOUND_FORWARDER_REF contains the details of the forwarded DLLs.

For example the function HeapAlloc in kernel32.dll is forwarded to RtlAllocateHeap in ntdll.dll. If we created an app which imports HeapAlloc and used bind.exe on the app, there would be an IMAGE_BOUND_IMPORT_DESCRIPTOR for kernel32.dll followed by an IMAGE_BOUND_FORWARDER_REF for ntdll.dll.

NOTE: the names of the functions themselves are not included in these structures as the loader knows which functions are bound from the IMAGE_IMPORT_DESCRIPTOR (see above). There was an older style binding mechanism which differs slightly from this but has been phased out so I have omitted details here.

11. The Loader

This section is not essential but is for those who wish to dig a bit deeper into the workings of the OS. It shows how relevant the material in the last 2 sections is. First a brief overview of the stages involved in the loading process:

1. Read in the first page of the file with the DOS header, PE header, and section headers.
2. Determine whether the target area of the address space is available, if not allocate another area.
3. Using info in the section headers, map sections of the file to the appropriate places in the allocated address space.
4. If the file is not loaded at its target address (ImageBase), apply relocation fix-ups.
5. Go through list of DLLs in the imports section and load any that aren't already loaded (recursive).
6. Resolve all the imported symbols in the imports section.
7. Create the initial stack and heap using values from the PE header.
8. Create the initial thread and start the process.

What the loader does

When an executable is run, the windows loader creates a virtual address space for the process and maps the executable module from disk into the process' address space. It tries to load the image at the preferred base address but relocates it if that address is already occupied. The loader goes through the section table and maps each section at the address calculated by adding the RVA of the section to the base address. The page attributes are set according to the section's characteristic requirements. After mapping the sections in memory, the loader performs base relocations if the load address is not equal to the preferred base address in ImageBase.

The import table is then checked and any required DLLs are mapped into the process address space. After all of the DLL modules have been located and mapped in, the loader examines each DLL's export section and the IAT is fixed to point to the actual imported function address. If the symbol does not exist (which is very rare), the loader displays an error. Once all required modules have been loaded execution passes to the app's entry point.

The area of particular interest in RCE is that of loading the DLLs and resolving imports. This process is complicated and is accomplished by various internal (forwarded) functions and routines residing in ntdll.dll which are not documented by Microsoft. As we said previously function forwarding is a way for M\$ to expose a common Win32 API set and hide low level functions which may differ in different versions of the OS. Many familiar kernel32 functions such as GetProcAddress are simply thin wrappers around ntdll.dll exports such as LdrGetProcAddress which do the real work.

In order to see these in action you will need to install windbg and the windows symbol package (available free in Debugging Tools For Windows from M\$) or another kernel-mode debugger like SoftIce. You can only view these functions in Olly if you configure Olly to use

the M\$ symbolserver (search ARTeam forum for notes on this by Shub), otherwise all you will see is pointers and memory addresses without function names. However Olly is a user mode debugger and will only show you what's happening when your app has been loaded and will not allow you to see the loading process itself. Although the functionality of windbg is poor compared to Olly it does integrate with the OS well and will show the loading process:



The various APIs associated with loading an executable all converge on the kernel32.dll function LoadLibraryExW which in turn leads to the internal function LdrpLoadDll in ntdll.dll. This function directly calls 6 subroutines LdrpCheckForLoadedDll, LdrpMapDll, LdrpWalkImportDescriptor, LdrpUpdateLoadCount, LdrpRunInitializeRoutines, and LdrpClearLoadInProgress which perform the following tasks:

1. Check to see if the module is already loaded.
2. Map the module and supporting information into memory.
3. Walk the module's import descriptor table (find other modules this one is importing).
4. Update the module's load count as well as any others brought in by this DLL.
5. Initialize the module.
6. Clear some sort of flag, indicating that the load has finished.



A DLL may import other modules that start a cascade of additional library loads. The loader will need to loop through each module, checking to see if it needs to be loaded and then checking its dependencies. This is where `LdrpWalkImportDescriptor` comes in. It has two subroutines; `LdrpLoadImportModule` and `LdrpSnapIAT`. First it starts with two calls to `RtlImageDirectoryEntryToData` to locate the Bound Imports Descriptor and the regular Import Descriptor tables. Note that the loader is checking for bound imports first - an app which runs but doesn't have an import directory may have bound imports instead.

Next `LdrpLoadImportModule` constructs a Unicode string for each DLL found in the Import Directory and then employs `LdrpCheckForLoadedDll` to see if they have already been loaded.

Next the `LdrpSnapIAT` routine examines every DLL referenced in the Import Directory for a value of -1 (ie again checks for bound imports first). It then changes the memory protection of the IAT to `PAGE_READWRITE` and proceeds to examine each entry in the IAT before moving on to the `LdrpSnapThunk` subroutine.

`LdrpSnapThunk` uses a function's ordinal to locate its address and determine whether or not it is forwarded. Otherwise it calls `LdrpNameToOrdinal` which uses a binary search on the export table to quickly locate the ordinal. If the function is not found it returns `STATUS_ENTRYPOINT_NOT_FOUND`, otherwise it replaces the entry in the IAT with the API's entry point and returns to `LdrpSnapIAT` which restores the memory protection it changed at the beginning of its work, calls `NtFlushInstructionCache` to force a cache refresh on the memory block containing the IAT, and returns back to `LdrpWalkImportDescriptor`.

There is a peculiar difference between windows versions in that win2k insists that `ntdll.dll` is loaded either as a bound import or in the regular import directory before allowing an

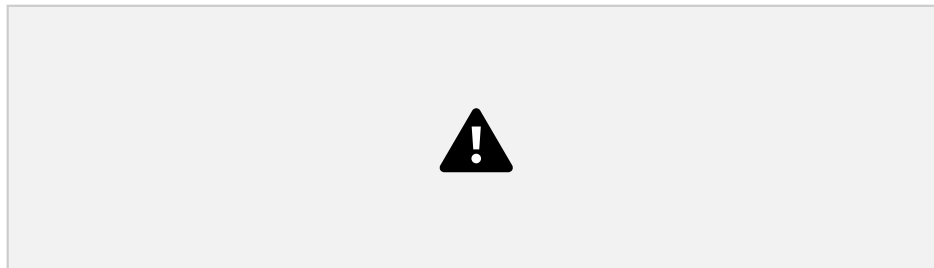
	executable to load, whereas win9x and XP will allow an app with no imports at all to load.
--	--

This brief overview is greatly simplified but illustrates how a call to LoadLibrary sets off a cascade of hidden internal subroutines which are deeply nested and recursive in places. The loader must examine every imported API in order to calculate a real address in memory and to see if an API has been forwarded. Each imported DLL may bring in additional modules and the process will be repeated over and over again until all dependencies have been checked.

