

...:[ARTeam Tutorial]:...

PORTABLE EXECUTABLE FILE FORMAT

Category :	Relates to cracking, unpacking, reverse engineering
Level :	Intermediate
Test OS :	XP Pro SP2
Author :	Goppit
Translated by :	kienmanowar (REA-cRaCkErTeAm)

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
ResHacker	http://delphi.icm.edu.pl/ftp/tools/ResHack.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

Lời mở đầu :

Bài viết này nhằm mục đích để đổi chiêu thông tin từ nhiều nguồn khác nhau và trình bày nó theo một phương pháp mà những người mới bắt đầu có thể tiếp cận dễ dàng nhất. Mặc dù bài viết được trình bày một cách tóm tắt trong nhiều phần, tuy nhiên nó được định hướng theo mục đích **reverse code engineering** cho nên các thông tin không cần thiết sẽ được bỏ qua. Bạn sẽ nhận thấy rằng trong bài viết này tôi đã vay mượn rất nhiều từ các bài viết khác nhau đã được công bố, phổ biến và tất cả các tác giả của những bài viết đó đã được tôi nhắc đến với lòng cảm ơn sâu sắc trong phần tài liệu tham khảo ở phía cuối của bài viết này.

PE là định dạng file riêng của Win32. Tất cả các file có thể thực thi được trên Win32 (ngoại trừ các tập tin VxDs và các file DLLs 16 bit) đều sử dụng định dạng PE. Các file DLLs 32 bit, các file COMs, các điều khiển OCX, các chương trình ứng dụng nhỏ trong Control Panel (.CPL files) và các ứng dụng .NET tất cả đều là định dạng PE. Thậm chí các chương trình điều khiển ở Kernel mode của các hệ điều hành NT cũng sử dụng định dạng PE.

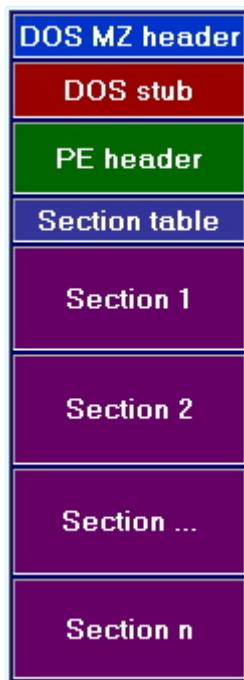
Tại sao chúng ta lại cần phải tìm hiểu về nó? Có 2 lý do chính như sau : Thứ nhất chúng ta muốn thêm các đoạn code vào trong những file thực thi (ví dụ : kỹ thuật **Keygen Injection** hoặc thêm các chức năng) và thứ hai là thực hiện công việc unpacking bằng tay (manual unpacking) các file thực thi. Hầu hết mọi sự quan tâm đều dồn về lý do thứ hai, đó là vì ngày nay hầu như các phần mềm shareware nào cũng đều được “**Packed**” lại với mục đích là làm giảm kích thước của file đồng thời cung cấp thêm một lớp bảo vệ cho file.

Ở bên trong một file thực thi đã bị Packed thì các bảng import tables thường thường là đã bị thay đổi, làm mất hiệu lực và phần dữ liệu thì luôn bị mã hóa. Các chương trình packer sẽ chèn thêm mã lệnh (code) để unpack file trong bộ nhớ vào lúc thực thi và sau đó nhảy tới **OEP** (original entry point) (Đây là nơi mà chương trình gốc thực sự bắt đầu thực thi, thi hành.). Nếu chúng ta tìm được cách để (dump) kết xuất vùng nhớ này sau khi mà chương trình packer hoàn tất được quá trình unpacking file thực thi, đồng thời thêm vào đó chúng ta cũng cần phải chỉnh sửa lại Section và bảng import tables trước khi mà ứng dụng của chúng ta sẽ run. Làm thế nào để chúng ta có thể thực hiện được điều này nếu như chúng ta không có hiểu biết tí tẹo nào về định dạng PE file ?

