ARTeam Tutorial

http://cracking.accessroot.com/

http://forum.accessroot.com/

PORTABLE EXECUTABLE FILE FORMAT

Category	Relates to cracking, unpacking, reverse engineering
Level	Beginner to Intermediate
Test OS	XP Pro SP2
Author	Goppit

Tools Used:	
Hexeditor	(any will do)
PEBrowse Pro	http://www.smidgeonsoft.prohosting.com/download/PEBrowse.zip
PEID	http://www.secretashell.com/codomain/peid/download.html
LordPE	http://mitglied.lycos.de/yoda2k/LordPE/LPE-DLX.ZIP (get DLX-b update also)
HexToText	http://www.buttuglysoftware.com/HexToTextMFC.zip
OllyDbg	http://home.t-online.de/home/Ollydbg/odbg110.zip
OllyDump	http://ollydbg.win32asmcommunity.net/stuph/g_ollydump221b.zip
WinDbg	http://msdl.microsoft.com/download/symbols/debuggers/dbg_x86_6.4.7.2.exe
ResHacker	http://delphi.icm.edu.pl/ftp/tools/ResHack.zip
UPX 1.25	http://upx.sourceforge.net/download/upx125w.zip
ImpREC	http://wasm.ru/tools/6/imprec.zip
BaseCalc	included in this archive
...and mentioned in the text:	
Snippet Creator	http://win32assembly.online.fr/files/sc.zip

First_Thunk Rebuilder	http://www.angelfire.com/nt/teklord/FirstThunk.zip
IIDKing	http://www.reteam.org/tools/tf23.zip
Cavewriter	http://sandsprite.com/CodeStuff/cavewriter.zip
RVA Converter	http://www.polarhome.com:793/~execution/00/ex-rva11.zip
Offset Calculator	http://protools.reverse-engineering.net/files/utilities/offcal.zip

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.

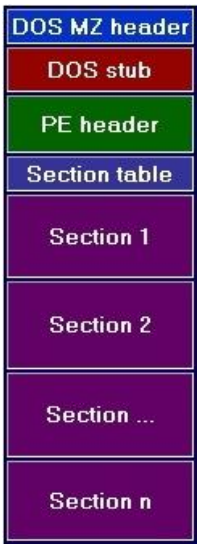
Why do we need to know about it? 2 main reasons. Adding code to executables (e.g. keygen injection or adding functionality) and manually unpacking executables. With respect to the latter, most shareware nowadays comes "packed" in order to reduce size and to provide an added layer of protection.

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).

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.

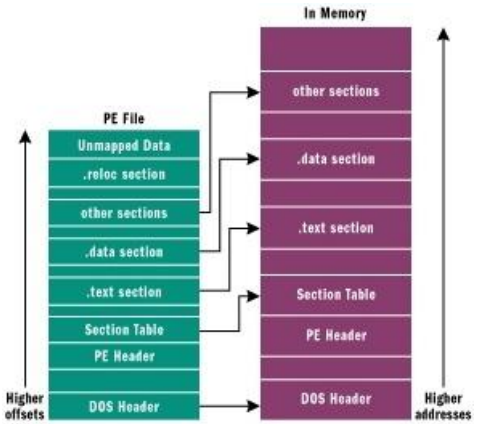
The sections that are most commonly present in an executable are:

- 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.

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 has19 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_lfarlc          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.)

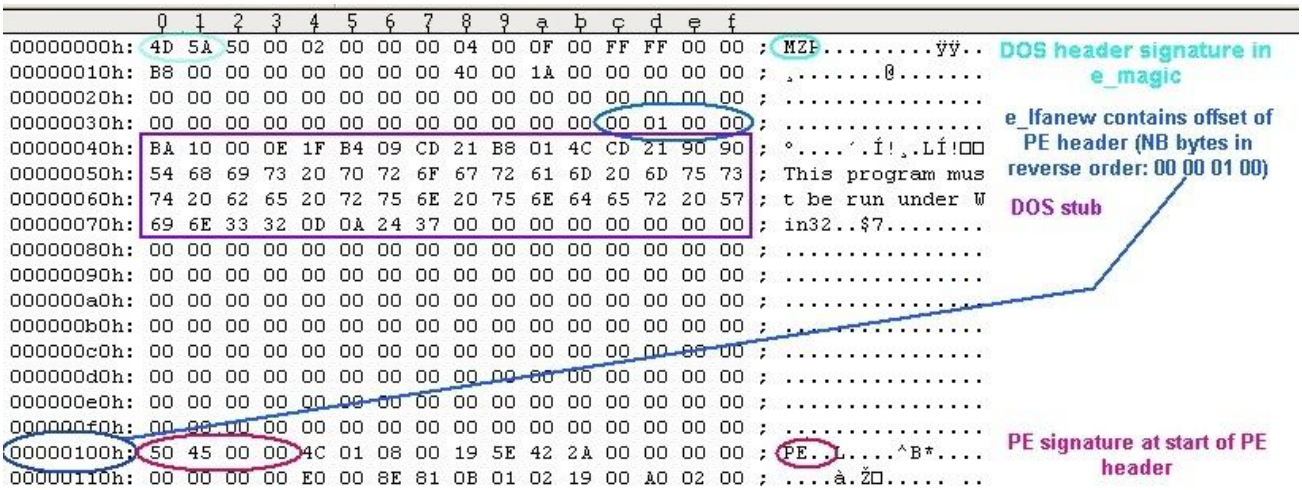
As we can see from its definition above, **lfanew** is a DWORD which sits at the end of the DOS header directly before the DOS stub begins. It contains the offset of the PE header, relative to the file beginning. The windows loader looks for this offset so it can skip the DOS stub and go directly to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



We will discuss this in the next section.

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

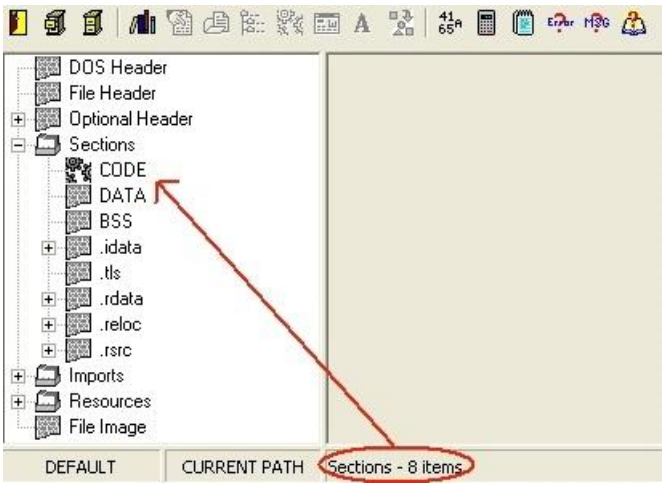
FileHeader is defined as follows:

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

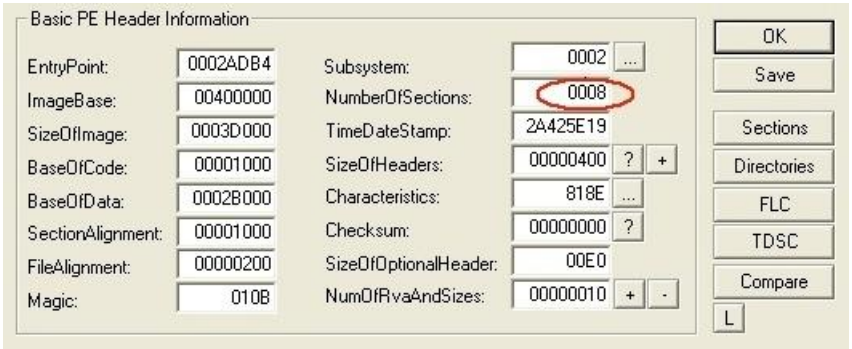
Most of these members are not of use to us but we must modify **NumberOfSections** if we add or delete any sections in the PE file. **Characteristics** contains flags which dictate for instance whether this PE file is an executable or a DLL. Back to our example in the Hexeditor, we can find **NumberOfSections** 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.........ÿÿ..
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 00 01 00 ; ................
00000040h: BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90 ; º....´.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 E0 00 8E 81 0B 01 02 19 00 A0 02 00 ; ....à.Ž□..... ..
```

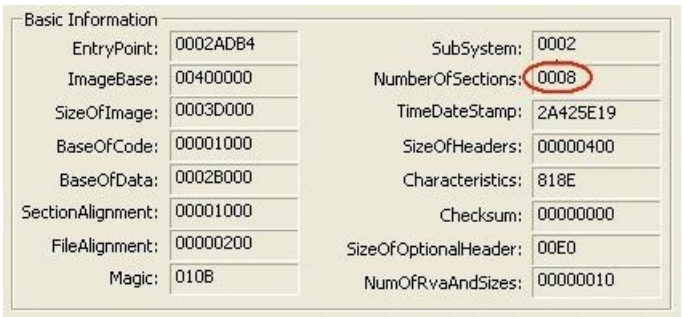
This can be verified by using any number of different (freeware) PE tools. For instance in PEBrowsePro:



Or in LordPE:



Or even from the "Subsystem" button of PEID:



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.

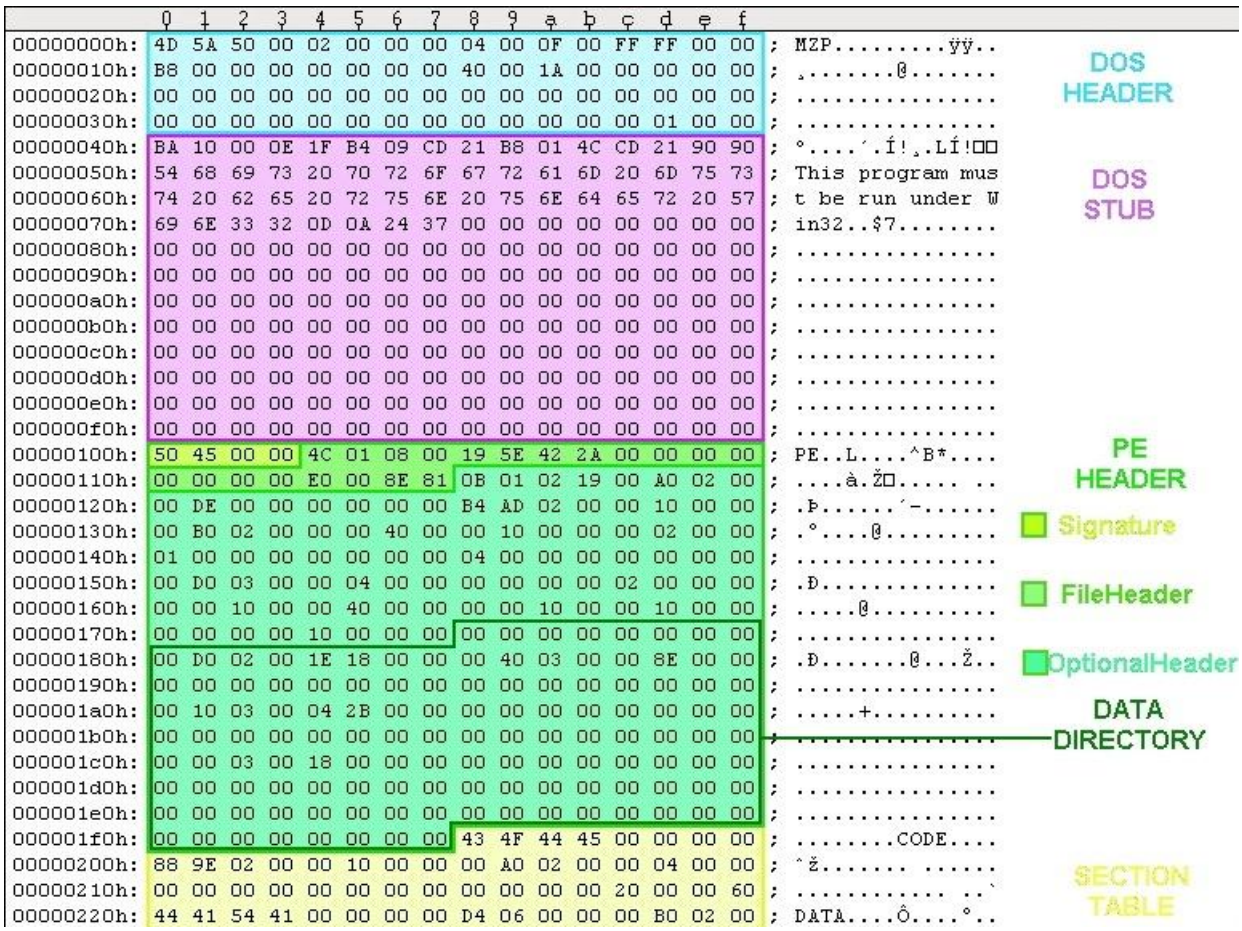
FileAlignment -- The granularity of the alignment of the sections in the file. For example, if the value in this field is 512 (200h), each section must start at multiples of 512 bytes. If the first section is at file offset 200h and the size is 10 bytes, the next section must be located at file offset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

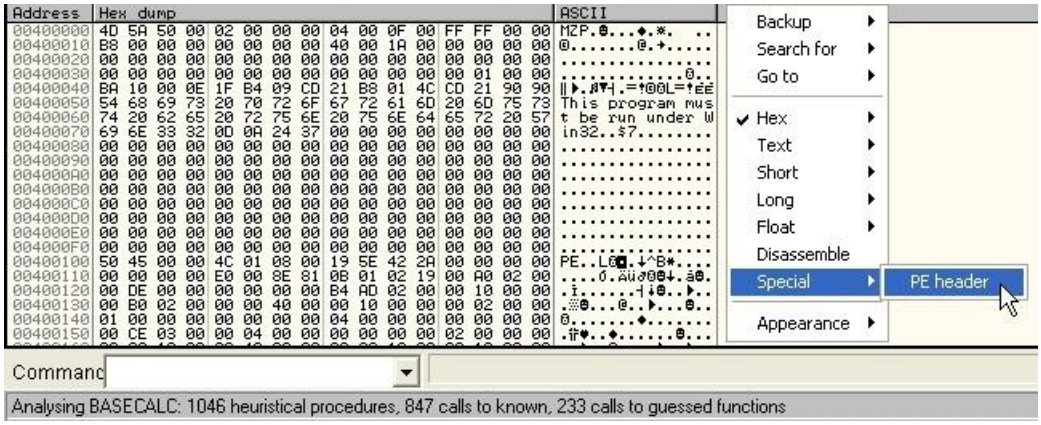
The overall layout of the PE Header can be seen from the following picture in the hexeditor. Note the DOS header and the parts of the PE header are always the same size (and shape). When viewed in the hexeditor, the DOS STUB can vary in size:



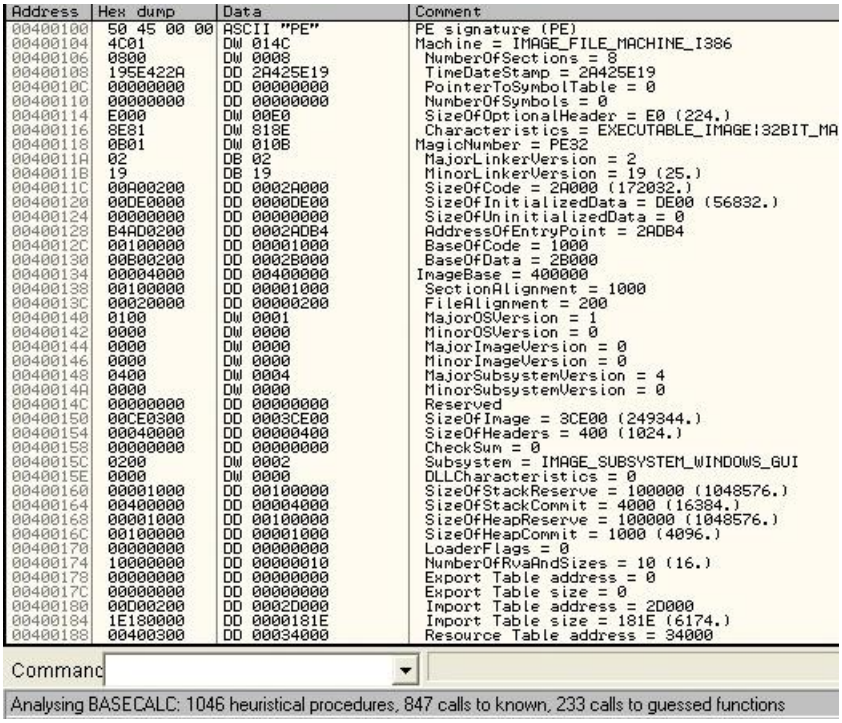
Besides the PE tools mentioned above, our favourite Ollydbg can also parse the PE headers into a meaningful display. Open our example in Olly and Press the M button or Alt+M to open the memory map - this shows how the PE file has been loaded into memory:

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00400000	00001000	BASECALC		PE header	Imag 01001002	R	RWE	
00401000	0002A000	BASECALC	CODE	code	Imag 01001002	R	RWE	
0042B000	00001000	BASECALC	DATA	data	Imag 01001002	R	RWE	
0042C000	00001000	BASECALC	BSS		Imag 01001002	R	RWE	
0042D000	00002000	BASECALC	.idata	imports	Imag 01001002	R	RWE	
0042F000	00001000	BASECALC	.tls		Imag 01001002	R	RWE	
00430000	00001000	BASECALC	.rdata		Imag 01001002	R	RWE	
00431000	00003000	BASECALC	.reloc	relocations	Imag 01001002	R	RWE	
00434000	00009000	BASECALC	.rsrc	resources	Imag 01001002	R	RWE	

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



Now you should see this:



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:

```
IMAGE_DATA_DIRECTORY STRUCT
  VirtualAddress    DWORD      ?
  isize             DWORD      ?
IMAGE_DATA_DIRECTORY ENDS
```

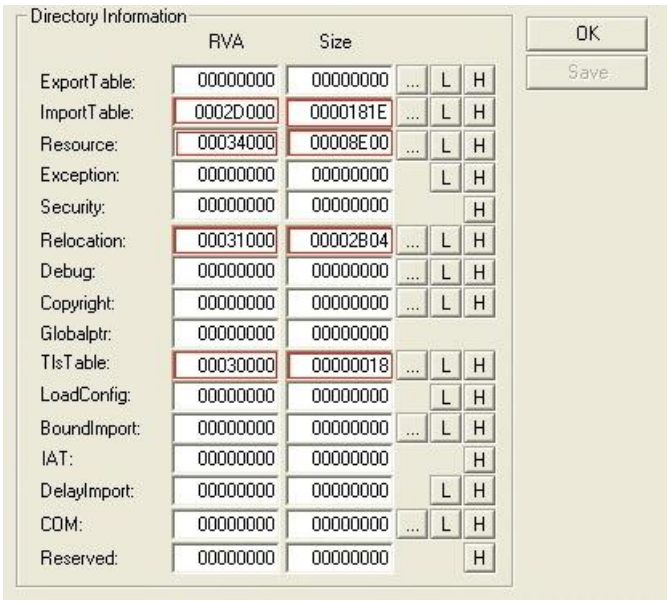
VirtualAddress is the relative virtual address (RVA) of the data structure (see later section).

isize contains the size in bytes of the data structure.

The 16 directories to which these structures refer are themselves defined in windows.inc:

```
IMAGE_DIRECTORY_ENTRY_EXPORT              equ  0
IMAGE_DIRECTORY_ENTRY_IMPORT              equ  1
IMAGE_DIRECTORY_ENTRY_RESOURCE            equ  2
IMAGE_DIRECTORY_ENTRY_EXCEPTION           equ  3
IMAGE_DIRECTORY_ENTRY_SECURITY            equ  4
IMAGE_DIRECTORY_ENTRY_BASERELOC           equ  5
IMAGE_DIRECTORY_ENTRY_DEBUG               equ  6
IMAGE_DIRECTORY_ENTRY_COPYRIGHT           equ  7
IMAGE_DIRECTORY_ENTRY_GLOBALPTR           equ  8
IMAGE_DIRECTORY_ENTRY_TLS                 equ  9
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG         equ 10
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT        equ 11
IMAGE_DIRECTORY_ENTRY_IAT                 equ 12
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT        equ 13
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR      equ 14
IMAGE_NUMBEROF_DIRECTORY_ENTRIES          equ 16
```

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.**

```
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 A0 02 00 ; ....à.Ž□..... ..
00000120h: 00 DE 00 00 00 00 00 00 B4 AD 02 00 00 10 00 00 ; .Þ......´-......
00000130h: 00 B0 02 00 00 00 40 00 00 10 00 00 00 02 00 00 ; .°....@.........
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 ; .Ð..............
00000160h: 00 00 10 00 00 40 00 00 00 00 10 00 00 10 00 00 ; .....@..........
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 ; .Ð.......@...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 A0 02 00 00 04 00 00 ; ˆž....... ......
```

The 16 Data Directories

Import Table

Resource

Relocation

TLS Table

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.

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:

```
IMAGE_SECTION_HEADER STRUCT
    Name1                   BYTE        IMAGE_SIZEOF_SHORT_NAME dup(?)
    union Misc
        PhysicalAddress     DWORD       ?
        VirtualSize         DWORD       ?
    ends
    VirtualAddress          DWORD       ?
    SizeOfRawData           DWORD       ?
    PointerToRawData        DWORD       ?
    PointerToRelocations    DWORD       ?
    PointerToLinenumbers    DWORD       ?
    NumberOfRelocations     WORD        ?
    NumberOfLinenumbers     WORD        ?
    Characteristics         DWORD       ?
IMAGE_SECTION_HEADER ENDS

IMAGE_SIZEOF_SHORT_NAME equ 8
```

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.

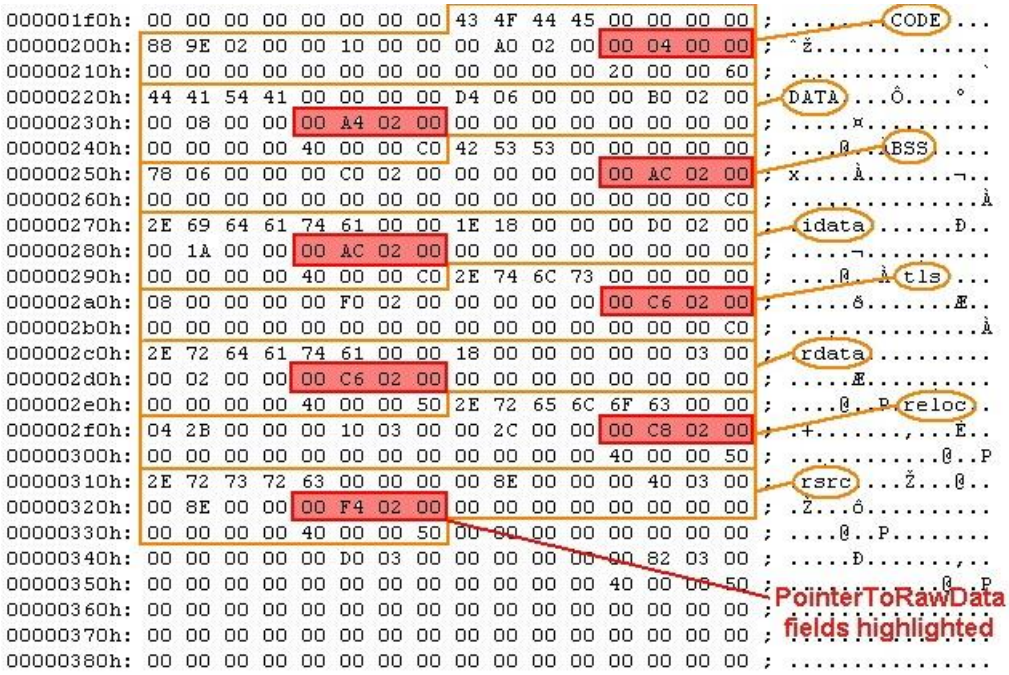
SizeOfRawData -- The size of the section's data in the file on disk, rounded up to the next multiple of file alignment by the compiler.

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 hexditor.

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.

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 section

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 sections

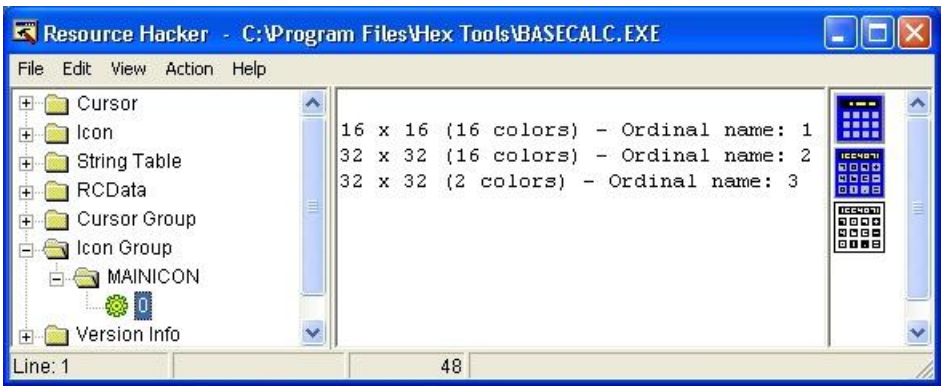
The **.bss** section represents uninitialized data for the application, including all variables declared as static within a function or source module.

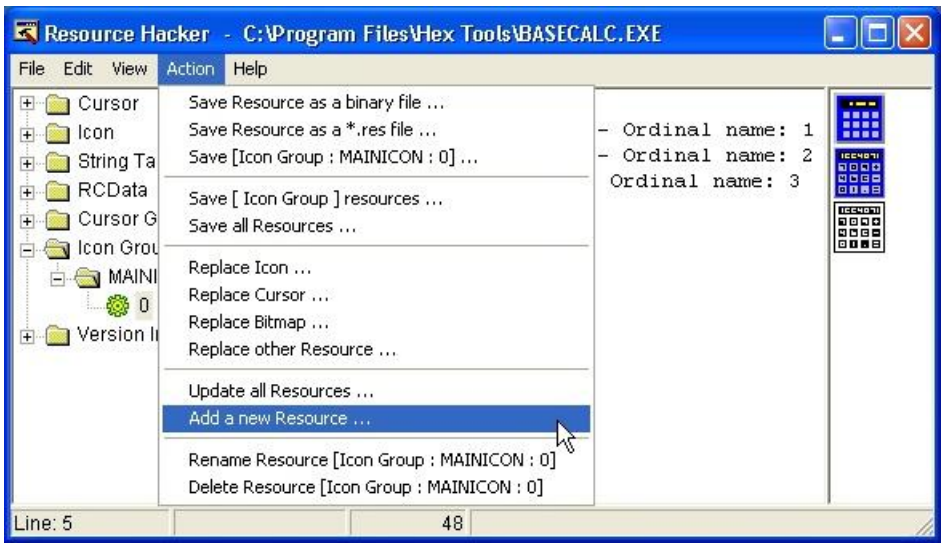
The **.rdata** section represents read-only data, such as literal strings, constants, and debug directory information.

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 section

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 section

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 section

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 section

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.

Base Relocations section

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.

Let's look at an example to see how base relocations work. 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.

The Export Section

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

In Microsoft® Windows®, dynamic-link libraries (DLL) are modules that contain functions and data. A DLL is loaded at runtime by its calling modules (.EXE or DLL). When a DLL is loaded, it is mapped into the address space of the calling process.

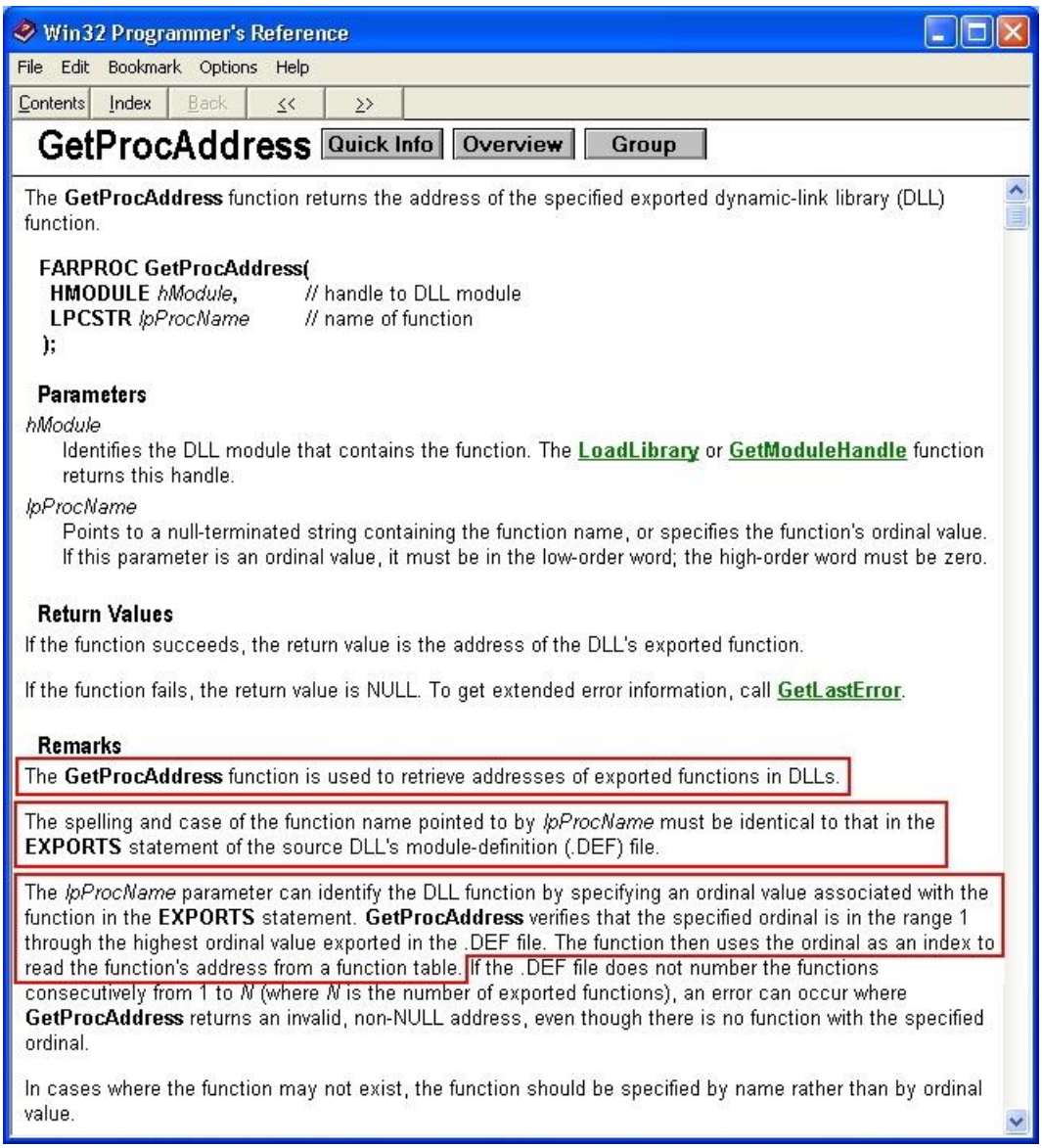
DLLs can define two kinds of functions: exported and internal. The exported functions can be called by other modules. Internal functions can only be called from within the DLL where they are defined. Although DLLs can export data, its data is usually only used by its functions.

DLLs provide a way to modularize applications so that functionality can be updated and reused more easilly. They also help reduce memory overhead when several applications use the same functionality at the same time, because although each application gets its own copy of the data, they can share the code.

The Microsoft® Win32® application programming interface (API) is implemented as a set of dynamic-link libraries, so any process using the Win32 API uses dynamic linking.

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:

```
IMAGE_EXPORT_DIRECTORY STRUCT
  Characteristics           DWORD      ?
  TimeDateStamp             DWORD      ?
  MajorVersion              WORD       ?
  MinorVersion              WORD       ?
  nName                     DWORD      ?
  nBase                     DWORD      ?
  NumberOfFunctions         DWORD      ?
  NumberOfNames             DWORD      ?
  AddressOfFunctions        DWORD      ?
  AddressOfNames            DWORD      ?
  AddressOfNameOrdinals     DWORD      ?
IMAGE_EXPORT_DIRECTORY ENDS
```

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).

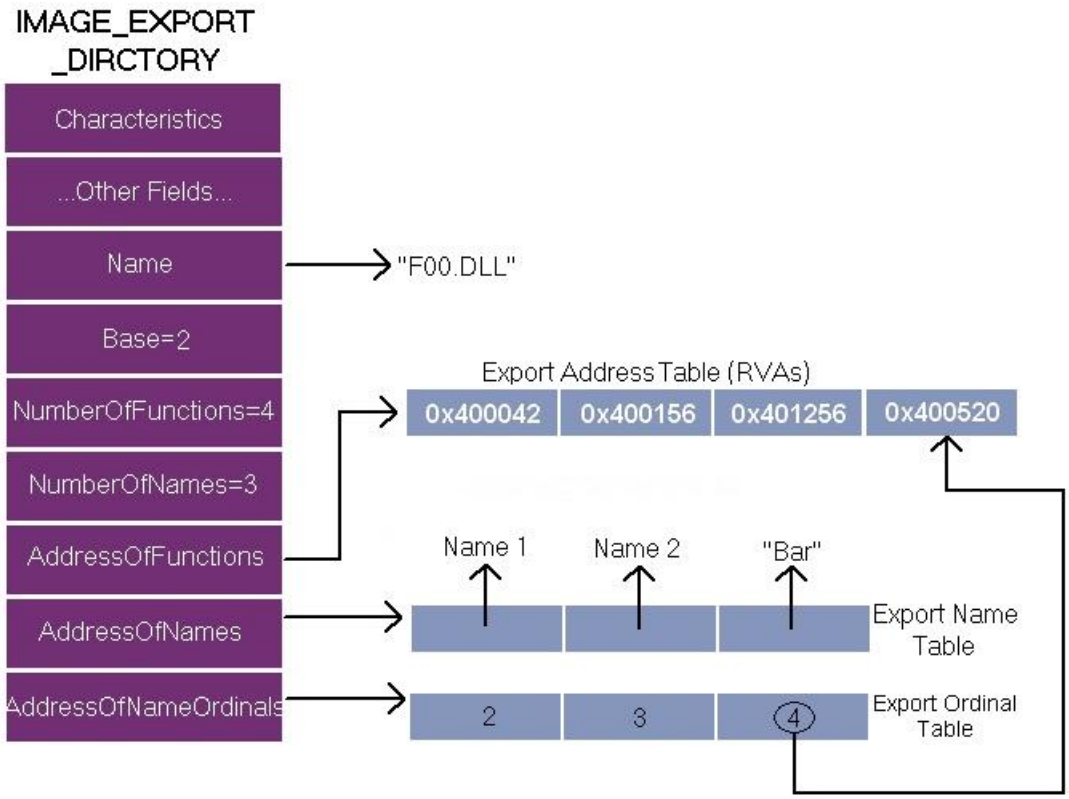
NumberOfFunctions -- Total number of functions (also referred to as symbols) that are exported by this module.

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.

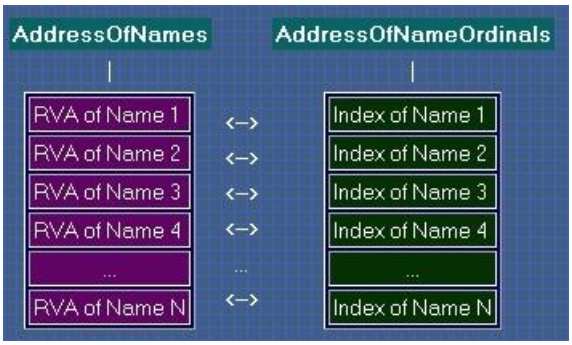
AddressOfFunctions -- An RVA that points to an array of pointers to (RVAs of) the functions in the module - the Export Address Table (EAT). To put it another way, the RVAs to all functions in the module are kept in an array and this field points to the head of that array.

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.



Since the EOT array exists as the linkage between the names and the addresses, it cannot contain more elements than the ENT array, i.e. each name can have one and only one associated address. The reverse is not true: an address may have several names associated with it. If there are functions with "aliases" that refer to the same address then the ENT will have more elements than the EOT.

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 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 be 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 that function is forwarded.

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:

```
IMAGE_IMPORT_DESCRIPTOR STRUCT
    union
        Characteristics     DWORD      ?
        OriginalFirstThunk  DWORD      ?
   ends
    TimeDateStamp           DWORD      ?
    ForwarderChain          DWORD      ?
    Name1                   DWORD      ?
    FirstThunk              DWORD      ?
IMAGE_IMPORT_DESCRIPTOR ENDS
```

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:

```
IMAGE_THUNK_DATA32 STRUCT
    union u1
        ForwarderString DWORD       ?
        Function        DWORD       ?
        Ordinal         DWORD       ?
        AddressOfData   DWORD       ?
    ends
IMAGE_THUNK_DATA32 ENDS
```

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:

```
IMAGE_IMPORT_BY_NAME STRUCT
    Hint     WORD      ?
    Name1    BYTE      ?
IMAGE_IMPORT_BY_NAME ENDS
```

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.

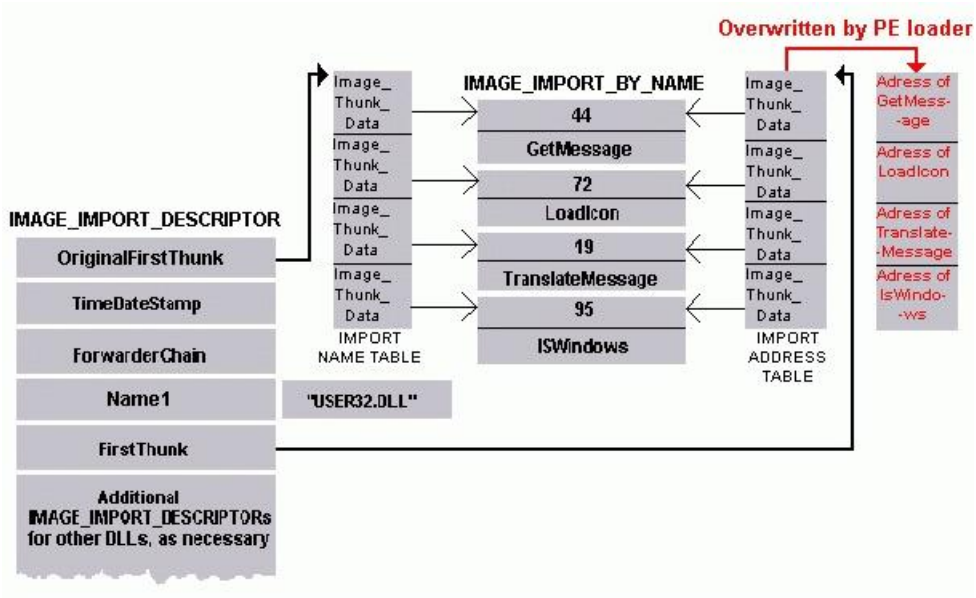
The number of elements in the OriginalFirstThunk and FirstThunk arrays depends on the number of functions imported from the DLL. For example, if the PE file imports 10 functions from user32.dll, Name1 in the IMAGE_IMPORT_DESCRIPTOR structure will contain the RVA of the string "user32.dll" and there will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

```
IMAGE_BOUND_IMPORT_DESCRIPTOR STRUCT
    TimeDateStamp                DWORD    ?
    OffsetModuleName             WORD     ?
    NumberOfModuleForwarderRefs  WORD     ?
IMAGE_BOUND_IMPORT_DESCRIPTOR ENDS
```

The **TimeDateStamp** 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:

```
IMAGE_BOUND_FORWARDER_REF STRUCT
    TimeDateStamp     DWORD    ?
    OffsetModuleName  WORD     ?
    Reserved          WORD     ?
IMAGE_BOUND_FORWARDER_REF ENDS
```

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.

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.

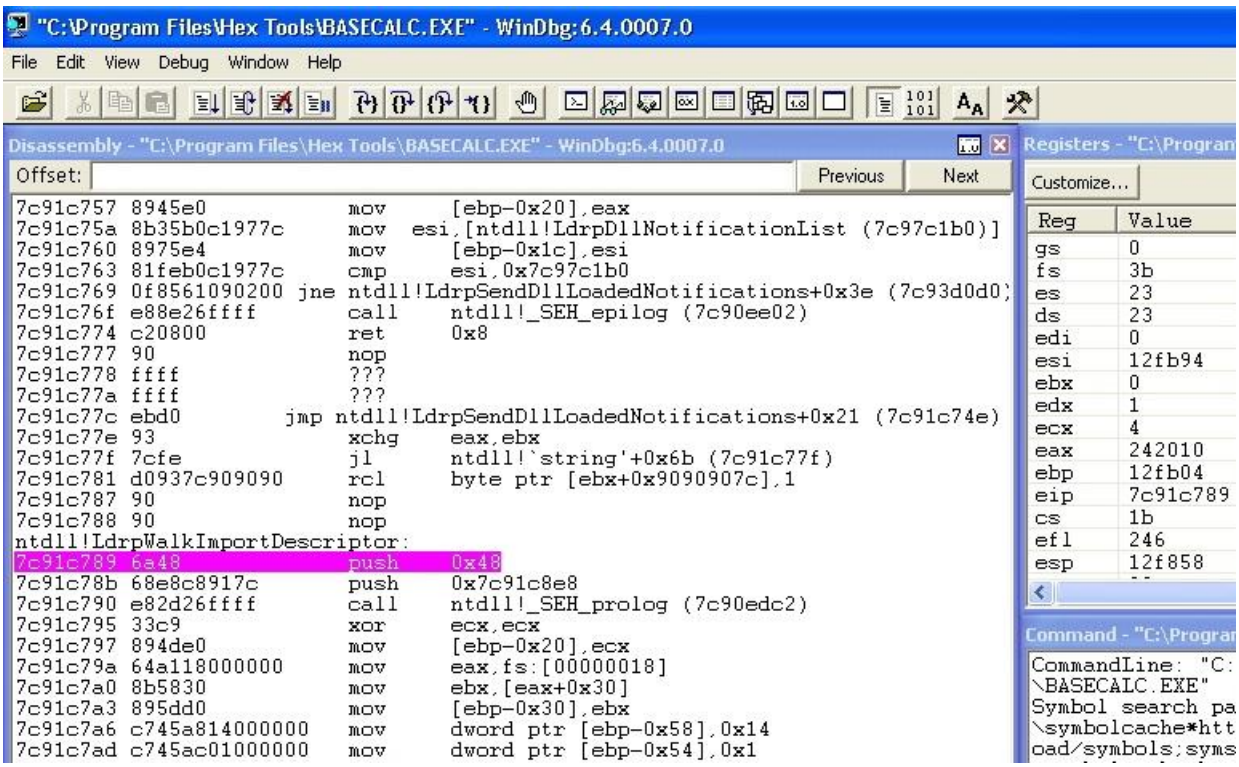
What the loader does

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 and maps the sections in memory. 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 Micro$oft. As we said previously function forwarding is a way for M$ 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.

```
LdrLoadDll (0x77f889a9)
    LdrpLoadDll 0x77f887e0
        LdrpCheckForLoadedDll (0x77f87122)
        LdrpMapDll (0x77f8bc77)
            LdrpCheckForKnownDll (0x77f8c62b)
            LdrpResolveDllName (0x77f8c3df)
            LdrpCreateDllSection (0x77f8c355)
            LdrpAllocateDataTableEntry (0x77f8be69)
            LdrpFetchAddressOfEntryPoint (0x77f8bf23)
            LdrpInsertMemoryTableEntry (0x77f8bebb)
        LdrpWalkImportDescriptor (0x77f8be15)
            LdrpLoadImportModule (0x77f8bfd1)
                *LdrpCheckForLoadedDll
            LdrpSnapIAT (0x77f8c047)
                LdrpSnapThunk (0x77f87bd1)
                    LdrpNameToOrdinal (0x77f87cf0)
                    **LdrpLoadDll
                    LdrpGetProcedureAddress (0x77f87a20)
                        LdrpCheckforLoadedDllHandle (0x77f870cc)
                        **LdrpSnapThunk
        LdrpUpdateLoadCount (0x77f88afa)
            *LdrpCheckForLoadedDll
            **LdrpUpdateLoadCount
        LdrpRunInitializeRoutines (0x77f8bcb8)
        LdrpClearLoadInProgress (0x77f88c12)
```

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.

Navigating Imports

Navigating Imports on Disk

If you want to look up information about the imported function "foo" from DLL "bar", you first find the RVA of the Import Directory from the Data Directory, find that address in the raw section data and now you have an array of IMAGE_IMPORT_DESCRIPTORs. Get the member of this array that relates to bar.dll by inspecting the strings pointed to by the 'Name' fields. When you have found the right IMAGE_IMPORT_DESCRIPTOR, follow its 'FirstThunk' and get hold of the pointed-to array of IMAGE_THUNK_DATAs; inspect the RVAs and find the function "foo".

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** DWORD**.** In our example the offset is 2AC00h (see section table page). Any PE Editor will show this, e.g. LordPE:

Name	VOffset	VSize	ROffset	RSize	Flags
CODE	00001000	00029E88	00000400	0002A000	60000020
DATA	0002B000	000006D4	0002A400	00000800	C0000040
BSS	0002C000	00000678	0002AC00	00000000	C0000000
.idata	0002D000	0000181E	0002AC00	00001A00	C0000040
.tls	0002F000	00000008	0002C600	00000000	C0000000
.rdata	00030000	00000018	0002C600	00000200	50000040
.reloc	00031000	00002B04	0002C800	00002C00	50000040
.rsrc	00034000	00008E00	0002F400	00008E00	50000040

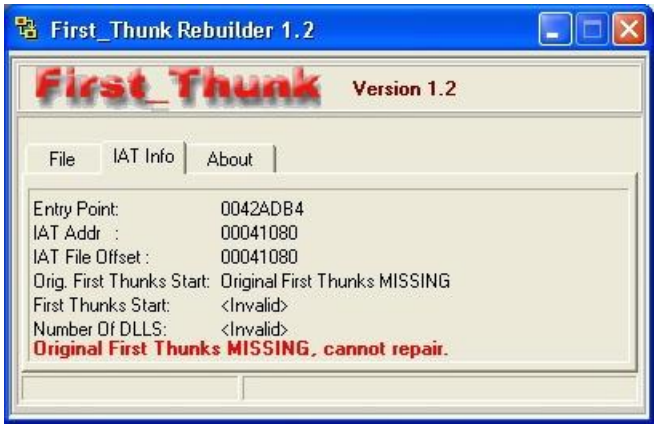
The difference between the RVA and Raw Offset is 2D000-2AC00=2400h. Make a note of this as it will be useful for converting further offsets. See appendix for more info on converting RVAs.

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:

```
0002ac00h: 00 00 00 00 00 00 00 00 00 00 00 00 30 D5 02 00 ; ............0O..
0002ac10h: B4 D0 02 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ´Ð..............
0002ac20h: 06 D7 02 00 24 D1 02 00 00 00 00 00 00 00 00 00 ; .×..$Ñ..........
0002ac30h: 00 00 00 00 20 D7 02 00 2C D1 02 00 00 00 00 00 ; .... ×..,Ñ......
0002ac40h: 00 00 00 00 00 00 00 00 8A D7 02 00 44 D1 02 00 ; ........Š×..DÑ..
0002ac50h: 00 00 00 00 00 00 00 00 00 00 00 00 FC D7 02 00 ; ............ü×..
0002ac60h: 60 D1 02 00 00 00 00 00 00 00 00 00 00 00 00 00 ; `Ñ..............
0002ac70h: 4E DA 02 00 F0 D1 02 00 00 00 00 00 00 00 00 00 ; NÚ..ðÑ..........
0002ac80h: 00 00 00 00 30 DE 02 00 D4 D2 02 00 00 00 00 00 ; ....0Þ..ÔÒ......
0002ac90h: 00 00 00 00 00 00 00 00 F6 E6 02 00 FC D4 02 00 ; ........öæ..üÔ..
0002aca0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002acb0h: 00 00 00 00 3E D5 02 00 56 D5 02 00 6E D5 02 00 ; ....>Õ..VÕ..nÕ..
0002acc0h: 86 D5 02 00 A2 D5 02 00 B0 D5 02 00 C0 D5 02 00 ; †Õ..¢Õ..°Õ..ÀÕ..
0002acd0h: CC D5 02 00 DA D5 02 00 F0 D5 02 00 FE D5 02 00 ; ÌÕ..ÚÕ..ðÕ..þÕ..
```

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:

```
0002b110h: 86 E7 02 00 9C E7 02 00 B0 E7 02 00 C6 E7 02 00 ; †ç..œç..°ç..Æç..
0002b120h: DE E7 02 00 F6 E7 02 00 0A E8 02 00 00 00 00 00 ; Þç..öç...è......
0002b130h: 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00 ; kernel32.dll....
0002b140h: 44 65 6C 65 74 65 43 72 69 74 69 63 61 6C 53 65 ; DeleteCriticalSe
0002b150h: 63 74 69 6F 6E 00 00 00 4C 65 61 76 65 43 72 69 ; ction...LeaveCri
0002b160h: 74 69 63 61 6C 53 65 63 74 69 6F 6E 00 00 00 00 ; ticalSection....
```

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:

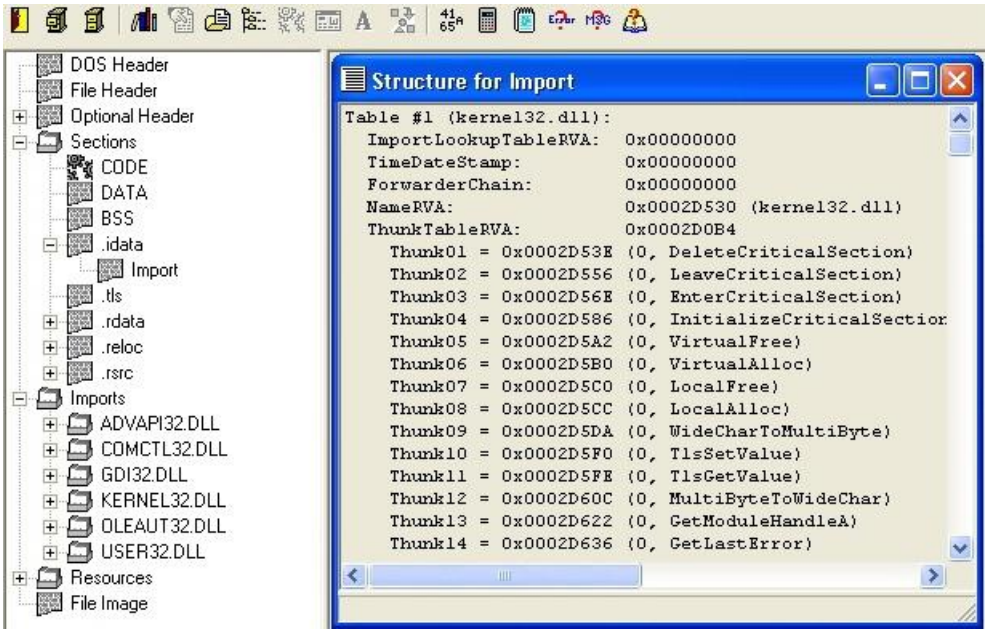
```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0002ac90h: 00 00 00 00 00 00 00 00 F6 E6 02 00 FC D4 02 00 ; ........öæ..üÔ..
0002aca0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002acb0h: 00 00 00 00 3E D5 02 00 56 D5 02 00 6E D5 02 00 ; ....>Õ..VÕ..nÕ..
0002acc0h: 86 D5 02 00 A2 D5 02 00 B0 D5 02 00 C0 D5 02 00 ; †Õ..¢Õ..°Õ..ÀÕ..
0002acd0h: CC D5 02 00 DA D5 02 00 F0 D5 02 00 FE D5 02 00 ; ÌÕ..ÚÕ..ðÕ..þÕ..
```

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:

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0002b120h: DE E7 02 00 F6 E7 02 00 0A E8 02 00 00 00 00 00 ; Þç..öç...è......
0002b130h: 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00 ; kernel32.dll....
0002b140h: 44 65 6C 65 74 65 43 72 69 74 69 63 61 6C 53 65 ; DeleteCriticalSe
0002b150h: 63 74 69 6F 6E 00 00 00 4C 65 61 76 65 43 72 69 ; ction...LeaveCri
0002b160h: 74 69 63 61 6C 53 65 63 74 69 6F 6E 00 00 00 00 ; ticalSection....
```