12. Navigating Imports on

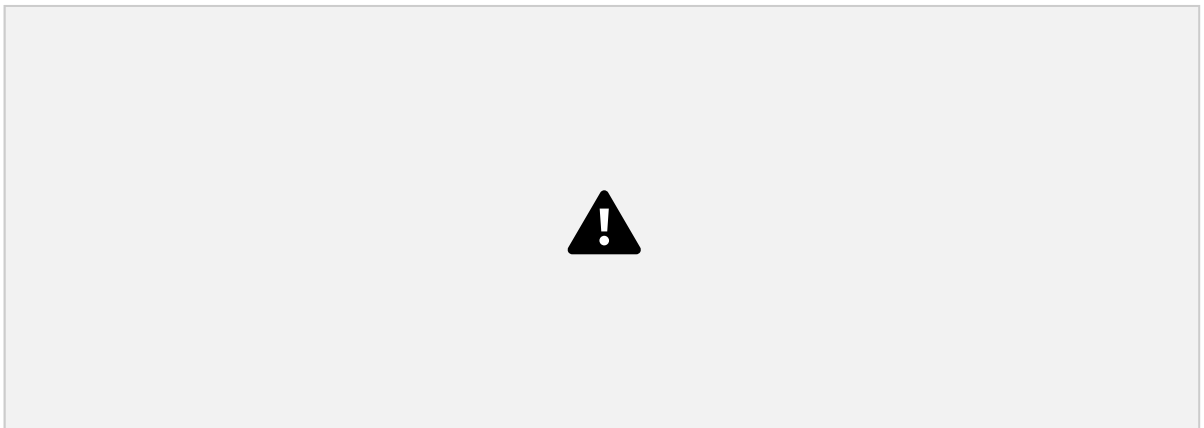
Disk

Back to our example in the hexeditor, we will navigate the import table to see what we can find. As we said previously, the RVA of the Import Directory is stored in the DWORD 80h bytes from the PE header which in our example is offset 180h and the RVA is 2D000h (see [Data Directory](#)). We now have to convert that RVA to a raw offset to peruse the correct area of our file on disk. Check the Section Table to see which section the address of the Import Directory lies in. In our case, the Import Directory starts at the beginning of the .idata section and we know that the section table holds the raw offset in the PointerToRawData field. In our example the offset is 2AC00h (see section table page). Any PE Editor will show this, e.g. LordPE:



The difference between the RVA and Raw Offset is $2D000 - 2AC00 = 2400h$. Make a note of this as it will be useful for converting further offsets. See [appendix](#) for more info on converting RVAs.

At offset 2AC00 we have the Import Directory - an array of IMAGE_IMPORT_DESCRIPTORs each of 20 bytes and repeating for each import library (DLL) until terminated by 20 bytes of zeros. In our hexeditor we see at 2AC00h:

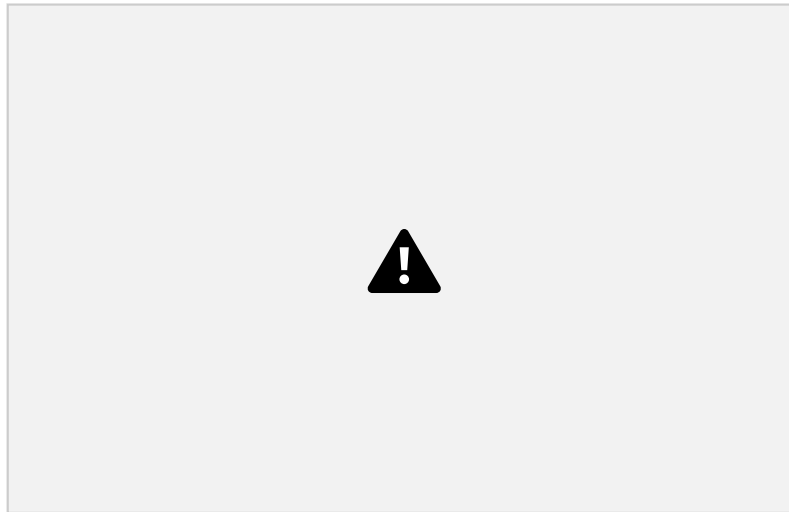


Each group of 5 DWORDS represents 1 IMAGE_IMPORT_DESCRIPTOR. The first shows that in this PE file OriginalFirstThunk, TimeDateStamp and ForwarderChain are set to 0. Eventually we come to a set of 5 DWORDS all set to 0 (also highlighted in red) which signifies the end of the array. We can see we are importing functions from 8 DLLs.

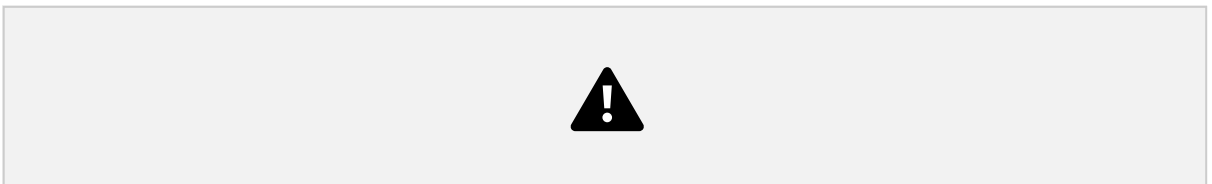
IMPORTANT NOTE: the OriginalFirstThunk fields in our example are all set to zero. This is common for executables made with Borland's compiler & linker and is noteworthy for the following reason. In a packed executable the FirstThunk pointers will have been destroyed but can sometimes be rebuilt by copying the duplicate OriginalFirstThunks

	(which many
--	-------------

simple packers do not seem to bother removing). There is actually a utility called First_Thunk Rebuilder by Lunar_Dust which will do this. However, with Borland created files this is not possible because the OriginalFirstThunks are all zero and there is no INT:

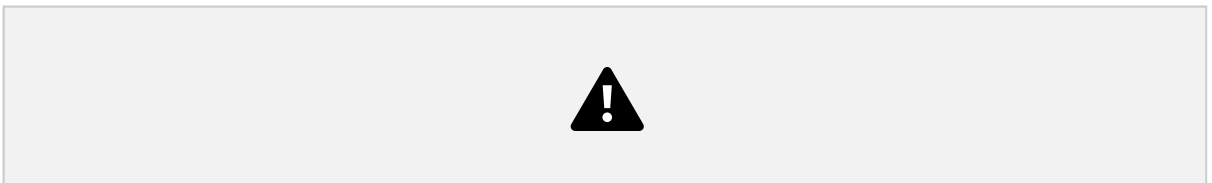


Back to our example above, the Name1 field of the first IMAGE_IMPORT_DESCRIPTOR contains the RVA 00 02 D5 30h (NB reverse byte order). Convert this to a raw offset by subtracting 2400h (remember above) and we have 2B130h. If we look there in our PE file we see the name of our DLL:



To continue, the FirstThunk field contains the RVA 00 02 D0 B4h which converts to Raw Offset 2ACB4h. Remember this is the offset to the array of DWORD-sized IMAGE_THUNK_DATA structures - the IAT. This will either have its most significant bit set (it will start with 8) and the lower part will contain the ordinal number of the imported function, or if the MSB is not set it will contain yet another RVA to the name of the function (IMAGE_IMPORT_BY_NAME).