Chương trình thực thi được tôi sử dụng làm ví dụ xuyên suốt toàn bộ bài viết này là **BASECALC.EXE**, một chương trình rất hữu ích từ trang Web của Fravia, nó cho phép tính toán và chuyển đổi giữa các số hệ decimal, hex, binary và octal. Chương trình này được tác giả của nó coded bằng ngôn ngữ Borland Delphi 2.0, chính vì thế mà nó là một file lý tưởng để tôi lấy làm ví dụ minh họa làm thế nào trình biên dịch Borland để cho OriginalFirstThunks null. (Chi tiết hơn sẽ được đề cập ở phần sau).

1. Cấu trúc cơ bản (Basic Structure) :

Hình minh họa dưới đây sẽ cho chúng ta thấy được cấu trúc cơ bản của một PE file.



Ở mức tối thiểu nhất thì một PE file sẽ có 2 Sections : 1 cho đoạn mã (code) và 1 cho phần dữ liệu (data). Một chương trình ứng dụng chạy trên nền tảng Windows NT có 9 sections được xác định trước có tên là **.text** , **.bss** , **.rdata** , **.data** , **.rsrc** , **.edata** , **.idata** , **.pdata** , và **.debug** . Một số chương trình ứng dụng lại không cần tất cả những sections này, trong khi các chương trình khác có thể được định nghĩa với nhiều sections hơn để phù hợp với sự cần thiết riêng biệt của chúng.

Những sections mà hiện thời đang tồn tại và xuất hiện thông dụng nhất trong một file thực thi là :

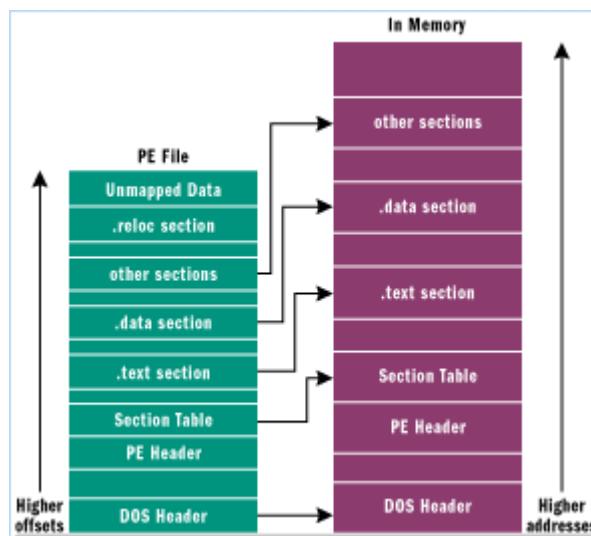
1. Executable Code Section, có tên là **.text** (Micro\$oft) hoặc là **CODE** (Borland).
2. Data Sections, có những tên như **.data**, **.rdata** hoặc **.bss** (Micro\$oft) hay **DATA** (Borland)
3. Resources Section, có tên là **.rsrc**
4. Export Data Section, có tên là **.edata**
5. Import Data Section. có tên là **.idata**
6. Debug Information Section, có tên là **.debug**

Những cái tên này thực sự là không thích hợp khi chúng bị lờ đi bởi hệ điều hành (OS) và chúng là tài liệu phục vụ cho lợi ích của các lập trình viên. Một điểm quan trọng khác nữa là cấu trúc của một PE file trên đĩa là chính xác , đúng đắn giống hệt như khi nó được nạp vào trong bộ nhớ vì vậy bạn có thể xác định thông tin chính xác của file trên đĩa mà bạn có thể sẽ muốn tìm kiếm nó khi file được nạp vào trong bộ nhớ.

Tuy nhiên nó không được sao chép lại một cách chính xác bên trong bộ nhớ. Các windows loader sẽ quyết định phần nào cần được ánh xạ lên bộ nhớ và bỏ qua những phần khác. Phần dữ liệu mà không được ánh xạ lên được đặt tại phía cuối của file sau bất kì phần nào mà sẽ được ánh xạ lên bộ nhớ ví dụ Debug Information.