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:

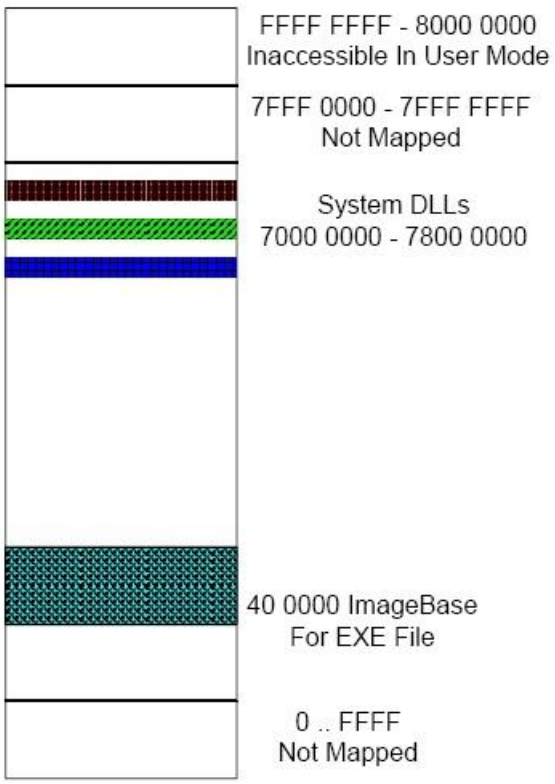


If the file had been loaded into memory, dumped and examined with the hexeditor then the DWORD at RVA 2D0B4h which contained 3E D5 02 00 on disk would have been overwritten by the loader with the address of DeleteCriticalSection in kernel32.dll:

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0002d0a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002d0b0h: 00 00 00 00 8A 18 91 7C ED 10 90 7C 05 10 90 7C ; ....Š.‘|í.□|..□|
0002d0c0h: A1 9F 80 7C 14 9B 80 7C 81 9A 80 7C 5D 99 80 7C ; ¡Ÿ€|.›€|□š€|]™€|
0002d0d0h: BD 99 80 7C C7 A0 80 7C F5 9B 80 7C 50 97 80 7C ; ½™€|Ç €|õ›€|P—€|
0002d0e0h: AD 9C 80 7C 29 B5 80 7C 31 03 91 7C 8D 2C 81 7C ; -œ€|)µ€|1.‘|□,□|
```

Allowing for reverse byte order this is 7C91188A.

IMPORTANT NOTE: functions in system DLLs always tend to start at the address 7XXXXXXX 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 the famous point-h breakpoint on your system it is prone to change unexpectedly since it is in a function inside user32.dll.

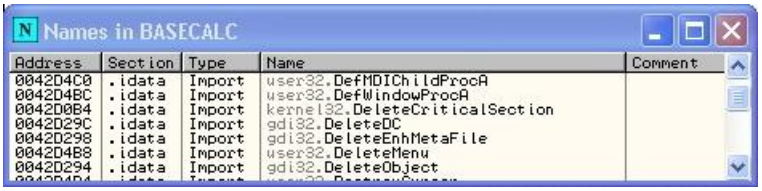
Navigating Imports in Memory

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

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00400000	00001000	BASECALC		PE header	Imag 01001002	R	RWE	
00401000	0002A000	BASECALC	CODE	code	Imag 01001002	R	RWE	
0042B000	00001000	BASECALC	DATA	data	Imag 01001002	R	RWE	
0042C000	00001000	BASECALC	BSS		Imag 01001002	R	RWE	
0042D000	00002000	BASECALC	.idata	imports	Imag 01001002	R	RWE	
0042F000	00001000	BASECALC	.tls		Imag 01001002	R	RWE	
00430000	00001000	BASECALC	.rdata		Imag 01001002	R	RWE	
00431000	00003000	BASECALC	.reloc	relocations	Imag 01001002	R	RWE	
00434000	00009000	BASECALC	.rsrc	resources	Imag 01001002	R	RWE	

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.

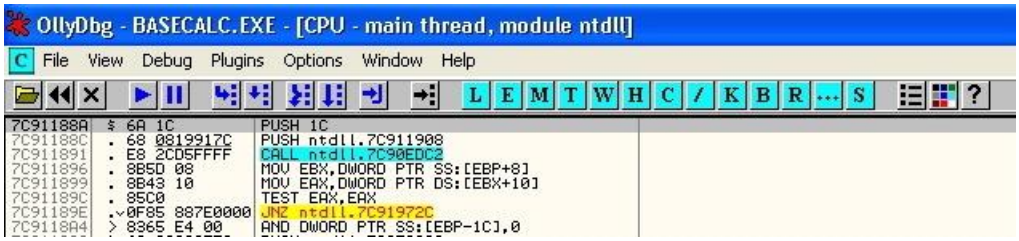
The main (CPU) window of Olly will only show the CODE section addresses (from 401000 to 42AFFF). You will only be able to examine the IAT in the disassembly window if it lies in the CODE section. In most cases it will be in its own section e.g. .idata but you can view this in the hex-dump window in Olly by rightclicking and selecting Dump in CPU. The names window (press Ctrl+N) will show you imported functions:



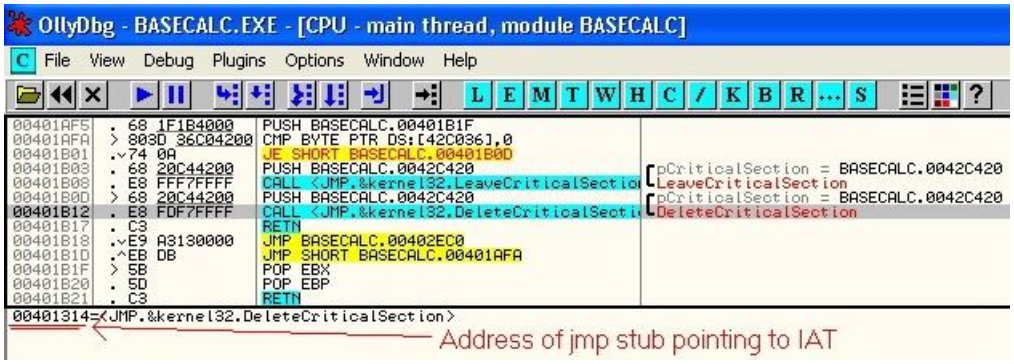
Rightclicking any of these and selecting Find References to Import will show you the jump thunk stub and the instances in the code where the function is called (only 1 in this case):

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).

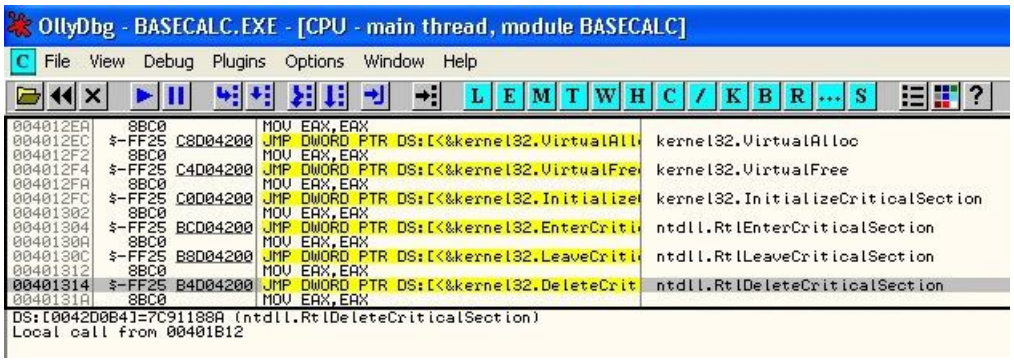
Rightclicking and selecting Follow Import in Disassembler will show you the address in the appropriate DLL where the function's code starts e.g. starts at 7C91188A in ntdll.DLL:



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.

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:

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
CODE	00001000	00029E88	00000400	0002A000	60000020
DATA	0002B000	000006D4	0002A400	00000800	C0000040
BSS	0002C000	00000678	0002AC00	00000000	C0000000
.idata	0002D000	0000181E	0002AC00	00001A00	C0000040
.tls	0002F000	00000008	0002C600	00000000	C0000000
.rdata	00030000	00000018	0002C600	00000200	50000040
.reloc	00031000	00002B04	0002C800	00002C00	50000040
.rsrc	00034000	00008E00	0002F400	00008E00	50000040

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:

```
0002a250h: 10 93 FD FF 8B E5 5D C3 FF FF FF FF 0F 00 00 00 ; .“ýÿ‹å]Ãÿÿÿÿ....
0002a260h: 42 61 73 65 20 43 61 6C 63 75 6C 61 74 6F 72 00 ; Base Calculator.
0002a270h: FF FF FF FF 0C 00 00 00 42 41 53 45 43 41 4C 43 ; ÿÿÿÿ....BASECALC
0002a280h: 2E 48 4C 50 00 00 00 00 00 00 00 00 00 00 00 00 ; .HLP............
0002a290h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a300h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a310h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a320h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a330h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a340h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a350h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a360h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a370h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a380h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a390h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a3a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a3b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a3c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a3d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a3e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a3f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a400h: 02 00 8B C0 00 8D 40 00 38 20 40 00 C0 21 40 00 ; ..‹À.□@.8 @.À!@.
0002a410h: 34 25 40 00 32 1F 8B C0 32 13 8B C0 52 75 6E 74 ; 4%@.2.‹À2.‹ÀRunt
0002a420h: 69 6D 65 20 65 72 72 6F 72 20 20 20 20 20 61 74 ; ime error     at
0002a430h: 20 30 30 30 30 30 30 30 30 00 45 72 72 6F 72 00 ;  00000000.Error.
```

Last bytes of CODE Section

368 byte cave

Start of DATA Section

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 highjacks 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:

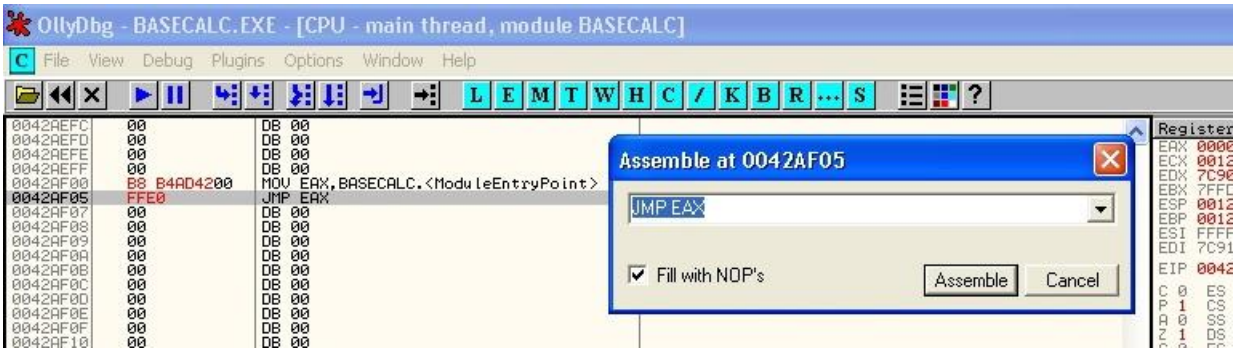
```
MOV  EAX,0042ADB4      ; Load in EAX the Original Entry Point (OEP)
JMP  EAX               ; Jump to OEP
```

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:

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0002a240h: 8B 03 E8 E1 72 FF FF 8B 03 E8 6A 73 FF FF 5B E8 ; ‹.èárÿÿ‹.èjsÿÿ[è
0002a250h: 10 93 FD FF 8B E5 5D C3 FF FF FF FF 0F 00 00 00 ; .“ýÿ‹å]Ãÿÿÿÿ....
0002a260h: 42 61 73 65 20 43 61 6C 63 75 6C 61 74 6F 72 00 ; Base Calculator.
0002a270h: FF FF FF FF 0C 00 00 00 42 41 53 45 43 41 4C 43 ; ÿÿÿÿ....BASECALC
0002a280h: 2E 48 4C 50 00 00 00 00 00 00 00 00 00 00 00 00 ; .HLP............
0002a290h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a2f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0002a300h: B8 B4 AD 42 00 FF E0 00 00 00 00 00 00 00 00 00 ; ¸´-B.ÿà.........
0002a310h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

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.

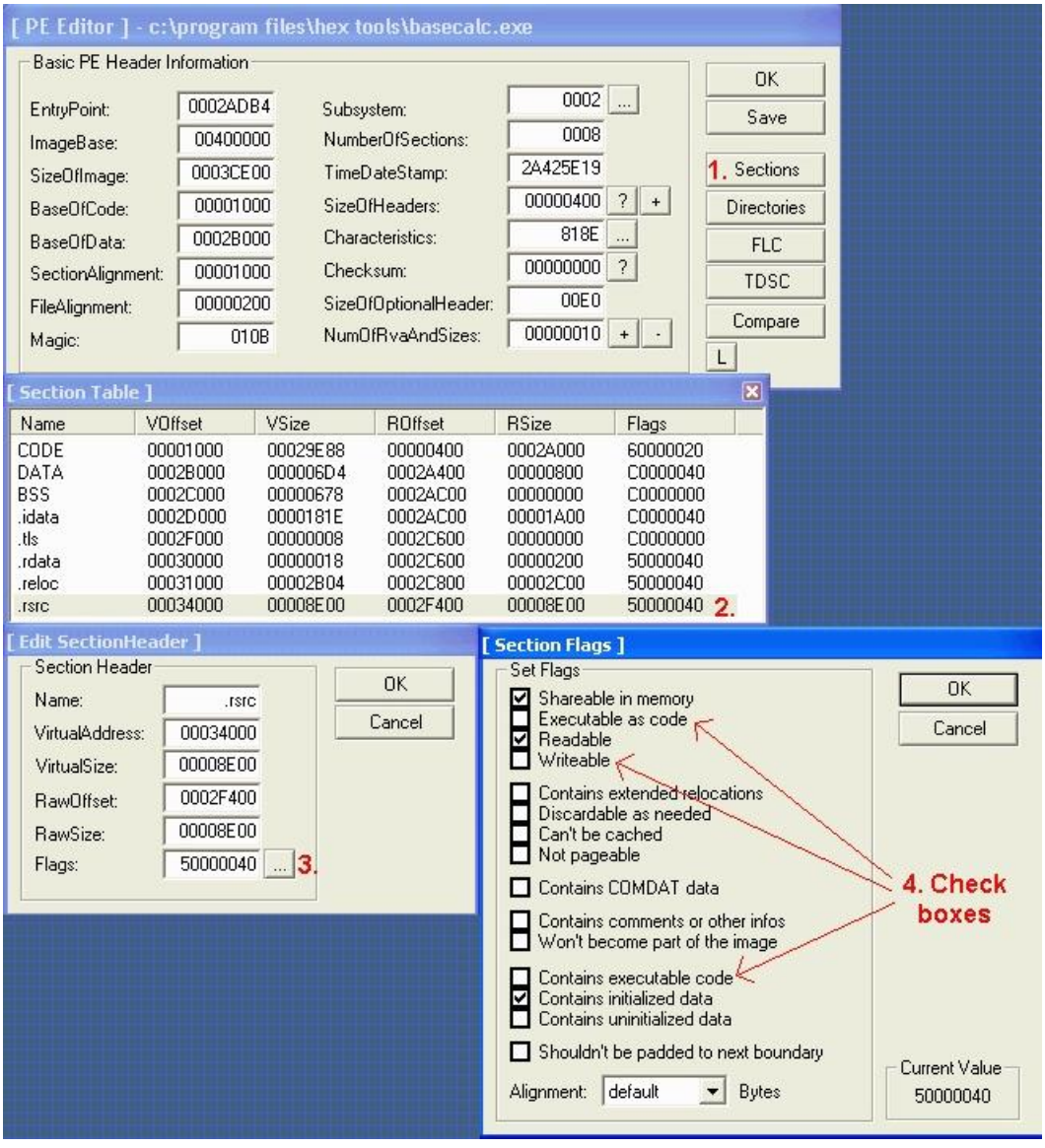
3. References between various sections (such as references to data values from the code section) will all need to be adjusted. This is practically impossible to do without re-compiling and re-linking the original file.

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.

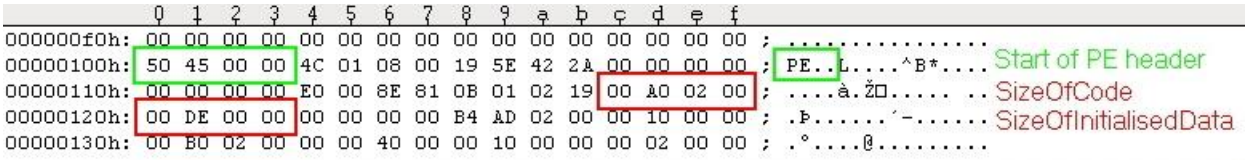
Likewise if the final section is .reloc then the flags will typically be 42000040 for initialised data, discardable and read-only. In order to use this section we must add code, executable and writable and we must subtract discardable to ensure that the loader maps this section into memory. This gives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



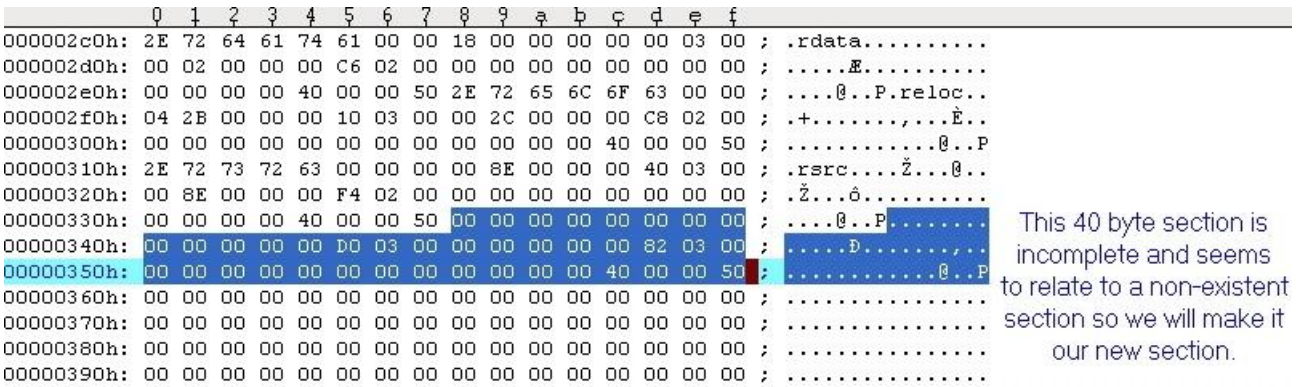
The values are 0002A000 and 0000DE00 respectively. Add 100h on to these to make 0002A100 and 0000DF00. With reverse byte order the values are: 00 A1 02 00 and 00 00 DF 00. Finally copy and paste 100h of 00 bytes (16 rows in the hexeditor) onto the end of the section and save changes. Run the file to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

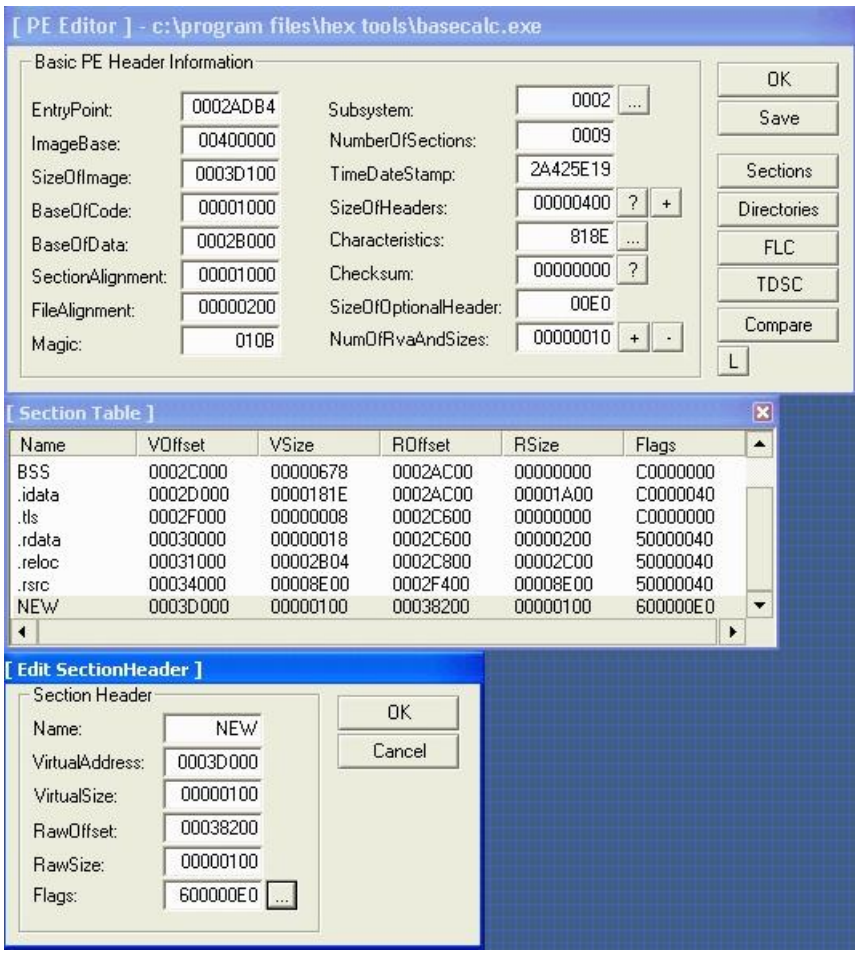
The RVA and raw offset of our new section must be aligned to the above boundaries. The Raw Offset of the section is 00038200h as we said above (which luckily fits with FileAlignment). To get the Virtual Offset of our section we have to calculate this: VirtualAddress of .rsrc + VirtualSize of .rsrc = 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:

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000310h: 2E 72 73 72 63 00 00 00 00 8E 00 00 00 40 03 00 ; .rsrc....Ž...@..
00000320h: 00 8E 00 00 00 F4 02 00 00 00 00 00 00 00 00 00 ; .Ž...ô..........
00000330h: 00 00 00 00 40 00 00 50 4E 45 57 00 00 00 00 00 ; ....@..PNEW.....
00000340h: 00 01 00 00 00 D0 03 00 00 01 00 00 00 82 03 00 ; .....Ð.......‚..
00000350h: 00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 60 ; ............à..`
00000360h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

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



Adding Imports 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 where 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):

VA = RawOffset - RawOffsetOfSection + VirtualOffsetOfSection

= 2C500 - 2AC00 + 2D000 = 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

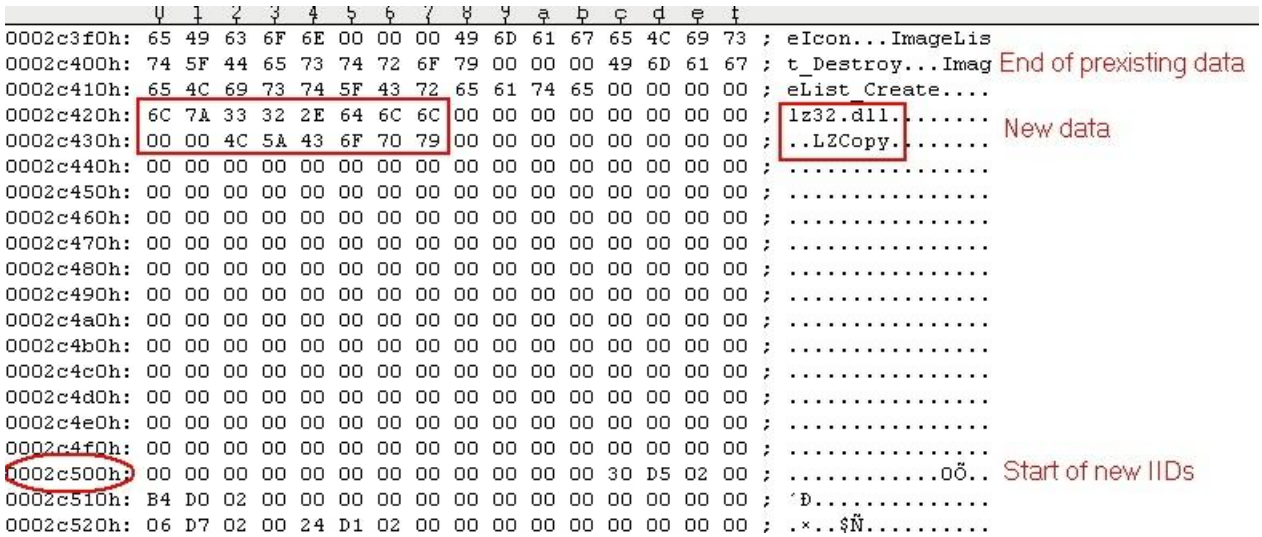
This involves the following steps:

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):

RVA = RawOffset - RawOffsetOfSection + VirtualOffsetOfSection + ImageBase

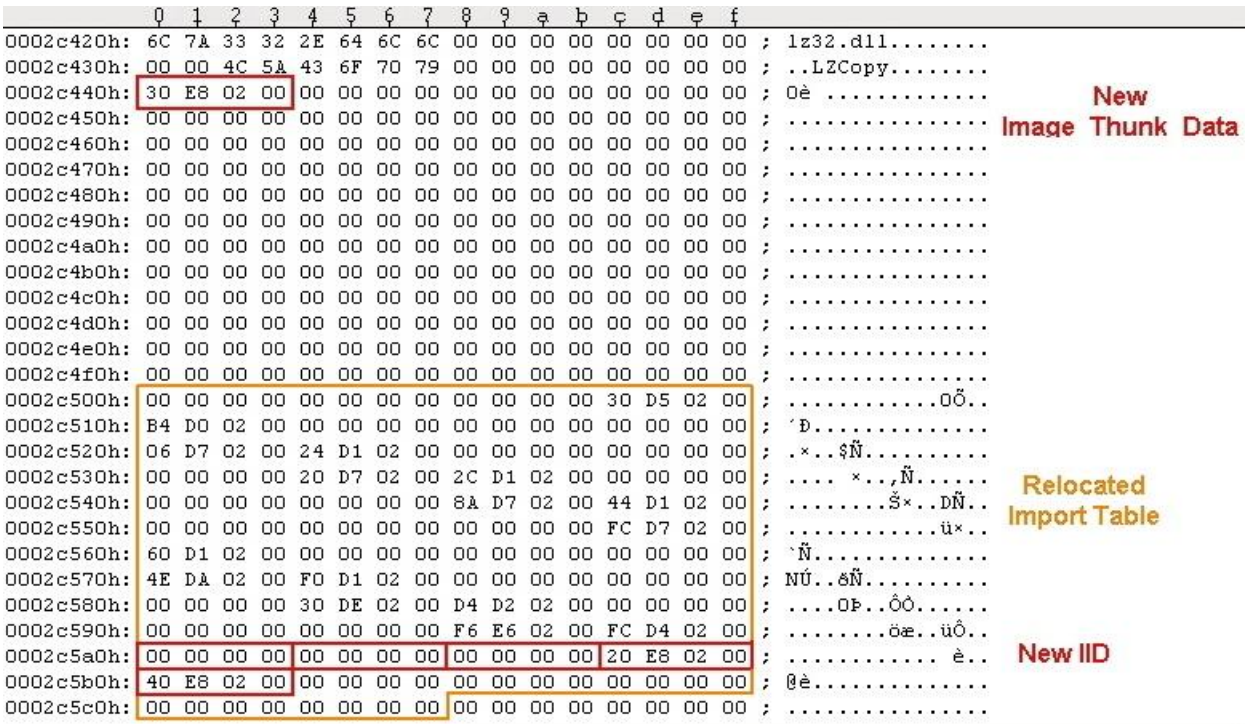
RVA of DLL name = 2C420 - 2AC00 + 2D000 = 2E820h (20 E8 02 00 in reverse)

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

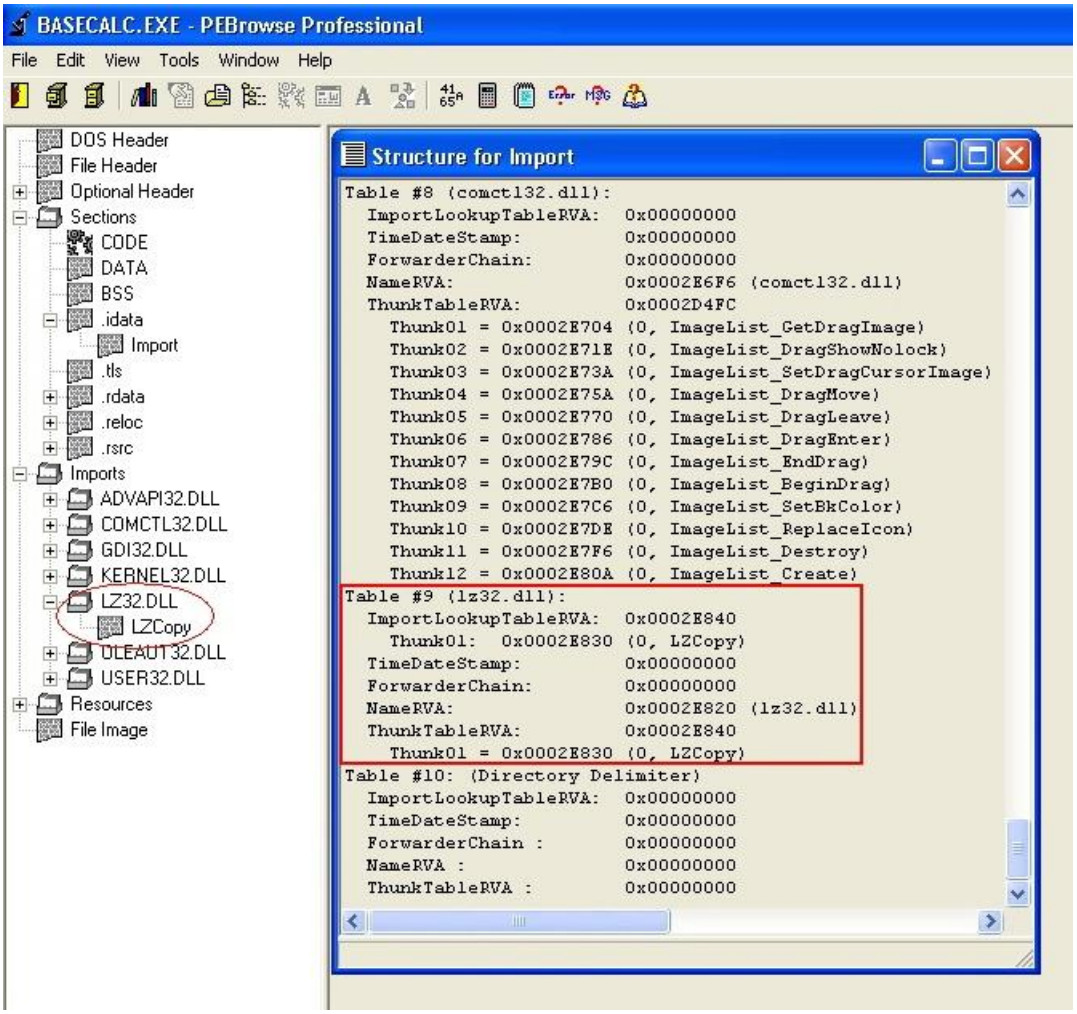
The first one can go into the Name1 field of our new IID but the second must go into an Image_Thunk_Data structure, the RVA of which we can then put into the FirstThunk field (and OriginalFirstThunk) of our new IID. We will put the Image_Thunk_Data structure below the function name string at offset 2C440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

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:

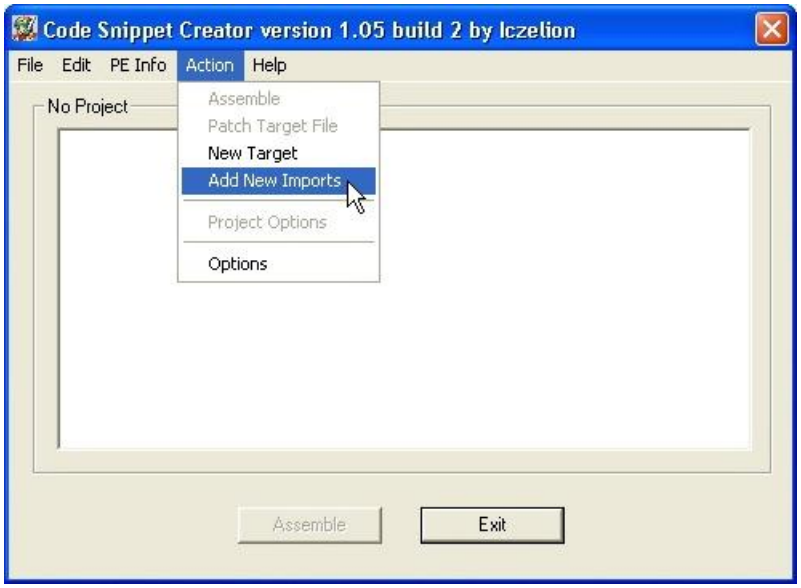
CALL DWORD PTR [XXXXXXXX] where **XXXXXXXX** = RVA of Image_Thunk_Data + ImageBase.

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

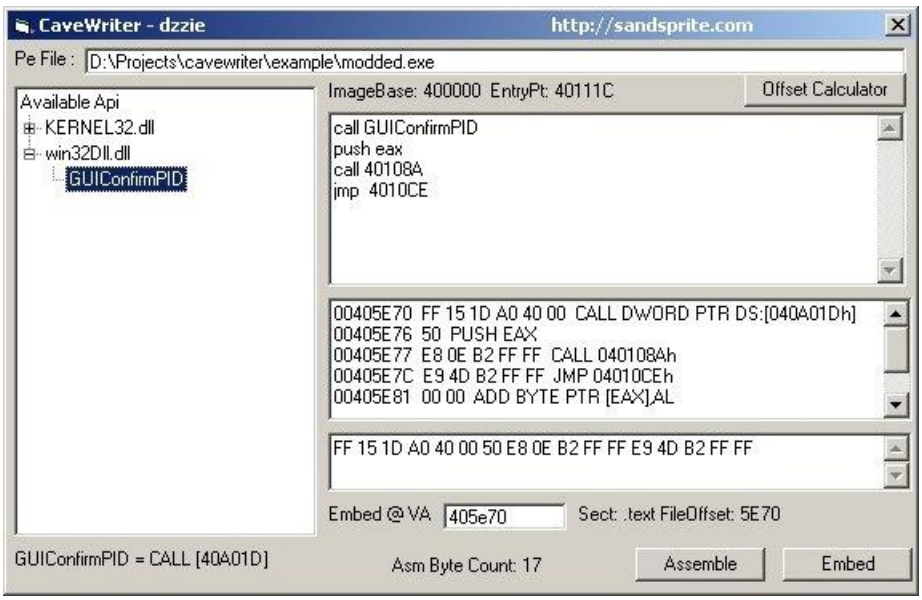
CALL DWORD PTR [0042E840]

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.

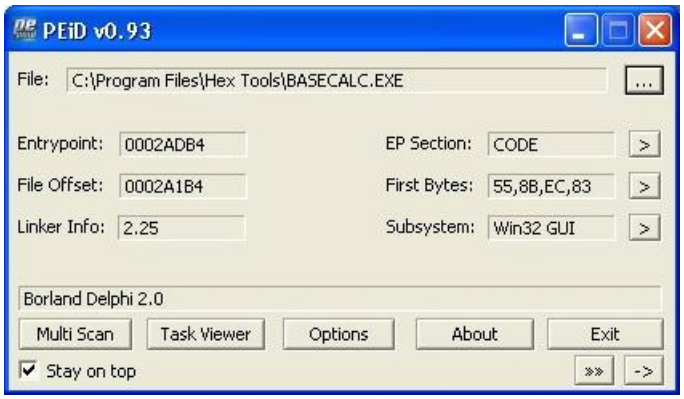


Introduction to Packers

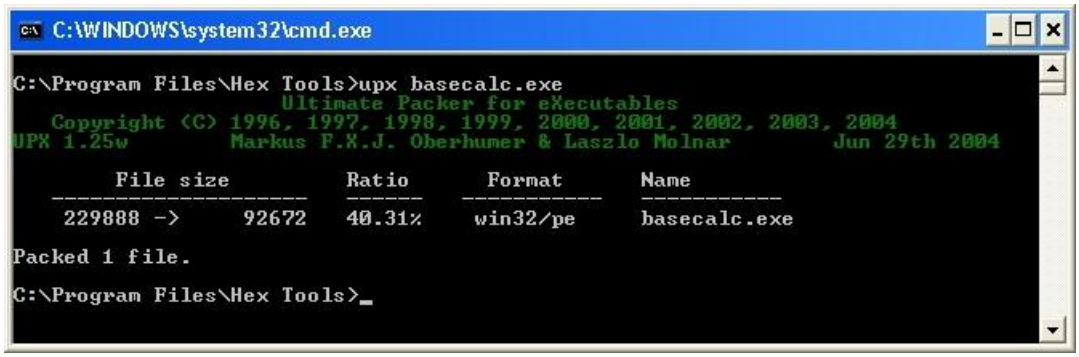
In this section we will examine the effect of a simple packer on our example app and cover 2 main ways of patching a packed executable - either by unpacking first or by inline-patching. We will use UPX1.25 since this is really an executable compressor and doesn't use any advanced protection mechanisms. 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 advertisment of authors of other executable compressors about this topic - just do a websearch on 'unpackers'..."

First we scan our app with PEID:

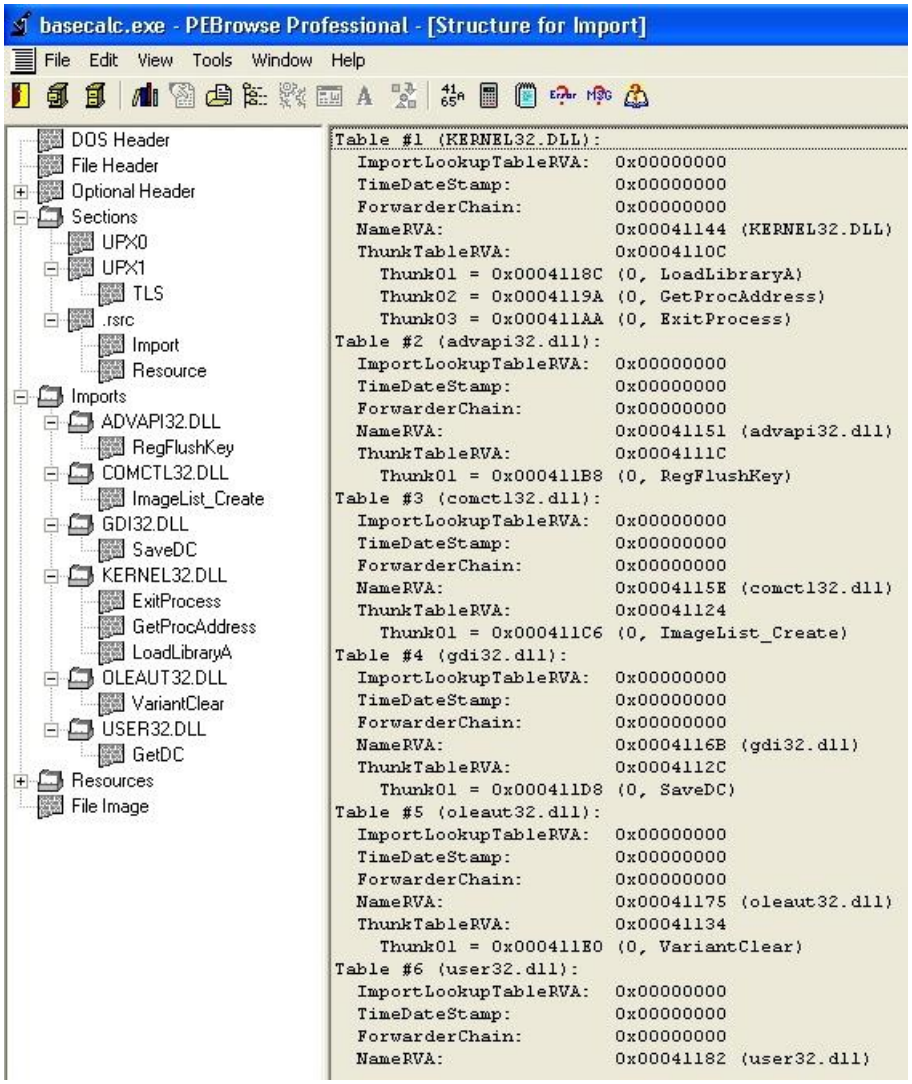


Next we pack our app with upx. This is a commandline utility so we open a DOS box where our app is and type "upx basecalc.exe":



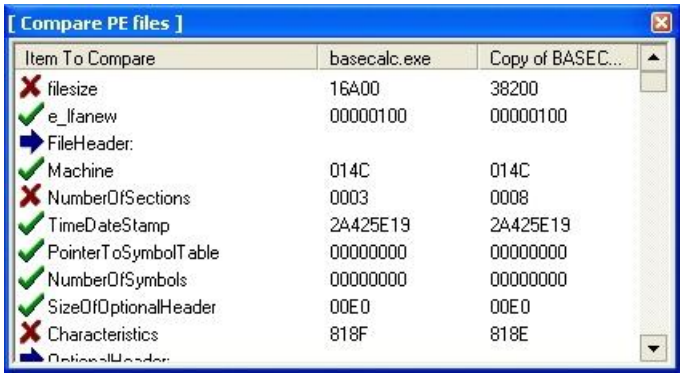
Now we notice file size down from 225Kb to 91 Kb and in PEID we see this:

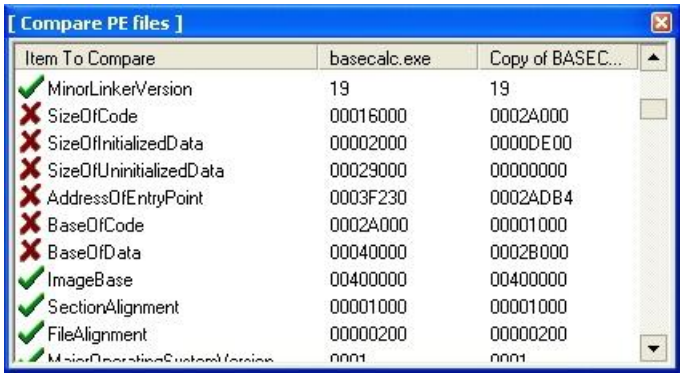
PEBrowse Pro shows that there are now only 3 sections called UPX0, UPX1 and .rsrc The resource section now contains the import directory but for each DLL there are only one or two imported functions - the others have disappeared:

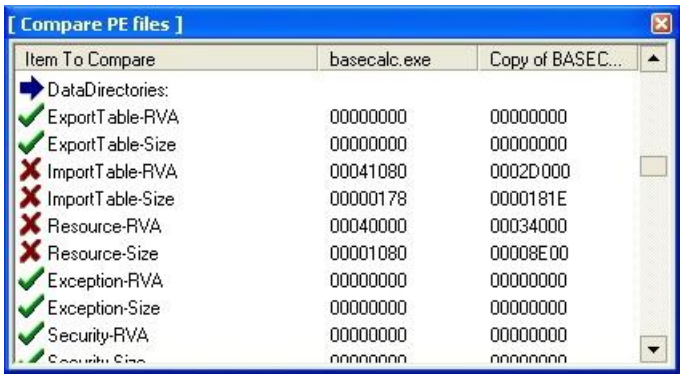


Note the .rsrc section has retained its original name even though the others have changed. Interestingly this dates back to a bug in the LoadTypeLibEx function in oleaut32.dll in Win95 in which the string "rsrc" was used to find and load the resource section. This created an error if the section was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

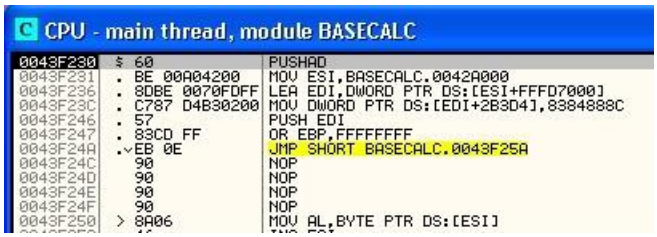
By opening the app in LordPE editor and pressing the compare button we can open an original copy of our app and see the changes made to the headers:







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:



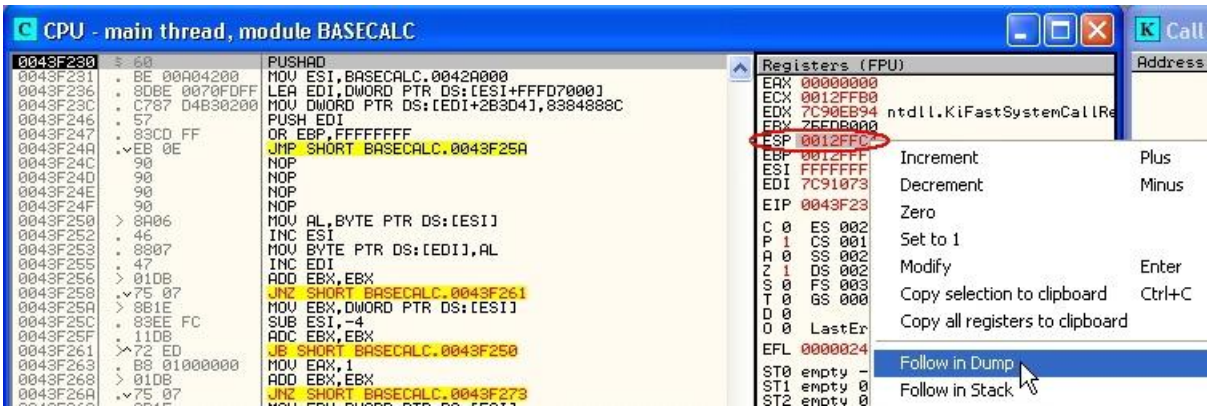
UPX has compressed our app and appended the code with a stub containing the decompression algorithm. The entrypoint of the app has been changed to the start of the stub and after the stub has done its job, execution jumps to the original entrypoint to start our now unpacked program.

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.

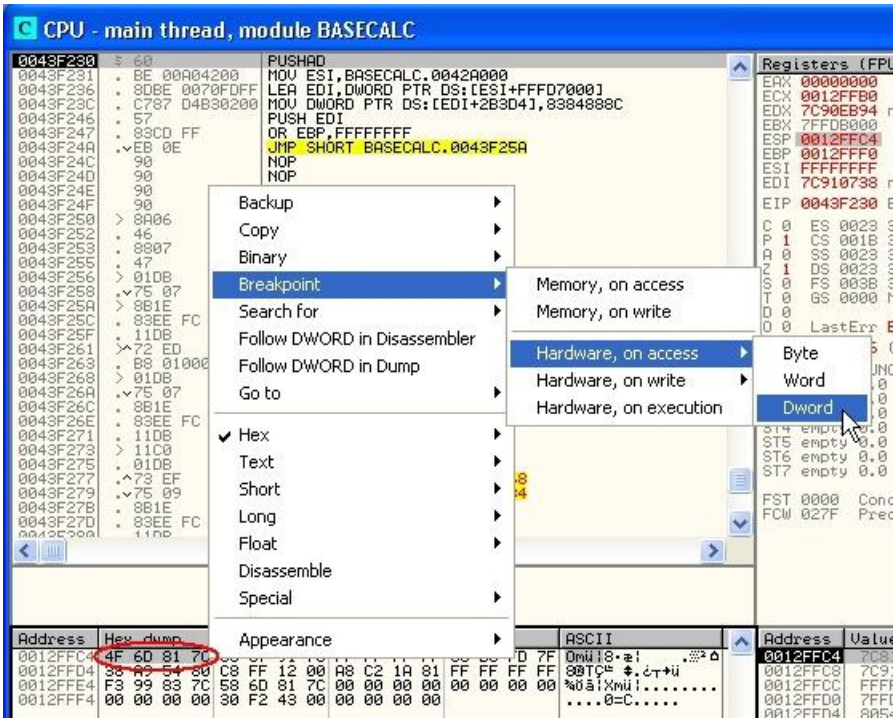
Note at our entrypoint in Olly the first instruction we see is PUSHAD. This stands for PUSH All Double and instructs the CPU to store the contents of all the 32bit (DWORD) registers on the stack, starting with EAX and ending with EDI. Following this the stub does its job and then ends with a POPAD instruction 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 instuction, 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.

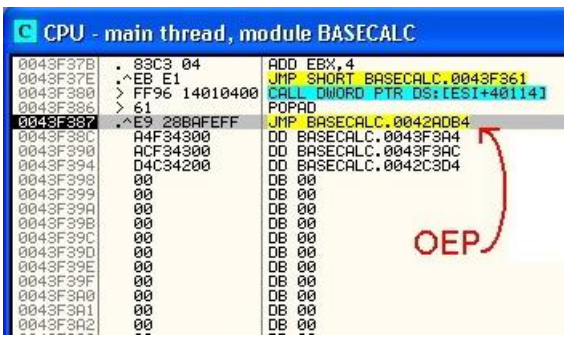
First we must execute the PUSHAD instruction so press F7 to single step. Next we will place our breakpoint. The ESP (Stack Pointer) register always contains the location of the top of the stack so . Rightclick on ESP and select follow in dump - this puts the stack in the hexdump window:



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



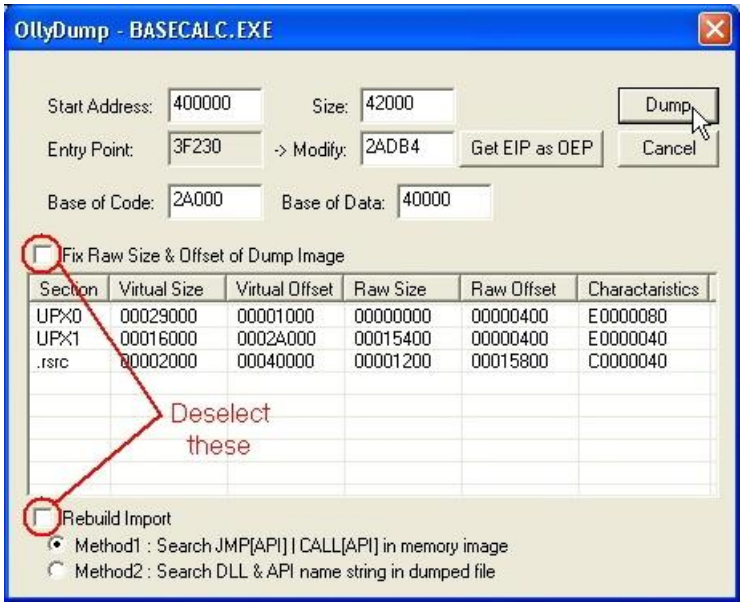
Next run the app by pressing F9 and Olly will break after the PUSHAD directly before the JMP to the OEP. The OEP shown here has the ImageBase 400000h added onto it so to make it into an RVA we subtract it which leaves 0002ADB4h:



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.

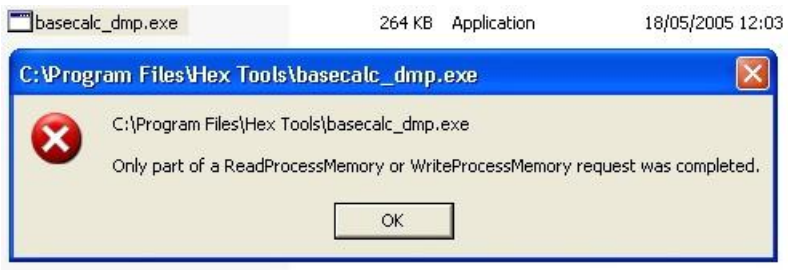
NOTE: other packers which use the same simple PUSHAD/POPAD mechanism may jump to the OEP by using a PUSH instruction to put the value of the OEP onto the top of the stack followed by a RET instruction. The CPU will think it is returning from a function call and conventionally the return address is left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

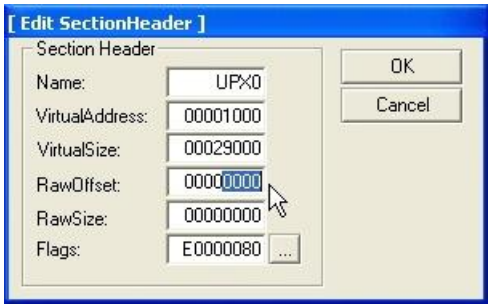


Note that OllyDump has already worked out the base address and size of image (which you could see by looking in the memory map window) and has offered to correct the entrypoint for us (although we could do this manually in the hexeditor). Press the DUMP button and save the file (eg as basecalc_dmp.exe).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



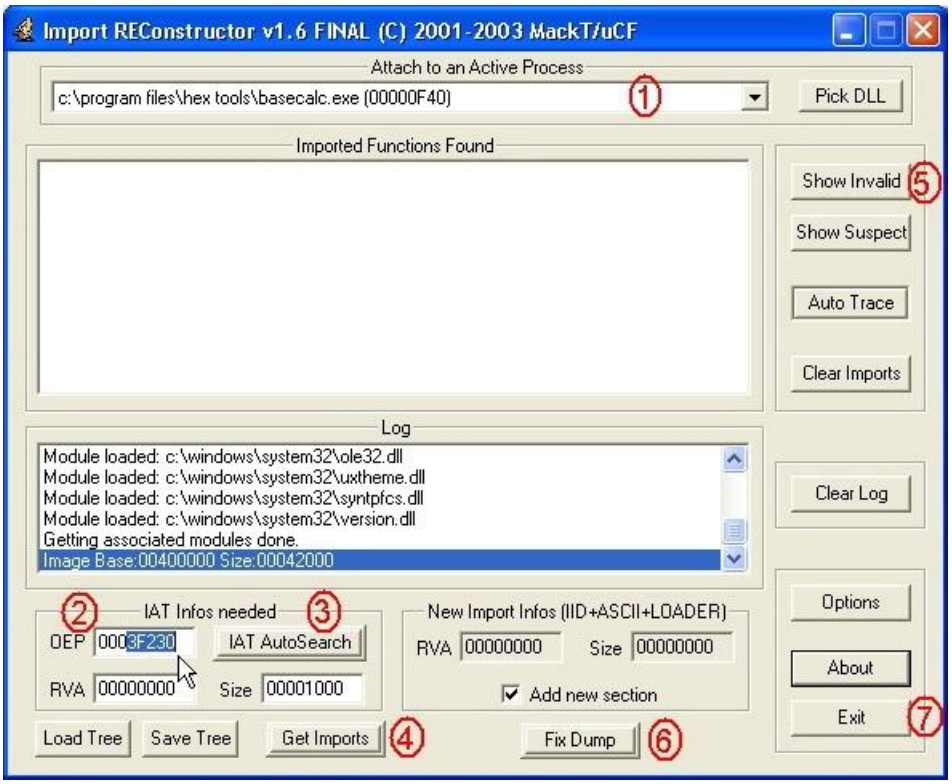
This is because of the alignment issues mentioned earlier - the filesize has also increased as a result. Open the app in LordPE and look at the sections. The raw offset and raw size values are wrong. We will have to make the Raw values equal to the Virtual values for each section for the app to work.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



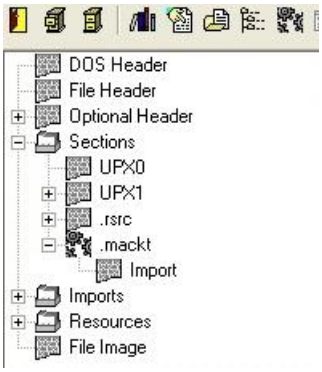
Now make RawOffset equal VirtualAddress and RawSize equal VirtualSize. Repeat for the other sections then click save and exit (this is what the "fix raw size" checkbox in OllyDump does automatically). Now the icon has returned and we get a different error when we try to run it: "The application failed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

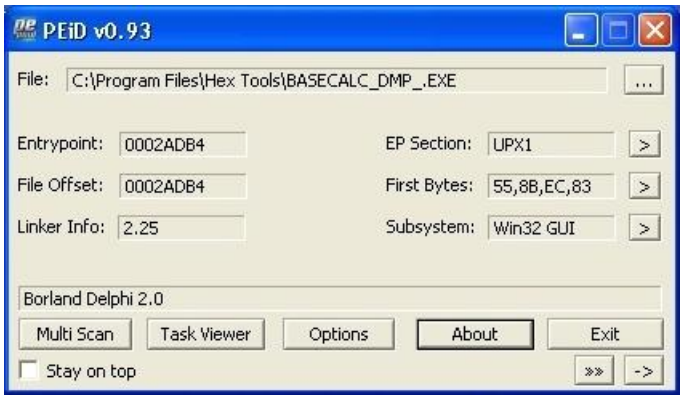


ImpREC will save a fixed copy of our dumped file appended with "_" so run basecalc_dmp_.exe to test it. If we examine this file we will see that size has increased and there is an extra section called "mackt" - this is where ImpREC puts the new import data:



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.

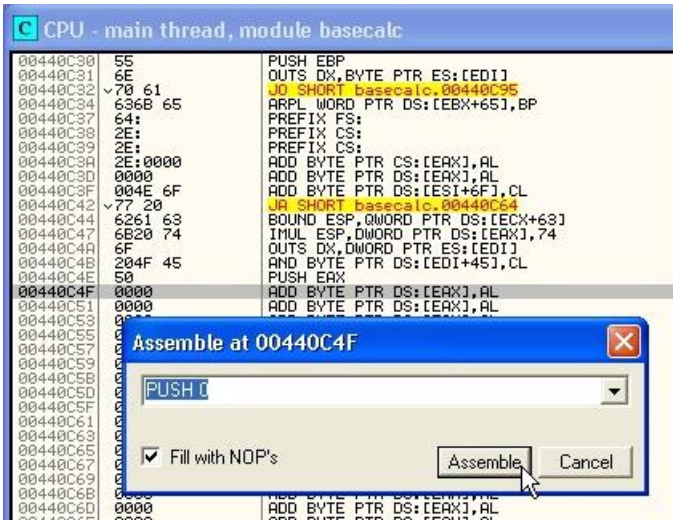
To illustrate this technique we will inject code into the packed executable to pop up a messagebox and let us know when the app is unpacked in memory. Clicking OK will then jump to the OEP and the app will run normally.

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:

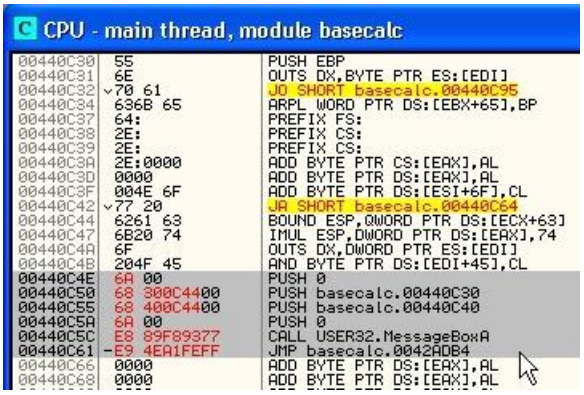
```
00016410h: CC CC CC C8 77 77 77 77 77 77 77 77 77 77 77 77 ; ÌÌÌÈwwwwwwwwwwww
00016420h: 77 77 77 77 00 00 00 00 00 00 00 00 00 00 00 00 ; wwww............
00016430h: 55 6E 70 61 63 6B 65 64 2E 2E 2E 00 00 00 00 00 ; Unpacked........
00016440h: 4E 6F 77 20 62 61 63 6B 20 74 6F 20 4F 45 50 00 ; Now back to OEP.
00016450h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00016460h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00016470h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00016480h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00016490h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000164a0h: 00 00 00 00 D4 53 03 00 28 00 00 00 20 00 00 00 ; ....ÔS..(... ...
000164b0h: 40 00 00 00 01 00 01 00 00 00 00 00 00 00 01 00 ; @...............
000164c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000164d0h: 00 00 00 00 FF FF FF 00 00 00 00 00 7F FF FF FE ; ....ÿÿÿ.....□ÿÿþ
000164e0h: 7F FF FF FF 60 C1 83 06 6E C9 93 76 6A C9 83 06 ; □ÿÿÿ`Á.nÉ"vjÉ.
```

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:

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00016420h: 77 77 77 77 00 00 00 00 00 00 00 00 00 00 00 00 ; wwww............
00016430h: 55 6E 70 61 63 6B 65 64 2E 2E 2E 00 00 00 00 00 ; Unpacked........
00016440h: 4E 6F 77 20 62 61 63 6B 20 74 6F 20 4F 45 50 00 ; Now back to OEP.
00016450h: 00 6A 00 68 30 0C 44 00 68 40 0C 44 00 6A 00 E8 ; .j.h0.D.h@.D.j.è
00016460h: 86 F8 93 77 E9 4B A1 FE FF 00 00 00 00 00 00 00 ; †ø"wéK¡þÿ.......
00016470h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00016480h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00016490h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000164a0h: 00 00 00 00 D4 53 03 00 28 00 00 00 20 00 00 00 ; ....ÔS..(... ...
```