In our file, the DWORD at 2ACB4h is 00 02 D5 3E:



This is another RVA which converts to Raw Offset 2B13E. This time it should be a null terminated ASCII string. In our file we see:



So the name of the first API imported from kernel32.dll is DeleteCriticalSection. You may notice the 2 zero bytes before the function name. This is the Hint element which is often set to 00 00.

All of this can be verified by using PEBrowse Pro to parse the IAT as shown:



If the file had been loaded into memory, dumped and examined with the hexeditor then the DWORD at RVA 2D0B4h which contained 3E D5 02 00 on disk would have been overwritten by the loader with the address of DeleteCriticalSection in kernel32.dll:



Allowing for reverse byte order this is 7C91188A.

IMPORTANT NOTE: functions in system DLLs always tend to start at the address 7XXXXXXXX and stay the same each time programs are loaded. However they tend to change if you reinstall your OS and differ from one computer to another.



The addresses also differ according to OS, for example:

OS Base of kernel32.dll

Win XP SP1 77E60000H

Win XP SP2 7C000000H

Win 2000 SP4 79430000H

Windows updates also sometimes change the base location of system DLLs. This is why some of you may have noticed that after taking the time to manually find point-h on your system it is prone to change unexpectedly since it is in a function inside user32.dll.

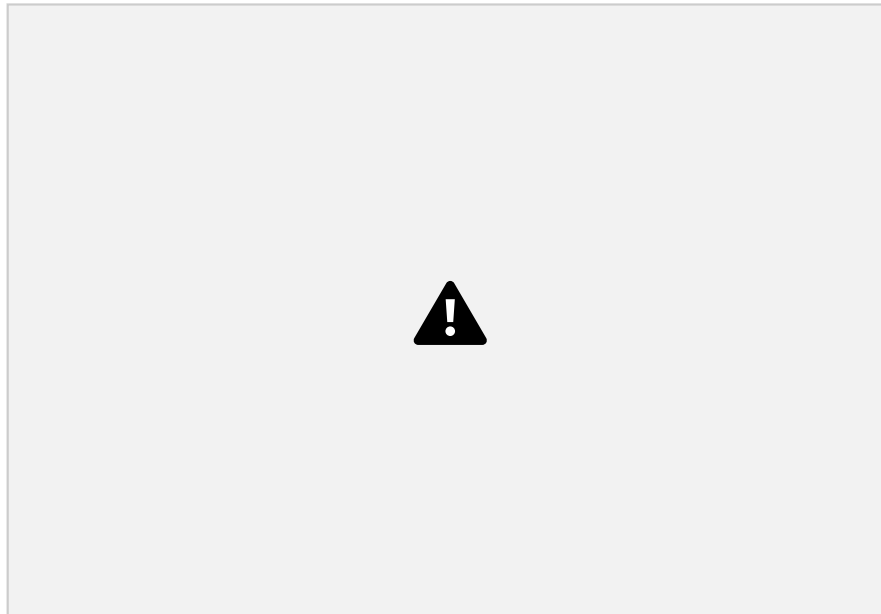
Navigating Imports in Memory

Load our example into Olly and again look at the Memory Map:



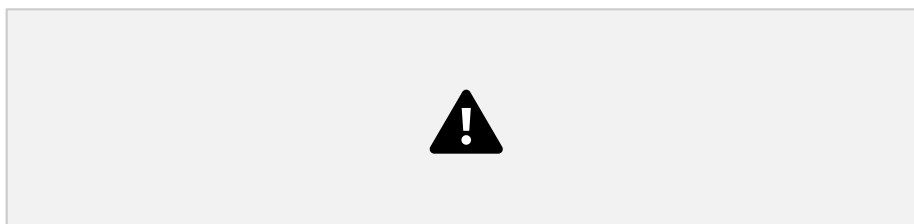
Note the address of the .idata section is 42D000 which corresponds to the RVA 2D000 shown at the top of this page as VOffset. The size has been rounded up to 2000 to fit memory page boundaries.

The main (CPU) window of Olly will only show the IAT if it lies in the executable CODE section (addresses 401000 to 42AFFF in our example), however in most cases it will be in its own section e.g. .idata. You can view the IAT in Olly's hex-dump window by rightclicking the appropriate section in the memory map and selecting Dump in CPU. Now rightclick in the hex window and select Long>Address and you will see the IAT in a readable list:

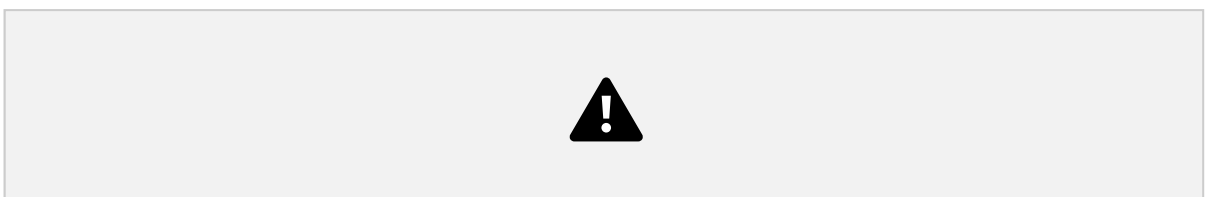


This makes finding the beginning and end of the IAT easy and is useful when using ImpREC as the IAT Autosearch function can be inaccurate. It is good to be able to check the beginning and endpoint to avoid having to type in a large size value which will give many false negatives with IAT Autosearch.

The names window (press Ctrl+N) will show you imported functions:



Rightclicking any of these and selecting Find References to Import will show you the jump thunk stub and the instances in the code where the function is called (only 1 in this case):



NOTE: in the comment column you will see that Olly has determined that the kernel32.dll function DeleteCriticalSection is actually forwarded to RtlDeleteCriticalSection in ntdll.dll (see [export forwarding](#) for explanation).

Rightclicking and selecting Follow Import in Disassembler will show you the address in the appropriate DLL where the function's code starts e.g. starts at 7C91188A in ntdll.DLL:



If we look at the call to DeleteCriticalSection at 00401B12 we see this:



This is really "CALL 00401314" but Olly has already substituted the function name for us. 401314 is the address of the jmp stub pointing to the IAT. Note it is part of a jmp thunk table as described previously:



This is really "JMP DWORD PTR DS:[0042D0B4]" but again Olly has substituted the symbolic name for us. Address 0042D0B4 contains the Image_Thunk_Data structure in the IAT which has been overwritten by the loader with the actual address of the function in kernel32.DLL: 7C91188A. This is what we found earlier by rightclicking and selecting Follow Import in Disassembler and also from the dumped file above.