Cũng vậy vị trí của một mục trong file trên đĩa sẽ luôn luôn khác biệt với vị trí của nó khi được nạp vào trong bộ nhớ bởi vì sự quản lý bộ nhớ ảo dựa trên các trang mà Windows sử dụng. Khi các sections được nạp vào trong bộ nhớ RAM chúng được căn để khớp với 4KB memory Pages, mỗi section sẽ bắt đầu trên

một Page mới. Một trường trong PE header sẽ thông báo cho hệ thống biết có bao nhiêu bộ nhớ cần được để riêng ra cho việc ánh xạ trong file. Bộ nhớ ảo được giải thích ở phần dưới đây.



Thuật ngữ bộ nhớ ảo (**virtual memory**) thay thế việc để cho Software truy cập trực tiếp lên bộ nhớ vật lý (physical memory), bộ xử lý và hệ điều hành tạo ra một lớp vô hình (invisible layer) giữa chúng. Bất kể lần nào một cổ găng được tạo ra để truy cập tới bộ nhớ , bộ vi xử lý sẽ tra cứu một “page table” để biết xem có những Process mà địa chỉ bộ nhớ vật lý đang thực sự được sử dụng. Nó sẽ không phải là một việc làm thiết thực để có một table entry cho mỗi byte của bộ nhớ (Page table sẽ lớn hơn tổng bộ nhớ vật lý), vì vậy thay thế việc bộ vi xử lý phân chia bộ nhớ thành các trang. Điều này có một số lợi thế như sau :

1. Nó cho phép sự tạo thành của những không gian địa chỉ phức tạp. Một không gian địa chỉ là một page table được cài đặt chỉ cho phép truy cập tới bộ nhớ mà thích hợp với chương trình hiện tại hoặc process. Nó đảm bảo rằng những chương trình bị cài đặt , cách ly hoàn toàn với các chương trình khác và một khi xảy ra lỗi khiến cho một chương trình bị crash thì nó sẽ không thể ảnh hưởng , hủy hoại tới không gian địa chỉ của các chương trình khác .
2. Nó cho phép bộ vi xử lý áp đặt những luật lệ nào đó đối với việc bộ nhớ được truy cập thế nào.Những sections được đòi hỏi , yêu cầu trong PE file bởi vì những khu vực khác nhau trong file được đòi hỏi một cách khác biệt bởi chương trình quản lý bộ nhớ khi một module được nạp. Tại thời điểm nạp , chương trình quản lý bộ nhớ thiết lập những quyền truy cập lên các trang bộ nhớ cho các sections khác nhau dựa trên những thiết lập của chúng trong Section header. Điều này sẽ quyết định rõ một section đã cho là có thể đọc được (readable) , có thể ghi được (writeable) hay có thể thực thi được (executable). Điều này có nghĩa là mỗi section phải được bắt đầu trên một trang mới. Tuy nhiên , kích thước trang mặc định cho hệ điều hành Windows là 4096 bytes (1000h) và nó sẽ là lãng phí để sắp các file thực thi vào một ranh giới 4KB Page trên đĩa khi mà điều này sẽ làm cho chúng trở nên quá lớn hơn mức cần thiết. Bởi vì điều này, PE header có hai trường alignment khác nhau là : **Section alignment và file alignment**. Section alignment là cách các sections được sắp trong bộ nhớ như đã nói ở trên. Còn **File Alignment** (sử dụng 512 bytes hay 200h) là cách các section được sắp trong file trên đĩa và là kích thước của nhiều sector để tối ưu quá trình loading (loading process).
3. Nó cho phép một file đánh số trang (paging file) được sử dụng trên ổ cứng để lưu trữ các trang một cách tạm thời từ bộ nhớ vật lý khi chúng không được sử dụng. Lấy ví dụ như sau, nếu một ứng dụng đã được nạp nhưng đang ở trong tình trạng rảnh rỗi (idle) ,không gian địa chỉ của nó có thể được đánh trang bên ngoài ổ đĩa để tạo ra không gian cho các ứng dụng khác cần được nạp vào trong bộ nhớ RAM. Nếu như tình hình đảo lộn , hệ điều hành có thể nạp một cách dễ dàng ứng dụng đầu tiên trở lại bộ nhớ RAM và hồi phục lại sự thi hành tại nơi mà nó bị ngưng lại . Một ứng dụng cũng có thể sử dụng nhiều bộ nhớ hơn là không gian hiện có của bộ nhớ vật lý bởi vì hệ