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.

References & Further Reading

The Portable Executable Format -- Micheal J. O'Leary
The Portable Executable File Format from Top to Bottom -- Randy Kath
Peering Inside the PE: A Tour of the Win32 Portable Executable File Format -- Matt Pietrek
An In-Depth Look into the Win32 Portable Executable File Format (2 parts)-- Matt Pietrek
Windows 95 Programming Secrets -- Matt Pietrek
Linkers and Loaders -- John R Levine
Secrets of Reverse Engineering -- Eldad Eilam
PE.TXT -- Bernd Luevelsmeyer
Converting virtual offsets to raw offsets and vice versa – Rheingold
PE Tutorial -- Iczelion
The Portable Executable File Format – KGL
PE Notes, Understanding Imports – yAtEs
Win32 Programmer's Reference
What Goes On Inside Windows 2000: Solving the Mysteries of the Loader -- Russ Osterlund
Tool Interface Standard (TIS) Formats Specification for Windows
Adding Imports by Hand -- Eduardo Labir (Havok), CBJ
Enhancing functionality of programs by adding extra code -- c0v3rt+
Working Manually with Import Tables -- Ricardo Narvaja
All tutorials concerning manual unpacking (especially those from ARTeam, with special reference to the Beginner Olly series by Shub and Gabri3l.)

Conclusion

I hope this tutorial has helped to make clear some of the complexities of PE format, particularly those relevant to RCE. There are areas which I have skirted over briefly and others which I have omitted altogether to save time and space. For those of you who may not have read all of the above referenced texts or who need further information on specifics, I will summarise the most important.

First on the must-read list are the 2 parts of "An In-Depth Look..." by Pietrek. These superseded his earlier article "Peering Inside the PE" which is dated and contains inaccuracies. The articles by O'Leary and Kath do not add significantly to Pietrek's. PE.TXT by Luevelsmeyer is long and detailed and best used as a reference, and there are certain details which even he admits unknown. There are sections in several books concerning PE format eg. Hacker Disassembling Uncovered, Hackproof your software, but these are brief and less detailed than Pietrek's articles. Secrets of Reverse Engineering, however is different and gives a most excellent overview of the more important concepts of PE format along with reversing in Windows in general.

Second on the must-read list are the excellent set of tutorials by Iczelion which are detailed and also oriented around writing routines in ASM to parse and manipulate various parts of the PE file including the imports.

Third on the list are the Win32 Programmer's Reference (a must for RCE) and WINNT.h or windows.inc files which contain all of the definitions of the structures outlined in this tutorial.

The next most valuable are the articles published in The CodeBreaker's Journal by Eduardo Labir (aka Havok). Apart from that listed above, there are titles concerning Asprotect which are very detailed and should be considered essential reading.

There is also a small collection of articles by Rheingold several of which concentrate on working with the PE format. Issues 1-4 survive on the internet and can be found here:

http://www.programming journal.com/issue4/

Note there is no root or index page, nor are they indexed by Google.

This knowledge will provide a solid background upon which to understand the mechanisms by which executable packers work and subsequently how to defeat them by manual unpacking, inline patching, writing loaders, etc. There are many good tutorials around concerning these things, especially the ones I have eluded to above from ARTeam and Ricardo Narvaja. Read everything you can find, take the time to understand it and finally "work well" as +ORC always said.

Thanks & Greetz

Thanks to the whole ARTeam:
[Nilrem] [JDog45] [Shub - Nigurrath] [MaDMAn_H3rCuL3s] [Ferrari] [Kruger] [Teerayoot]
[R@dier] [ThunderPwr] [Eggi] [EJ12N] [Stickman 373] [Bone Enterprise] [KaGra] [Gabri3l]

In particular thanks to Shub for suggestions and proof-reading.

Thanks also to the authors of the above referenced works and tools used.

If you have any comments, suggestions or corrections email me: goppit@hotmail.com

Appendix 1 - Complete PE Offset Reference

While there is a lot of data and various parts of the structure are at varying positions there are still a lot of useful fixed and relative offsets that will help when disassembling/examining PE files. Resource information and the such like are omitted - there are good tools available to manipulate these e.g. ResHacker.

The DOS Header

OFFSET	SIZE	NAME	EXPLANATION
00	WORD	e_magic	Magic DOS signature MZ (4Dh 5Ah)
02	WORD	e_cblp	Bytes on last page of file
04	WORD	e_cp	Pages in file
06	WORD	e_crlc	Relocations
08	WORD	e_cparhdr	Size of header in paragraphs
0A	WORD	e_minalloc	Minimum extra paragraphs needed
0C	WORD	e_maxalloc	Maximum extra paragraphs needed
0E	WORD	e_ss	Initial (relative) SS value
10	WORD	e_sp	Initial SP value
12	WORD	e_csum	Checksum
14	WORD	e_ip	Initial IP value
16	WORD	e_cs	Initial (relative) CS value
18	WORD	e_lfarlc	File address of relocation table
1A	WORD	e_ovno	Overlay number
1C	WORD	e_res[4]	Reserved words
24	WORD	e_oemid	OEM identifier (for e_oeminfo)
26	WORD	e_oeminfo	OEM information; e_oemid specific
28	WORD	e_res2[10]	Reserved words
3C	DWORD	e_lfanew	Offset to start of PE header

The PE Header

Offsets shown are from the beginning of this section.

00	DWORD	Signature	PE Signature PE.. (50h 45h 00h 00h)
04	WORD	Machine	014Ch = Intel 386, 014Dh = Intel 486, 014Eh = Intel 586, 0200h = Intel 64-bit, 0162h = MIPS
06	WORD	NumberOfSections	Number Of Sections
08	DWORD	TimeDateStamp	Date & time image was created by the linker
0C	DWORD	PointerToSymbolTable	Zero or offset of COFF symbol table in older files
10	DWORD	NumberOfSymbols	Number of symbols in COFF symbol table
14	WORD	SizeOfOptionalHeader	Size of optional header in bytes (224 in 32bit exe)
16	WORD	Characteristics	see below
18	******	**START OF OPTIONAL HEADER**	**********

18	WORD	Magic	010Bh=32-bit executable image 020Bh=64-bit executable image 0107h=ROM image
1A	BYTE	MajorLinkerVersion	Major version number of the linker
1B	BYTE	MinorLinkerVersion	Minor version number of the linker
1C	DWORD	SizeOfCode	size of code section or sum if multiple code sections
20	DWORD	SizeOfInitializedData	as above
24	DWORD	SizeOfUninitializedData	as above
28	DWORD	AddressOfEntryPoint	Start of code execution, optional for DLLs, zero when none present
2C	DWORD	BaseOfCode	RVA of first byte of code when loaded into RAM
30	DWORD	BaseOfData	RVA of first byte of data when loaded into RAM
34	DWORD	ImageBase	Preferred load address
38	DWORD	SectionAlignment	Alignment of sections when loaded in RAM
3C	DWORD	FileAlignment	Alignment of sections in file on disk
40	WORD	MajorOperatingSystemVersion	Major version no. of required operating system
42	WORD	MinorOperatingSystemVersion	Minor version no. of required operating system
44	WORD	MajorImageVersion	Major version number of the image
46	WORD	MinorImageVersion	Minor version number of the image
48	WORD	MajorSubsystemVersion	Major version number of the subsystem
4A	WORD	MinorSubsystemVersion	Minor version number of the subsystem
4C	DWORD	Reserved1	
50	DWORD	SizeOfImage	Amount of memory allocated by loader for image. Must be a multiple of SectionAlignment
54	DWORD	SizeOfHeaders	Offset of first section, multiple of FileAlignment
58	DWORD	CheckSum	Image checksum (only required for kernel-mode drivers and some system DLLs).
5C	WORD	Subsystem	0002h=Windows GUI, 0003h=console
5E	WORD	DllCharacteristics	0001h=per-process library initialization 0002h=per-process library termination 0003h=per-thread library initialization 0004h=per-thread library termination
60	DWORD	SizeOfStackReserve	Number of bytes reserved for the stack
64	DWORD	SizeOfStackCommit	Number of bytes actually used for the stack
68	DWORD	SizeOfHeapReserve	Number of bytes to reserve for the local heap
6C	DWORD	SizeOfHeapCommit	Number of bytes actually used for local heap
70	DWORD	LoaderFlags	This member is obsolete.
74	DWORD	NumberOfRvaAndSizes	Number of directory entries.