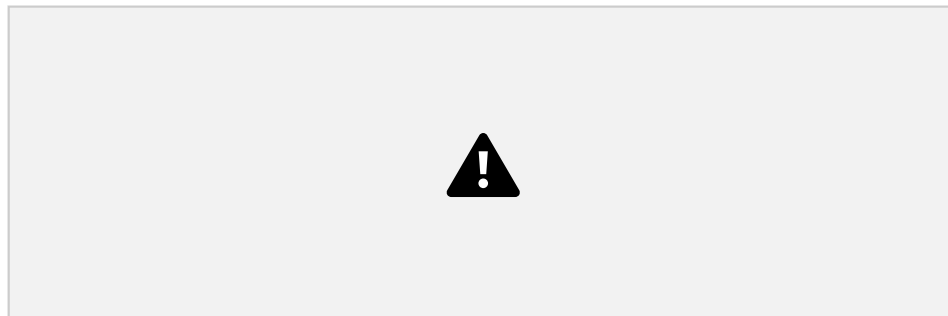
13. Adding Code to a PE File

It is often necessary to add code to a program in order to either crack a protection scheme or more usually to add functionality to it. There are 3 main ways to add code to an executable:

1. Add to an existing section when there is enough space for your code.
2. Enlarge an existing section when there is not enough space.
3. Add an entirely new section.

Adding to an existing section

We need a section in the file that is mapped with execution privileges in memory so the simplest is to try the CODE section. We then need an area in this section occupied by 00 byte padding. This is the concept of "caves". To find a suitable cave, look at the CODE Section details in LORDPE:



Here we see that the VirtualSize is slightly less than SizeOfRawData. The virtual size represents the amount of actual code. The size of raw data defines the amount of space taken up in the file sitting on your hard disk. Note that the virtual size in this case is lower than that on the hard disk. This is because compilers often have to round up the size to align a section on some boundary. In the hexeditor at the end of the code section (just before DATA section begins at 2A400h) we see:



This extra space is totally unused and not loaded into memory. We need to ensure that instructions we place there will be loaded into memory. We do this by altering the size attributes. Right now the virtual size of this section is only 29E88, because that is all the compiler needed. We need a little more, so in LordPE change the virtual size of the CODE section all the way up to 29FFF which is the max size we can use (the entire raw size is only 2A000). To do this rightclick the CODE line and select edit header, make the changes click save and enter.

Once that is done we have a suitable place to store our patch code. The only thing we have changed is the VirtualSize DWORD for the CODE section in the Section Table. We could have done this manually with the hexeditor.

To illustrate this further we will add to our example program a small ASM stub that hijacks the entrypoint and then just returns execution to the OriginalEntryPoint. We will do this in Olly.

First note in LordPE the EntryPoint is 0002ADB4 and ImageBase is 400000. When we load the app in Olly the EP will therefore be 0042ADB4. We will add the following lines and then change the entry point to the first line of code:

```
MOV EAX,0042ADB4 ; Load in EAX the Original Entry Point (OEP)
JMP EAX ; Jump to OEP
```

We will put them at 0002A300h as seen above in the hexeditor. To convert this raw offset to an RVA for use in Olly use the following formula (see appendix):

RVA = raw offset - raw offset of section + virtual offset of section + ImageBase
= 2A300h - 400h + 1000h + 400000h = **42AF00h**.

So load the app in Olly and jump to our target section (press Ctrl+G and enter 42AF00). Press space, type in the first line of code and click assemble. The next line down should now be highlighted so type in the second line of code and click assemble:



Now rightclick, select copy to executable and all modifications. Click copy all then a new window will open. Rightclick in the new window and select save file etc. Now back in LordPE (or hexeditor) change the EntryPoint to 0002AF00 (ImageBase subtracted) click save and then OK. Now run the app to test it and reopen it in Olly to see your new EntryPoint. In the hexeditor it looks like this - new code is highlighted:



Although this was only a tiny patch, we actually had room for 368 bytes of new code!

Enlarging an Existing Section

If there is not sufficient space at the end of the text section you will need to extend it. This poses a number of problems:

1. If the section is followed by other sections then you will need to move the following sections up to make room
2. There are various references within the file headers that will need to be adjusted if you change the file size.
3. References between various sections (such as references to data values from the code

section) will all need to be adjusted. This is practically impossible to do without re compiling and re-linking the original file.

Most of these problems can be avoided by appending to the last section in the exe file. It is not relevant what that section is as we can make it suit our needs by changing the Characteristics field in the Section Table either manually or with LordPE.

First we locate the final section and make it readable and executable. As we said earlier the code section is ideal for a patch because its characteristics flags are 60000020 which means code, executable and readable (see [appendix](#)). However if we were to put code and data into this section we would get a page fault since it is not writable. To alter this we would need to add the flag 80000000 which gives a new value of E0000020 for code, executable, readable and writable.

Likewise if the final section is .reloc then the flags will typically be 42000040 for initialised data, discardable and read-only. In order to use this section we must add code, executable and writable and we must subtract discardable to ensure that the loader maps this section into memory. This gives us a new value of E0000060.

This can either be done manually by adding up the flags and editing the Characteristics field of the Section header with your hexeditor or LordPE will do it. In our example the last section is Resources:



This gives us a final Characteristics value of F0000060. Above we see the RawSize (on disk) of this section is 8E00 bytes but all of this seems to be in use (the VirtualSize is the same). Now edit these and add 100h bytes to both to extend the section, the new value is 8F00h. There are some other important values which need to be changed. The SizeOfImage field in the PE header needs to be increased by the same amount from 0003CE00 to 0003CF00h.

There are 2 other fields which are not shown in LordPE which are less critical; SizeOfCode and SizeOfInitialisedData fields in the Optional Header. The app will still run without these being altered but you may wish to change them for completeness. We will have to edit these manually. Both are DWORDs at offsets 1C and 20 from the start of the PE header

	(see appendix):
--	----------------------------------



The values are 0002A000 and 0000DE00 respectively. Add 100h on to these to make 0002A100 and 0000DF00. With reverse byte order the values are: 00 A1 02 00 and 00 00 DF 00. Finally copy and paste 100h of 00 bytes (16 rows in the hexeditor) onto the end of the section and save changes. Run the file to test for errors.

Adding a New Section

In some circumstances you may need to make a copy of an existing section to defeat self checking procedures (such as in SafeDisk) or make a new section to hold code when proprietary information has been appended to the end of the file (as in Delphi compiled apps).

The first job is to find the NumberOfSections field in the PE header and increase it by 1. Again most of these changes can be made with LordPE or manually with your trusty hexeditor. Now in your hexeditor copy and paste 100h of 00 bytes (16 rows) onto the end of the file and make a note of the offset of the first new line. In our case it is 00038200h. This will be the start of our new section and will go in the RawOffset field of the section header. While we are here it is probably a good time to increase SizeOfImage by 100h bytes as we have done before.