thống có thể sử dụng ổ cứng như là một nơi lưu trữ thứ cấp bất cứ khi nào mà bộ nhớ vật lý không còn đủ không gian lưu trữ.

Khi PE file đã được nạp vào bộ nhớ bởi windows loader, phiên bản trong bộ nhớ này được biết đến như là một **module**. Địa chỉ bắt đầu nơi mà ánh xạ file bắt đầu được gọi là một **HMODULE**. Một module trong bộ nhớ biểu diễn tất cả đoạn mã , dữ liệu và toàn bộ tài nguyên từ một file thực thi mà điều này là cần thiết cho sự thi hành khi mà thuật ngữ **Process** về cơ bản tham chiếu tới một không gian địa chỉ có lập mà có thể được sử dụng để running như là một module.

2. The DOS Header :

Tất cả các file PE bắt đầu bằng DOS Header , vùng này chiếm giữ 64 bytes đầu tiên của file. Nó được dùng trong trường hợp chương trình của bạn chạy trên nền DOS, do đó hệ điều hành DOS có thể nhận biết nó như là một file thực thi hợp lệ và thi hành DOS stub , phần mà đã được lưu trữ trực tiếp sau Header. Hầu hết DOS stub thường sử dụng hàm 9 của ngắt int 21h để hiện ra một chuỗi kí tự thông báo tương tự như sau : "**This program must be run under Microsoft Windows**" nhưng nó có thể là một chương trình DOS đang phát triển mạnh (full-blown DOS program) (Nói tóm lại là DOS Stub chỉ là một chương trình DOS EXE nhỏ hiển thị một thông báo lỗi thường là như trên, chính do header này được đặt nằm ở đầu của file , cho nên các virus DOS có thể lây nhiễm vào PE image chính xác tại DOS stub. Tuy nhiên chương trình DOS Stub vẫn còn được giữ lại vì lí do để tương thích với các hệ thống Windows 16-bit). Khi xây dựng một ứng dụng phát triển trên nền tảng Windows , chương trình linker liên kết một stub program mặc định có tên gọi là **WINSTUB.EXE** vào trong file thực thi của bạn. Bạn có thể ghi đè , phủ quyết cách hành xử của chương trình linker mặc định này bằng cách thay thế một chương trình MS-DOS-based của riêng bạn thay cho WINSTUB và sử dụng **-STUB**: một tùy chọn của chương trình linker khi liên kết file thực thi.

DOS Header là một cấu trúc được định nghĩa trong các file **windows.inc** hoặc **winnt.h** (Nếu như bạn có một chương trình dịch hợp ngữ hoặc một trình biên dịch đã được cài đặt trên máy , bạn sẽ tìm thấy các file này trong thư mục \include\). Nó có 19 thành phần (members) mà trong đó thành phần **magic** và **lfanew** là đáng chú ý.

```
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
```

Trong PE file , phần **magic** của DOS Header chứa giá trị **4Dh, 5Ah** (Đó chính là các kí tự “**MZ**”, viết tắt của **Mark Zbikowsky** một trong những người sáng tạo chính của MS-DOS), các giá trị này là dấu hiệu

thông báo cho chúng ta biết đây là DOS Header hợp lệ. MZ là 2 bytes đầu tiên mà bạn sẽ nhìn thấy trong bất kì một PE file nào , khi file đó được mở bằng một chương trình Hex editor. (Xem hình minh họa phía dưới).