78	******	**START OF DATA DIRECTORY**	**********
78	DWORD	IMAGE_DATA_DIRECTORY0	RVA of Export Directory
7C	DWORD		size of Export Directory
80	DWORD	IMAGE_DATA_DIRECTORY1	RVA of Import Directory (array of IIDs)
84	DWORD		size of Import Directory (array of IIDs)
88	DWORD	IMAGE_DATA_DIRECTORY2	RVA of Resource Directory
8C	DWORD		size of Resource Directory
90	DWORD	IMAGE_DATA_DIRECTORY3	RVA of Exception Directory
94	DWORD		size of Exception Directory
98	DWORD	IMAGE_DATA_DIRECTORY4	Raw Offset of Security Directory
9C	DWORD		size of Security Directory
A0	DWORD	IMAGE_DATA_DIRECTORY5	RVA of Base Relocation Directory
A4	DWORD		size of Base Relocation Directory
A8	DWORD	IMAGE_DATA_DIRECTORY6	RVA of Debug Directory
AC	DWORD		size of Debug Directory
B0	DWORD	IMAGE_DATA_DIRECTORY7	RVA of Copyright Note
B4	DWORD		size of Copyright Note
B8	DWORD	IMAGE_DATA_DIRECTORY8	RVA to be used as Global Pointer (IA-64 only)
BC	DWORD		Not used
C0	DWORD	IMAGE_DATA_DIRECTORY9	RVA of Thread Local Storage Directory
C4	DWORD		size of Thread Local Storage Directory
C8	DWORD	IMAGE_DATA_DIRECTORY10	RVA of Load Configuration Directory
CC	DWORD		size of Load Configuration Directory
D0	DWORD	IMAGE_DATA_DIRECTORY11	RVA of Bound Import Directory
D4	DWORD		size of Bound Import Directory
D8	DWORD	IMAGE_DATA_DIRECTORY12	RVA of first Import Address Table
DC	DWORD		total size of all Import Address Tables
E0	DWORD	IMAGE_DATA_DIRECTORY13	RVA of Delay Import Directory
E4	DWORD		size of Delay Import Directory
E8	DWORD	IMAGE_DATA_DIRECTORY14	RVA of COM Header (top level info & metadata...
EC	DWORD		size of COM Header ... (in .NET executables)
F0	DWORD	ZERO (Reserved)	Reserved
F4	DWORD	ZERO (Reserved)	Reserved
F8	******	**START OF SECTION TABLE**	*******Offsets shown from here********
00	8 Bytes	Name1	Name of first section header
08	DWORD	misc (VirtualSize)	Actual size of data in section
0C	DWORD	virtual address	RVA where section begins in memory
10	DWORD	SizeOfRawData	Size of data on disk (multiple of FileAlignment)
14	DWORD	pointerToRawData	Raw offset of section on disk

18	DWORD	pointerToRelocations	Start of relocation entries for section, zero if none
1C	DWORD	PointerToLinenumbers	Start of line-no. entries for section, zero if none
20	WORD	NumberOfRelocations	This value is zero for executable images.
22	WORD	NumberOfLineNumbers	Number of line-number entries for section.
24	DWORD	Characteristics	see end of page below
00	8 Bytes	Name1	Name of second section header
	******	**Repeats for rest of sections**	**********

The Export Table

Offsets shown from beginning of table (given at offset 78 from start of PE header). The following 40 Bytes repeat for each export library (DLL whose functions are imported by the executable) and ends with one full of zeroes.

OFFSET	SIZE	NAME	EXPLANATION
00	DWORD	Characteristics	Set to zero (currently none defined)
04	DWORD	TimeDateStamp	often set to zero
08	WORD	MajorVersion	user-defined version number, otherwise zero
0A	WORD	MinorVersion	as above
0C	DWORD	Name	RVA of DLL name in null-terminated ASCII
10	DWORD	Base	First valid exported ordinal, normally=1
14	DWORD	NumberOfFunctions	Number of entries in EAT
18	DWORD	NumberOfNames	Number of entries in ENT
1C	DWORD	AddressOfFunctions	RVA of EAT (export address table)
20	DWORD	AddressOfNames	RVA of ENT (export name table)
24	DWORD	AddressOfNameOrdinals	RVA of EOT (export ordinal table)

The Import Table

Offsets shown from beginning of table (given at offset 80 from start of PE header). The following 5 DWORDS repeat for each import library (DLL whose functions are imported by the executable) and ends with one full of zeroes.

OFFSET	SIZE	NAME	EXPLANATION
00	DWORD	OriginalFirstThunk	RVA to Image_Thunk_Data
04	DWORD	TimeDateStamp	zero unless bound against imported DLL
08	DWORD	ForwarderChain	pointer to 1st redirected function (or 0)
0C	DWORD	Name1	RVA to name in null-terminated ASCII
10	DWORD	FirstThunk	RVA to Image_Thunk_Data

Image Characteristics Flags

FLAG	EXPLANATION
0001	Relocation info stripped from file
0002	File is executable (no unresolved external references)
0004	Line numbers stripped from file
0008	Local symbols stripped from file
0010	Lets OS aggressively trim working set
0020	App can handle >2Gb addresses
0080	Low bytes of machine word are reversed
0100	requires 32-bit WORD machine
0200	Debugging info stripped from file into .DBG file
0400	If image is on removable media, copy and run from swap file
0800	If image is on a network, copy and run from swap file
1000	System file
2000	File is a DLL
4000	File should only be run on a single-processor machine
8000	High bytes of machine word are reversed

Section Characteristics Flags

FLAG	EXPLANATION
00000008	Section should not be padded to next boundary
00000020	Section contains code
00000040	Section contains initialised data (which will become initialised with real values before the file is launched)
00000080	Section contains uninitialised data (which will be initialised as 00 byte values before launch)
00000200	Section contains comments for the linker
00000800	Section contents will not become part of image
00001000	Section contents comdat (Common Block Data)
00008000	Section contents cannot be accessed relative to GP
00100000 to 00800000	Boundary alignment settings
01000000	Section contains extended relocations
02000000	Section can be discarded (e.g. .reloc)
04000000	Section is not cacheable
08000000	Section is pageable
10000000	Section is shareable
20000000	Section is executable

40000000	Section is readable
80000000	Section is writable

Appendix 2 - Relative Virtual Addressing Explained

In an executable file or DLL, an **RVA** is always the address of an item once loaded into memory, with the base address (**ImageBase**) of the image file subtracted from it: **RVA = VA - ImageBase**

hence also: **VA = RVA + ImageBase**

It's exactly the same thing as file offset but it's relative to a point in virtual address space, not the beginning of the PE file. E.g. if a PE file loads at 400000h in the virtual address (VA) space and the program starts execution at the virtual address 401000h, we can say that the program starts execution at RVA 1000h. An RVA is relative to the starting VA of the module. The RVA of an item will almost always differ from its position within the file on disk - the offset. This is a pitfall for newcomers to PE programming. **Most of the addresses in the PE file are RVAs and are meaningful only when the PE file is loaded into memory by the PE loader**.

The term "Virtual Address" is used because Windows creates a distinct virtual address space for each process, independent of physical memory. For almost all purposes, a virtual address should be considered just an address. As above, a virtual address is not as predictable as an RVA, because the loader might not load the image at its preferred base address.

Why does the PE file format use RVA? It's to help reduce the load of the loader. Since a module can be relocated anywhere in the virtual address space, it would be hell for the loader to fix every hardcoded address in the module. In contrast, if all relocatable items in the file use RVA, there is no need for the loader to fix anything: it simply relocates the whole module to a new starting VA.

Converting virtual offsets to raw offsets and vice versa (from Rheingold)

Converting raw offsets (the one in a file you see in a HexEditor) to virtual offsets (the one you see in a debugger) is very useful if you work with the PE header. For this you need to know some values from the PE header. You need to know the **ImageBase,** the name of the section in which your offset lies, . Below you see an example of a PE header from the beginning of the file (where it is actually a MZ header until offset 80h) until the section definitions end (offset 23Fh). The example is taken from my notepad.exe.

```
00000000 4D5A 9000 0300 0000 0400 0000 FFFF 0000 MZ..............
00000010 B800 0000 0000 0000 4000 0000 0000 0000 ........@.......
00000020 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000030 0000 0000 0000 0000 0000 0000 8000 0000 ................
00000040 0E1F BA0E 00B4 09CD 21B8 014C CD21 5468 ........!..L.!Th
00000050 6973 2070 726F 6772 616D 2063 616E 6E6F is program canno
00000060 7420 6265 2072 756E 2069 6E20 444F 5320 t be run in DOS
00000070 6D6F 6465 2E0D 0D0A 2400 0000 0000 0000 mode....$.......
00000080 5045 0000 4C01 0500 6591 4635 0000 0000 PE..L...e.F5....
00000090 0000 0000 E000 0E01 0B01 030A 0040 0000 .............@..
000000A0 0072 0000 0000 0000 CC10 0000 0010 0000 .r..............
000000B0 0050 0000 0000 4000 0010 0000 0010 0000 .P....@.........
000000C0 0400 0000 0000 0000 0400 0000 0000 0000 ................
000000D0 00E0 0000 0004 0000 D509 0100 0200 0000 ................
000000E0 0000 1000 0010 0000 0000 1000 0010 0000 ................
000000F0 0000 0000 1000 0000 0000 0000 0000 0000 ................
00000100 0060 0000 8C00 0000 0070 0000 E453 0000 .`.......p...S..
00000110 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000120 00D0 0000 3C09 0000 0000 0000 0000 0000 ....<...........
00000130 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000140 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000150 0000 0000 0000 0000 E062 0000 4002 0000 .........b..@...
00000160 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000170 0000 0000 0000 0000 2E74 6578 7400 0000 .........text...
00000180 9C3E 0000 0010 0000 0040 0000 0010 0000 .>.......@......
00000190 0000 0000 0000 0000 0000 0000 2000 0060 ............ ..`
000001A0 2E64 6174 6100 0000 4C08 0000 0050 0000 .data...L....P..
000001B0 0010 0000 0050 0000 0000 0000 0000 0000 .....P..........
000001C0 0000 0000 4000 00C0 2E69 6461 7461 0000 ....@....idata..
000001D0 E80D 0000 0060 0000 0010 0000 0060 0000 .....`.......`..
000001E0 0000 0000 0000 0000 0000 0000 4000 0040 ............@..@
000001F0 2E72 7372 6300 0000 0060 0000 0070 0000 .rsrc....`...p..
00000200 0060 0000 0070 0000 0000 0000 0000 0000 .`...p..........
00000210 0000 0000 4000 0040 2E72 656C 6F63 0000 ....@..@.reloc..
00000220 9C0A 0000 00D0 0000 0010 0000 00D0 0000 ................
00000230 0000 0000 0000 0000 0000 0000 4000 0042 ............@..B
```

Example 1 - Converting raw offset 7800h to a virtual offset:

The **ImageBase** (DWORD value 34h bytes after the PE header begins, in our case B4h) is 40000h. The Section Table starts F8h bytes after the PE header starts, in our case 178h. It is this part:

```
00000170                          2E74 6578 7400 0000 .........text...
00000180 9C3E 0000 0010 0000 0040 0000 0010 0000 .>.......@......
00000190 0000 0000 0000 0000 0000 0000 2000 0060 ............ ..`
000001A0 2E64 6174 6100 0000 4C08 0000 0050 0000 .data...L....P..
000001B0 0010 0000 0050 0000 0000 0000 0000 0000 .....P..........
000001C0 0000 0000 4000 00C0 2E69 6461 7461 0000 ....@....idata..
000001D0 E80D 0000 0060 0000 0010 0000 0060 0000 .....`.......`..
000001E0 0000 0000 0000 0000 0000 0000 4000 0040 ............@..@
000001F0 2E72 7372 6300 0000 0060 0000 0070 0000 .rsrc....`...p..
00000200 0060 0000 0070 0000 0000 0000 0000 0000 .`...p..........
00000210 0000 0000 4000 0040 2E72 656C 6F63 0000 ....@..@.reloc..
00000220 9C0A 0000 00D0 0000 0010 0000 00D0 0000 ................
00000230 0000 0000 0000 0000 0000 0000 4000 0042 ............@..B
```

The colored values tell us the following values:

Name of the Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset
.text	3E9C	1000	4000	1000
.data	84C	5000	1000	5000
.idata	DE8	6000	1000	6000
.rsrc	6000	7000	6000	7000
.reloc	A9C	D000	1000	D000

The Virtual Size and orange coloured values in the hexeditor output above are not of interest for the conversion but have other functions (see Section Table page).

We want to convert raw offset 7800h. It seems obvious that this offset lies in the .rsrc section because it starts at 7000h (Raw Offset) and is 6000h bytes long (Raw Size). Offset 7800h is located 800h bytes after the section starts in the file. Since the sections are copied to the memory just like they are in the file, this address will be found 800h bytes after the section starts in memory (7000h; Virtual Offset). The offset we search is at 7800h. This is absolutely **not** common that the raw offset equals the virtual offset (without **ImageBase**). In this case it is only because the sections start at the same offset in memory and in the file.

The general formula is:

RVA = RawOffset_YouHave - RawOffsetOfSection + VirtualOffsetOfSection + ImageBase

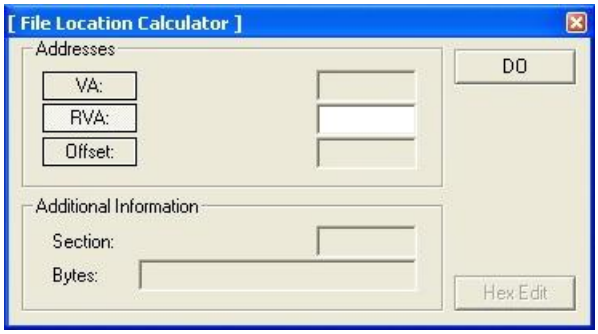
(ImageBase = DWORD value 34h bytes after the PE header begins)

The conversion from a virtual offset to a raw offset just goes the other way round. The general formula is:

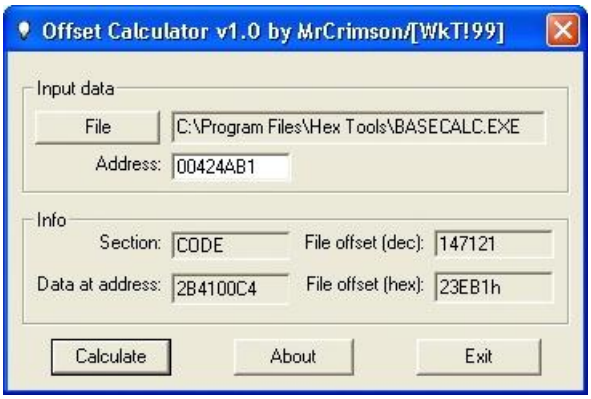
Raw Offset = RVA_YouHave - ImageBase - VirtualOffsetOfSection + RawOffsetOfSection

For 40A000 that is: 40A000-400000-7000+7000 = A000

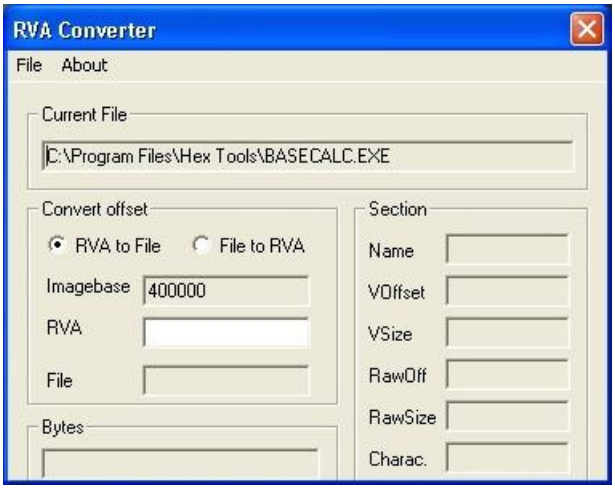
There are also automated tools to perform the above conversions. Pressing the "FLC" button on the PE Editor of LordPE will allow you to convert an RVA to an offset:



Offset Calculator also only allows conversion one-way from RVA to Raw Offset:



RVA Calculator allows conversion **both** ways:


RVA Converter
File
About
Current File
C:\Program Files\Hex Tools\BASECALC.EXE
Convert offset
RVA to File
File to RVA
Imagebase
400000
RVA
File
Bytes
Section
Name
VOffset
VSize
RawOff
RawSize
Charac.