Next we need to find the section headers beginning at offset F8 from the PE header. It is not necessary for these to be terminated by a header full of zeros. The number of headers is given by NumberOfSections and there is usually some space at the end before the sections themselves start (aligned to the FileAlignment value). Find the last section and add a new one after it:



The next thing we have to do is decide which Virtual Offset/Virtual Size/Raw Offset and Raw Size our section should have. To decide this, we need the following values:

Virtual offset of formerly last section (.rsrc): 34000h
Virtual size of formerly last section (.rsrc): 8E00h
Raw offset of formerly last section (.rsrc): 2F400h
Raw size of formerly last section (.rsrc): 8E00h
Section Alignment: 1000h
File Alignment: 200h

The RVA and raw offset of our new section must be aligned to the above boundaries. The Raw Offset of the section is 00038200h as we said above (which luckily fits with FileAlignment). To get the Virtual Offset of our section we have to calculate this: VirtualAddress of .rsrc + VirtualSize of .rsrc = 3CE00h. Since our SectionAlignment is 1000h we must round this up to the nearest 1000 which makes 3D000h. So let's fill the header of our section:

The first 8 bytes will be Name1 (max. 8 chars e.g. "NEW" will be 4E 45 57 00 00 00 00 00 (byte order not reversed))

The next DWORD is VirtualSize = 100h (with reverse byte order = 00 01 00 00) The next DWORD is VirtualAddress = 3D000h (with reverse byte order = 00 D0 03 00) The next DWORD is SizeOfRawData = 100h (with reverse byte order = 00 01 00 00) The next DWORD is PointerToRawData = 38200h (with reverse byte order = 00 82 03 00) The next 12 bytes can be left null

The final DWORD is Characteristics = E0000060 (for code, executable, read and write as discussed above)

In our hexeditor we see:



Save changes, run to test for errors and examine in LordPE:



14. Adding Import to an

Executable

This is most often used in the context of patching a target app where we don't have the API's we need. To recap, the minimum information needed by the loader to produce a valid IAT is:

1. Each DLL must be declared with an **IMAGE_IMPORT_DESCRIPTOR (IID)**, remembering to close the Import Directory with a null-filled one.
2. Each **IID** needs at least Name1 and FirstThunk fields, the rest can be set to 0 (setting OriginalFirstThunk = FirstThunk i.e. duplicating the RVAs also works).
3. Each entry of the FirstThunk must be an RVA to an Image_Thunk_Data (the IAT) which in turn contains a further RVA to the API name. The name will be a null terminated ASCII string of variable length and preceded by 2 bytes (**hint**) which can be set to 0.
4. If **IIDs** have been added then the **isize** field of the Import Table in the Data Directory may need changing. The IAT entries in Data Directory need not be altered (see import theory section).

Writing new import data in a hexeditor and then pasting into your target can be very time consuming. There are tools which can automate this process (e.g. SnippetCreator, IIDKing, Cavewriter - see bottom of page) but as always an understanding of how to do it manually is much better. The main task is to append a new IID onto the end of the import table - you need 20 bytes for each DLL used, not forgetting 20 for the null-terminator. In nearly all cases there will be no space at the end of the existing import table so we will make a copy and relocate it somewhere there is space.

Step 1 - create space for new a new IID

This involves the following steps:

- 1) Move all the IIDs to a location where there is plenty of space. This can be anywhere; the end of the current .idata section or an entirely new section.
- 2) Update the RVA of the new Import Directory in the Data Directory of the PE header.
- 3) If necessary, round up the size of the section where you've put the new Import Table so everything is mapped in memory (e.g. VirtualSize of the .idata section rounded up 1000h).
- 4) Run it and if it works proceed to step 2. If it doesn't check the injected descriptors are mapped in memory and that the RVA of the Import Directory is correct...

IMPORTANT NOTE: the IIDs, FirstThunk and OriginalFirstThunk contain RVAs - RELATIVE ADDRESSES - which means you can cut and paste the Import Directory (IIDs) wherever you want in your PE file (taking into account the destination has to be mapped into memory) and simply changing the RVA (and size if necessary) of the Import Directory in the Data Directory will make the app work perfectly.

Back to our example in the hexeditor, the first IID and the null terminator are outlined in red. As you can see there is no space after the null IID:



However there is a large amount of space at the end of the .idata section before .rdata starts. We will copy and paste the existing IIDs shown above to offset 2C500h at this new location:



To convert the new offset to an RVA (see [appendix](#)):

$$\text{VA} = \text{RawOffset} - \text{RawOffsetOfSection} + \text{VirtualOffsetOfSection}$$
$$= 2C500 - 2AC00 + 2D000 = \mathbf{2E900h}$$

So change the virtual address of the import table in the data directory from 2D000 to 2E900. Now edit the .idata section header and make VirtualSize equal to RawSize so the loader will map the whole section in. Run the app to test it.

Step 2 - Add the new DLL and function details

This involves the following steps:

- 1) Add null-terminated ASCII strings of the names of your DLL and function to a free space in the .idata section. The function name will actually be an Image_Import_By_Name structure preceded by a null WORD (the hint field).
- 2) Calculate the RVAs of the above strings.
- 3) Add the RVA of the DLL name to the Name1 field of your new IID.
- 4) Find another DWORD sized space and put in it the RVA of the hint/function name. This becomes the Image_Thunk_Data or IAT of our new DLL.
- 5) Calculate the RVA of the above Image_Thunk_Data DWORD and add it to the FirstThunk field of your new IID.
- 6) Run the app to test...your new API is ready to be called...

In order to fill in our new IID we need at the very least Name1 and FirstThunk fields (the others can be nulled). As we already know, the Name1 field contains the RVA of the name of the DLL in null-terminated ASCII. The FirstThunk field contains the RVA of an Image_Thunk_Data structure which in turn contains yet another RVA of the name of the function in null-terminated ASCII. The name however is preceded by 2 bytes (Hint) which can be set to zero.

Say for example we want to use the function LZCopy which copies a source file to a destination file. If the source file is compressed with the Microsoft File Compression Utility (COMPRESS.EXE), this function creates a decompressed destination file. If the source file is not compressed, this function duplicates the original file.

This function resides in lz32.dll which is not currently used by our app. Therefore we first need to add strings for the names "lz32.dll" and "LZCopy". Scroll upwards in the hexeditor from your new import table towards the end of the preexisting data and add the DLL name then the function name onto the end. Note the null bytes after each string and the null WORD before the function name:



Now we need to calculate the RVAs of these (see [appendix](#)):

$$\text{RVA} = \text{RawOffset} - \text{RawOffsetOfSection} + \text{VirtualOffsetOfSection} + \text{ImageBase}$$

RVA of DLL name = $2C420 - 2AC00 + 2D000 = 2E820h$ (20 E8 02 00 in reverse)
RVA of function name = $2C430 - 2AC00 + 2D000 = 2E830h$ (30 E8 02 00 in reverse)

The first one can go into the Name1 field of our new IID but the second must go into an Image_Thunk_Data structure, the RVA of which we can then put into the FirstThunk field (and OriginalFirstThunk) of our new IID. We will put the Image_Thunk_Data structure below the function name string at offset 2C440 and calculate the RVA which we will put in FirstThunk:

$$\text{RVA of Image_Thunk_Data} = 2C440 - 2AC00 + 2D000 = 2E840$$
 (40 E8 02 00 in reverse)

If we fill in the data in the hexeditor we see this:



Finally save changes, run the app to test and re-examine the imported functions in PEBrowse:



In order to call your new function, you would use the following code:

CALL DWORD PTR [XXXXXXXX] where XXXXXXXX = RVA of Image_Thunk_Data + ImageBase.