Như bạn đã nhìn thấy trong hình minh họa phía trên, bạn thấy rằng phần **Ifanview** là một giá trị DWORD (tức là một Double Word = 4bytes) và nó nằm ở vị trí cuối cùng của DOS Header và紧跟 trước của nơi bắt đầu DOS Stub. Nó chứa offset của PE Header, có liên quan đến phần đầu file (file beginning). Windows Loader sẽ tìm kiếm offset này vì vậy nó có thể bỏ qua Dos Stub và đi trực tiếp tới PE Header.

Hình minh họa ở trên đã giúp ích cho chúng ta rất nhiều khi nó chỉ cho ta thấy rõ kích thước của từng phần tử. Điều này cho phép chúng ta truy xuất những thông tin mà chúng ta quan tâm dựa trên việc đếm số lượng các bytes từ điểm bắt đầu của section hoặc một điểm có thể nhận biết được.

Như chúng ta đã nói ở trên, DOS Header chiếm 64 bytes đầu tiên của file – ví dụ 4 hàng đầu được nhìn thấy trong một chương trình Hex Editor trong hình minh họa dưới đây.Giá trị DWORD cuối cùng trước điểm bắt đầu DOS Stub chứa những giá trị **00h 01h 00h 00h**. Để ý đến việc reverse trật tự byte , điều này sẽ giúp chúng ta biết **00 00 01 00h** là những offset nơi mà PE Header bắt đầu. PE Header bắt đầu với phần signatures của nó là **50h, 45h, 00h, 00h** (Các kí tự “PE” được đi kèm bởi các giá trị tận cùng là 0)

Nếu tại trường Signature của PE Header , bạn tìm thấy một NE signature ở đó chứ không phải là PE , thì lúc này bạn đang làm việc với một file NE Windows 16-bit. Cũng tương tự như vậy, nếu bạn thấy là **LE** nằm tại Signature field thì có nghĩa là nó cho ta biết đó là một trình điều khiển thiết bị ảo Window 3.x (VxD). Còn tại đó là một **LX** thì đó là dấu hiệu của một file cho OS/2 2.0

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	4D	5A	50	00	02	00	00	04	00	0F	00	FF	FF	00	00	; MZP.....yy..	
00000010h:	B8	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	;	
00000030h:	00	00	00	00	00	00	00	00	00	00	00	01	00	00	00	;	
00000040h:	B8	10	00	0E	1F	B4	09	CD	21	B8	01	4C	CD	21	90	; ^....!..Í!..LÍ!..	
00000050h:	54	68	69	73	20	70	72	6F	67	72	61	6D	20	6D	75	; This program mus	
00000060h:	74	20	62	65	20	72	75	6E	20	75	6E	64	65	72	20	; t be run under W	
00000070h:	69	6E	33	32	0D	0A	24	37	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	;	
00000090h:	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	;	
000000b0h:	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	;	
000000d0h:	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	;	
000000f0h:	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	; PE..L...^B*..	
00000110h:	00	00	00	00	E0	00	8E	81	0B	01	02	19	00	A0	02	; ..à.Ž..	

OKi.... tạm nghỉ chút xíu !! Chúng ta sẽ tiếp tục thảo luận trong phần tiếp theo của bài viết này. :)

3. The PE Header :

PE Header là thuật ngữ chung đại diện cho một cấu trúc được đặt tên là **IMAGE_NT_HEADERS** . Cấu trúc này bao gồm những thông tin thiết yếu được sử dụng bởi loader. **IMAGE_NT_HEADERS** có 3 thành phần và được định nghĩa trong file **windows.inc** như sau :

```

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

```

Signature là một DWORD chứa những giá trị như sau **50h, 45h, 00h, 00h** (Các kí tự “PE” được đi kèm bởi các giá trị tận cùng là 0).

FileHeader bao gồm 20 bytes tiếp theo của PE file ,nó chứa thông tin về sơ đồ bố trí vật lý và những đặc tính của file. Ví dụ : số lượng các sections.

OptionalHeader luôn luôn hiện diện và được tạo thành bởi 224 bytes tiếp theo . Nó chứa thông tin về sơ đồ Logic bên trong của một file PE. Ví dụ : **AddressOfEntryPoint**. Kích thước của nó được qui định bởi một thành phần của FileHeader. Các cấu trúc của những thành phần này cũng được định nghĩa trong file **windows.inc**

FileHeader được định nghĩa giống như hình minh họa dưới đây :

```

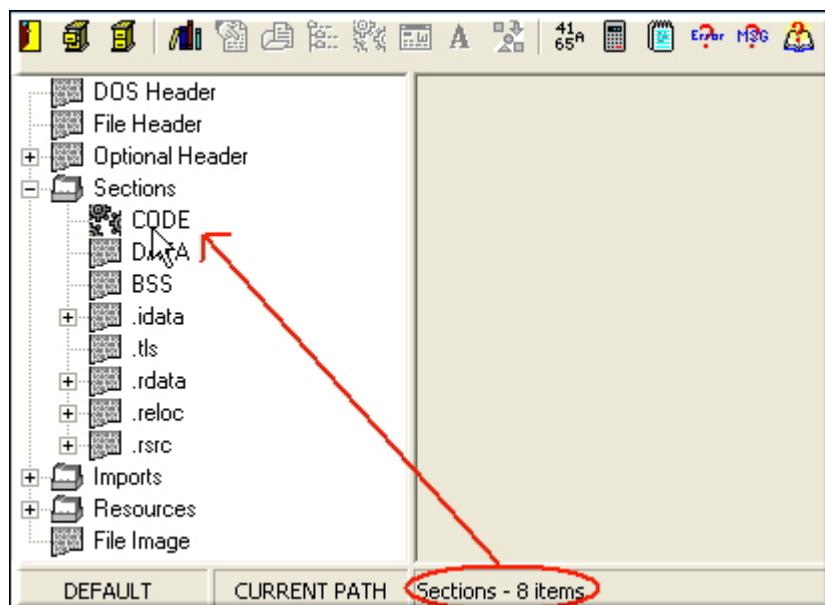
IMAGE_FILE_HEADER| STRUCT
    Machine           WORD      ?
    NumberOfSections WORD      ?
    TimeDateStamp    DWORD    ?
    PointerToSymbolTable DWORD    ?
    NumberOfSymbols  DWORD    ?
    SizeOfOptionalHeader WORD    ?
    Characteristics  WORD    ?
IMAGE_FILE_HEADER ENDS