In our example above for LZCopy, XXXXXXXX = 2E840 + 400000 = 42E840 so we would write:

CALL DWORD PTR [0042E840]

FINAL NOTE: even if we had added a function used by a DLL which was already in use eg kernel32.dll, we would still need to create a new IID for it to enable us to create a new IAT at a convenient location as above.

Just as an addendum to this page, here are a few shots of the automated tools mentioned above:



Of note, SnippetCreator adds jump-thunk stubs of new imports to your code whereas with the other utilities you have to do this manually.





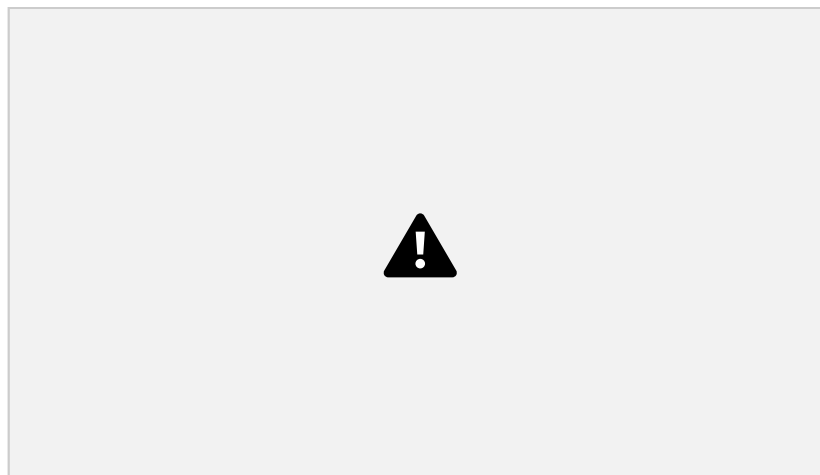
--	--

15. Introduction to Packers

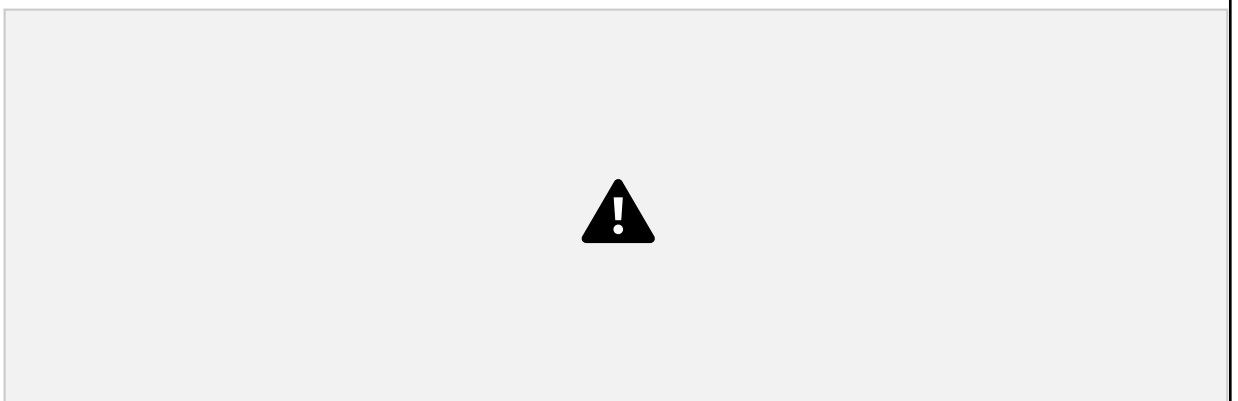
In this section we will examine the effect of a simple packer on our example app and cover 2 main ways of patching a packed executable - either by unpacking first or by inline patching. We will use UPX1.25 since this is really an executable compressor and doesn't use any advanced protection mechanisms. In the words of Marcus & Laszlo (the authors of UPX):

"We will ***NOT*** add any sort of protection and/or encryption. This only gives people a false feeling of security because by definition all protectors/compressors can be broken. And don't trust any advertisement of authors of other executable compressors about this topic - just do a websearch on 'unpackers'..."

First we scan our app with PEID:



Next we pack our app with upx. This is a commandline utility so we open a DOS box where our app is and type "upx basecalc.exe":



Now we notice file size down from 225Kb to 91 Kb and in PEID we see this:



PEBrowse Pro shows that there are now only 3 sections called UPX0, UPX1 and .rsrc. The resource section now contains the import directory but for each DLL there are only one or two imported functions - the others have disappeared:



Note the .rsrc section has retained its original name even though the others have changed. Interestingly this dates back to a bug in the LoadTypeLibEx function in oleaut32.dll in Win95 in which the string "rsrc" was used to find and load the resource section. This created an error if the section was renamed. Although this bug has been fixed it seems most packers do not rename the rsrc section for compatibility reasons.

By opening the app in LordPE editor and pressing the compare button we can open an original copy of our app and see the changes made to the headers:



When we open our app in Olly we get a message that the executable is likely packed.
Just click OK and we land at the entrypoint:



UPX has compressed our app and appended the code with a stub containing the decompression algorithm. The entrypoint of the app has been changed to the start of the stub and after the stub has done its job, execution jumps to the original entrypoint to start our now unpacked program.

The rationale for dealing with this is to let the stub decompress our app in memory and then dump the memory region to a file to get an unpacked copy of the app. However the app will not run straight away because the dumped file will have its sections aligned to memory page boundaries rather than file alignment values, the entrypoint will still point to the decompression stub and the Import directory is clearly also wrong and will need fixing.

Note at our entrypoint in Olly the first instruction we see is PUSHAD. This stands for PUSH All Double and instructs the CPU to store the contents of all the 32bit (DWORD) registers on the stack, starting with EAX and ending with EDI. Following this the stub does its job and then ends with a POPAD instruction before jumping to the OEP. POPAD copies the contents of the registers back from the stack. This means the stub will have restored everything back the way it was and exited without trace before running the app. Since this method is ideal in this situation it is common to other simple packers eg ASPack.

From the time of the first PUSHAD instruction, the contents of the stack at that level must remain untouched until accessed by the final POPAD. If we put a Hardware breakpoint on the first 4 bytes of the stack at the time of the PUSHAD Olly will break when the same 4 bytes are accessed at the POPAD instruction and we will be sitting right in front of our JMP to OEP.