```

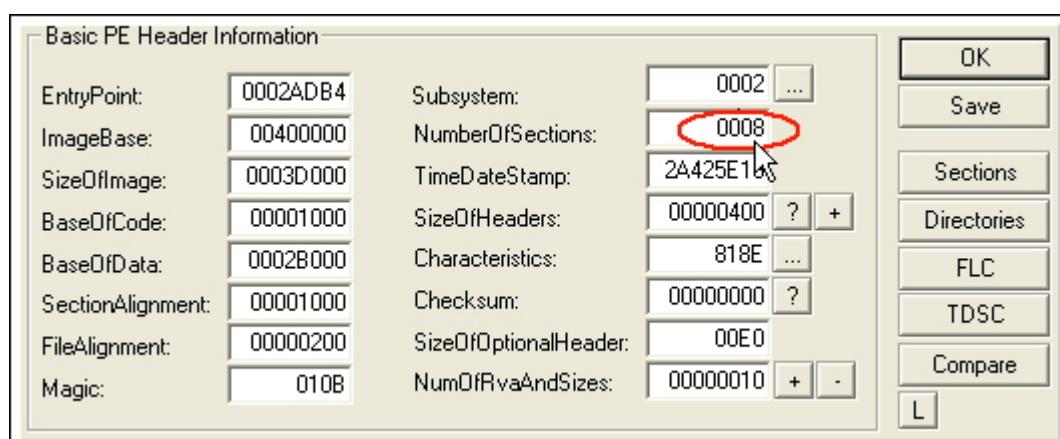
Hầu hết những thành phần này không còn hữu ích đối với chúng ta nhưng chúng ta phải thay đổi thành phần **NumberOfSections** nếu như chúng ta muốn thêm hoặc xóa bất kì sections nào trong một PE File. **Characteristics** bao gồm các cờ mà các cờ này xác định những thể hiện để chúng ta biết được PE File mà chúng ta làm việc là một file có thể thực thi (executable) hay là một file DLL. Quay trở lại ví dụ của chúng ta trong màn hình HexEditor, chúng ta có thể tìm thấy **NumberOfSections** bằng việc đếm một DWORD và một WORD (6 bytes) từ chỗ bắt đầu của PE Header (Tức là giá trị DWORD chính là **Signature** còn giá trị WORD chính là **Machine**) (note : trường **NumberOfSections** được sử dụng bởi viruses vì nhiều lí do khác nhau. Lấy ví dụ , trường này có thể bị thay đổi bằng cách viruses sẽ gia tăng nó lên để thêm một section mới vào PE image và đặt đoạn virus body vào section đó – Các hệ thống Windows NT có thể chấp nhận tối đa 96 sections trong một PE file. Trên hệ thống sử dụng Win95 thì không kiểm tra kĩ phần section number).. Xem hình minh họa dưới đây :

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
000000000h:	4D	5A	50	00	02	00	00	00	04	00	0F	00	FF	FF	00	00 ; MZP.....ÿÿ..
000000010h:	B8	00	00	00	00	00	00	40	00	1A	00	00	00	00	00 ;@.....	
000000020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 ;	
000000030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00 ;	
000000040h:	BA	10	00	0E	1F	B4	09	CD	21	B8	01	4C	CD	21	90	; °....'í!.LÍ!Ø
000000050h:	54	68	69	73	20	70	72	6F	67	72	61	6D	20	6D	75	; This program mus
000000060h:	74	20	62	65	20	72	75	6E	20	75	6E	64	65	72	20	; t be run under W
000000070h:	69	6E	33	32	OD	0A	24	37	00	00	00	00	00	00	00 ; in32..\$7.....	
000000080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 ;	
000000090h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 ;	
0000000a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 ;	
0000000b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 ;	
0000000c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 ;	
0000000d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 ;	
0000000e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 ;	
0000000f0h:	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 ; PE..L...^B*....	
00000110h:	00	00	00	00	E0	00	81	81	OB	01	02	19	00	A0	02	;à.Žo.....

Điều này có thể được kiểm tra lại bằng cách sử dụng bất cứ một công cụ PE nào. Ví dụ : Công cụ **PEBrowsePro**



Hoặc sử dụng một công cụ khá nổi tiếng là **LorDPE** :



Hoặc thậm chí nếu bạn đang sử dụng PEiD bạn cũng có thể kiểm nghiệm được điều này bằng cách nhấp vào button là **Subsystem** :

Basic Information	
EntryPoint:	0002ADB4
ImageBase:	00400000
SizeOfImage:	0003D000
BaseOfCode:	00001000
BaseOfData:	0002B000
SectionAlignment:	00001000
FileAlignment:	00000200
Magic:	010B
SubSystem:	0002
NumberOfSections:	0008
TimeDateStamp:	2A42AE19
SizeOfHeaders:	00000400
Characteristics:	818E
Checksum:	00000000
SizeOfOptionalHeader:	00E0
NumOfRvaAndSizes:	00000010