First we must execute the PUSHAD instruction so press F7 to single step. Next we will place our breakpoint. The ESP (Stack Pointer) register always contains the location of the top of the stack so . Rightclick on ESP and select follow in dump - this puts the stack in the hexdump window:



Now highlight the first DWORD of the stack, rightclick and select breakpoint, hardware on access, DWORD:



Next run the app by pressing F9 and Olly will break after the PUSHAD directly before the JMP to the OEP. The OEP shown here has the ImageBase 400000h added onto it so to make it into an RVA we subtract it which leaves 0002ADB4h:

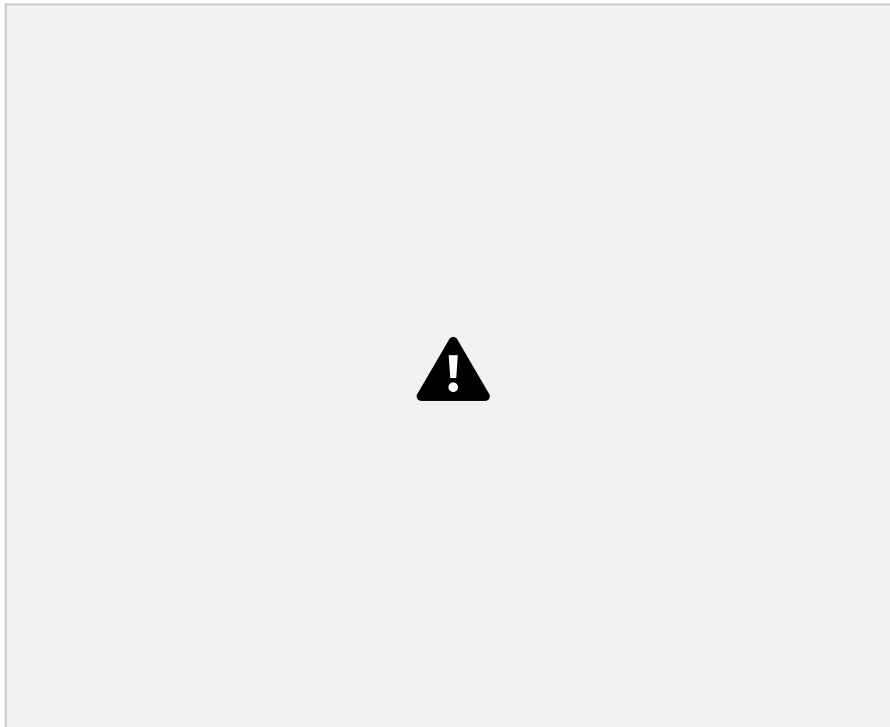


If you want to cheat there is a quick way which always works for upx. Simply scroll to the end of the code in the CPU window in Olly and just before all the zero padding starts you will see the POPAD instruction shown above.

NOTE: other packers which use the same simple PUSHAD/POPAD mechanism may jump to the OEP by using a PUSH instruction to put the value of the OEP onto the top of the stack followed by a RET instruction. The CPU will think it is returning from a function call and conventionally the return address is left on top of the stack.

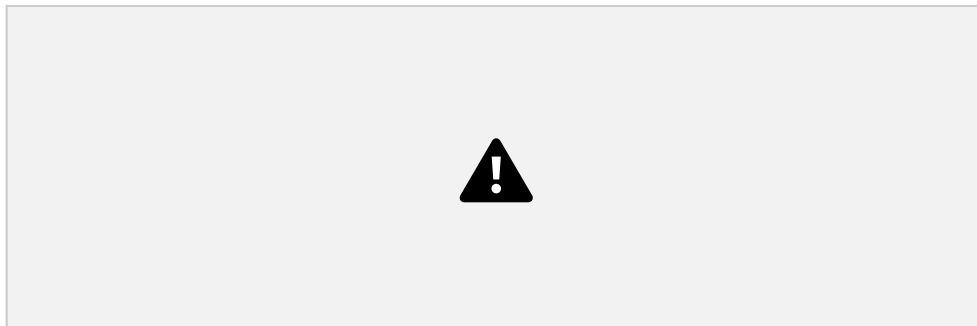
Next we single step once with F7 so we are at the OEP and dump the app using the OllyDump plugin. Just click on plugins, OllyDump and select dump debugged process. In

the next box we will deselect fix raw size and rebuild imports in order to illustrate some points of interest:



Note that OllyDump has already worked out the base address and size of image (which you could see by looking in the memory map window) and has offered to correct the entrypoint for us (although we could do this manually in the hexeditor). Press the DUMP button and save the file (eg as basecalc_dmp.exe). Leave Olly running for now.

Unfortunately we see something is wrong because our file has lost its icon and if we try to run it we get an error:



This is because of the alignment issues mentioned earlier - the filesize has also increased as a result. Open the app in LordPE and look at the sections. The raw offset and raw size values are wrong. We will have to make the Raw values equal the Virtual values for each section for the app to work. Rightclick the UPX0 section and select edit header:



Now make RawOffset equal VirtualAddress and RawSize equal VirtualSize. Repeat for the other sections then click save and exit (this is what the "fix raw size" checkbox in OllyDump does automatically). Now the icon has returned and we get a different error when we try to run it: "The application failed to initialize properly". This is because the imports still need rebuilding.

It is possible to do this manually using a process similar to adding imports which we discussed in a previous section. However this can be very time-consuming if there are a lot of imported functions and the method depends on how damaged the import data is. Here we will use ImpREC 1.6F by MackT to do this automatically. ImpREC needs to attach to a running process and also needs the packed file to find imports. Start up ImpREC and follow these steps:

1. select basecalc.exe in the box at the top (it should still be running in Olly.)
2. Next enter our OEP (2ADB4) in the appropriate box
3. Press the "IAT AutoSearch" button and click OK on the messagebox
4. Press the "Get Imports" button
5. Press "Show Invalid" - in this case there are none
6. Press "Fix Dump" and select basecalc_dmp.exe in the open dialogbox
7. Exit.

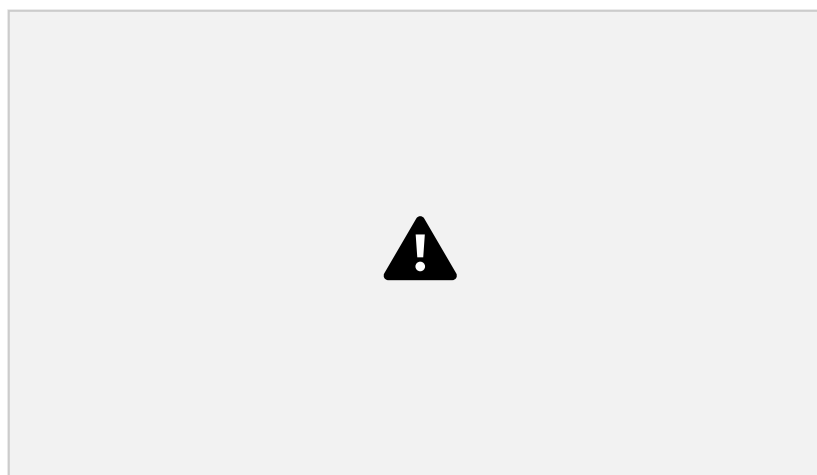


ImpREC will save a fixed copy of our dumped file appended with "_" so run `basecalc_dmp_.exe` to test it. If we examine this file we will see that size has increased and there is an extra section called "mackt" - this is where ImpREC puts the new import data:



Since UPX is purely a compressor, it has simply taken the existing import data and stored it in the resource section without encrypting or damaging it. This is why ImpREC finds all valid imports without resorting to tracing or rebuilding - it has taken the import directory from the packed executable in memory and transferred it to the new section in the unpacked executable.

Scanning with PEID now reveals:

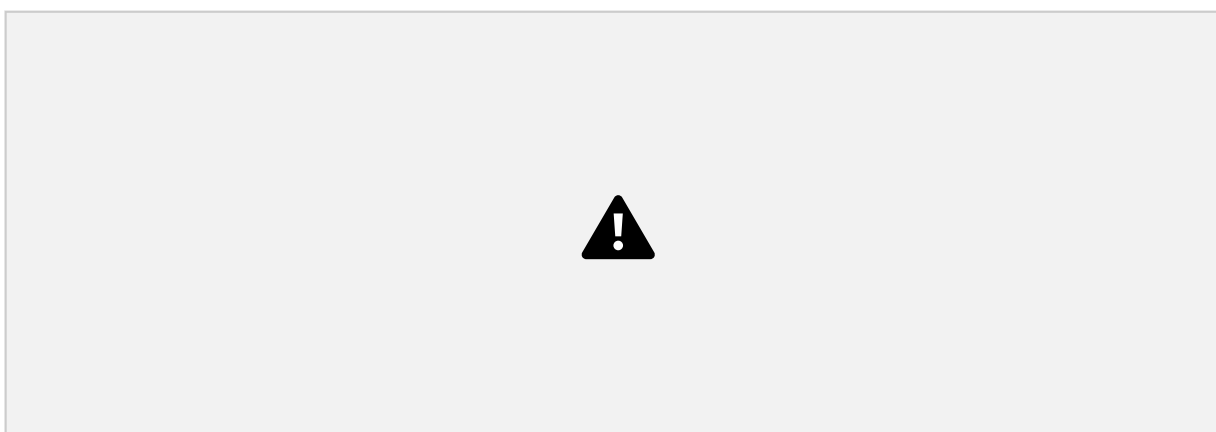


This illustrates the steps necessary to unpack an executable packed with a simple compressor. More advanced packers add various protection schemes to this eg antidebugging and anti-tampering tricks, encryption of code and IAT, stolen bytes, API redirection, etc. which are beyond the scope of this tutorial.

If it is necessary to patch a packed executable, it may be possible to avoid unpacking it first by using a technique called "inline-patching". This involves patching the code at runtime in memory after the decompression stub has done its work and then finally jumping to the OEP to run the app. In other words we wait until the app is unpacked in memory, jump to patching code which we have injected, then finally jump back to the OEP.

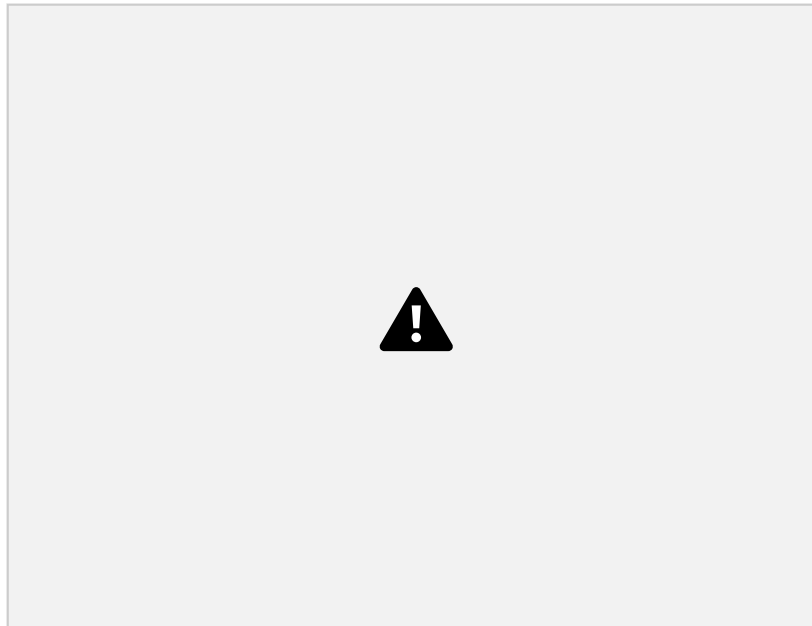
To illustrate this technique we will inject code into the packed executable to pop up a messagebox and let us know when the app is unpacked in memory. Clicking OK will then jump to the OEP and the app will run normally.

The first task is to find some free space for our code so open the packed app in the hexeditor and look for a suitable "cave". Free space at the end of a section is better as it is less likely to be used by the packer and is extensible by enlarging the section if necessary (see [adding code to a PE file.](#)) You can see how efficient UPX is - there is hardly any free space - but a small cave exists here. Now add the text "Unpacked..." and "Now back to OEP" in the ASCII column of the hexeditor as shown:

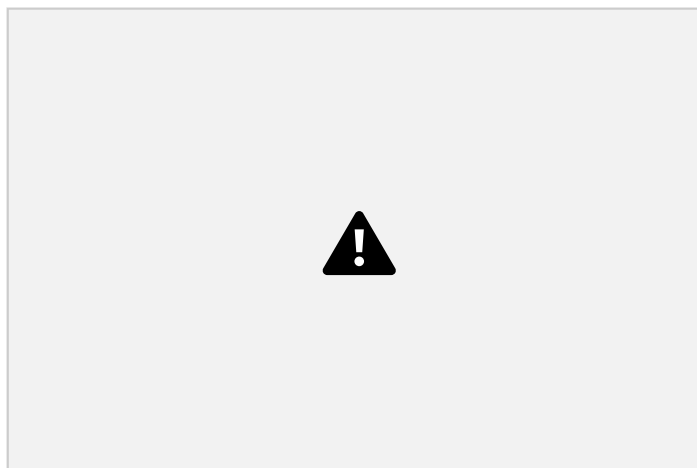


This will mark our spot for the patch in Olly without having to worry about calculating VAs. Save changes and open the app in Olly. Rightclick in the hex window and select search for binary string. Now enter "Unpacked" and note the VA of the 2 strings. In the CPU window, rightclick and select Goto expression. Enter the address of the first string and you will see the 2 strings in hexadecimal form. Olly has not analysed this properly so it displays nonsense code next to it. Highlight the next free row underneath and press the spacebar to assemble the following instructions:

```
PUSH 0 PUSH 440C30 [address of first string] PUSH 440C40 [address of second string]  
PUSH 0 CALL MessageBoxA JMP 42ADB4
```



Make a note of the address of our first PUSH instruction - 440C4E. Our code should look like this:



Next rightclick and select copy to executable, selection. In the new window rightclick and

select save file etc. If we check in the hexeditor we see our code has been added:



Finally we need to change the JMP at the end of the UPX stub to go to our code. Find it as shown earlier, doubleclick the JMP instruction to assemble and change the address to 440C4E. Save changes again and run the app to test it:



Clicking OK resumes BaseCalc.