Chú ý : PEiD là một công cụ cực kì hữu ích – Chức năng chính của nó là dùng để scan Executable files và chỉ cho chúng ta biết được loại Packer mà File này được sử dụng cho việc nén và protect file. Ngoài ra đi kèm với PEiD là một Plugin không kém phần quan trọng, đó chính là **Krypto ANALyser** . Khi bạn sử dụng Plug-in này thì nó sẽ cho chúng ta biết được file đó có sử dụng những mật mã (**cryptography**) gì. Chẳng hạn : CRC, MD4, MD5 hoặc SHA v...v.... Thậm chí công cụ này cũng sử dụng các danh sách được người dùng định nghĩa về các **Packer signatures**. Tóm lại PEiD là công cụ đầu tiên được sử dụng khi chúng ta bắt tay vào công việc unpacking.

Chúng ta tiếp tục nghiên cứu tới thành phần tiếp theo là **OptionalHeader**, nó chiếm 224 bytes , trong đó 128 bytes cuối cùng sẽ chứa thông tin về **Data Directory**. Nó được định nghĩa giống như hình minh họa dưới đây :

```

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 – RVA (địa chỉ ảo tương đối) của câu lệnh đầu tiên mà sẽ được thực thi khi chương trình PE Loader sẵn sàng để run PE File (thông thường nó trỏ tới section .text hay CODE). Nếu như bạn muốn làm thay đổi luồng của thứ tự thực hiện , bạn cần phải thay đổi lại giá trị trong trường này thành một RVA mới và do đó câu lệnh tại giá trị RVA mới này sẽ được thực thi đầu tiên. Các chương trình Packer thường thay thế giá trị này bằng giá trị decompression stub của chúng, sau đó sự thi hành sẽ nhảy trở về điểm bắt đầu của chương trình – hay còn gọi với tên thông dụng là OEP. Một lưu ý thêm nữa là ở chế độ bảo vệ **StarForce** thì CODE section sẽ không có mặt , hiện diện trong file trên đĩa nhưng lại được ghi lên bộ nhớ ảo trong quá trình thực thi. Vì thế mà giá trị trong trường này là một VA (xem thêm phần phụ lục sẽ được đề cập bên dưới). (note : Đây thực sự là một trường cốt yếu và cực kì quan trọng bởi vì trường này sẽ bị thay đổi bởi hầu hết các kiểu lây nhiễm virus để trỏ tới điểm thực thi thực sự của virus code)

ImageBase – Địa chỉ nạp được ưu tiên cho PE File. Lấy ví dụ : Nếu như giá trị trong trường này là 400000h, PE Loader sẽ cố gắng để nạp file vào trong không gian địa chỉ ảo mà bắt đầu tại 400000h. Từ “được ưu tiên” ở đây có nghĩa là PE Loader không thể nạp file tại địa chỉ đó nếu như có một module nào khác đã chiếm giữ vùng địa chỉ này. 99 % các trường hợp giá trị của **ImageBase** luôn là 400000h

SectionAlignment – Phần liên kết của các Sections trong bộ nhớ. . Khi file thực thi được ánh xạ vào trong bộ nhớ, thì mỗi section phải bắt đầu tại một địa chỉ ảo mà là một bội số của giá trị này. Giá trị của trường này nhỏ nhất là 0x1000(4096 bytes), nhưng trình các trình linkers của Borland thường sử dụng các giá trị mặc định lớn hơn, ví dụ như là 0x10000(64KB). Lấy ví dụ như sau : Nếu giá trị tại trường này là 4096 (1000h), thì mỗi section tiếp theo sẽ phải bắt đầu tại vị trí mà section trước đó cộng với 4096 bytes. Nếu section đầu tiên là tại 401000h và kích thước của nó là 10 bytes, vậy section tiếp theo là tại 402000h

cho dù là không gian địa chỉ giữa 401000h và 402000h sẽ hầu như không được sử dụng.(note: hầu hết các Win32 viruses sử dụng trường này để tính toán vị trí chính xác của virus body nhưng lại không thay đổi trường này).

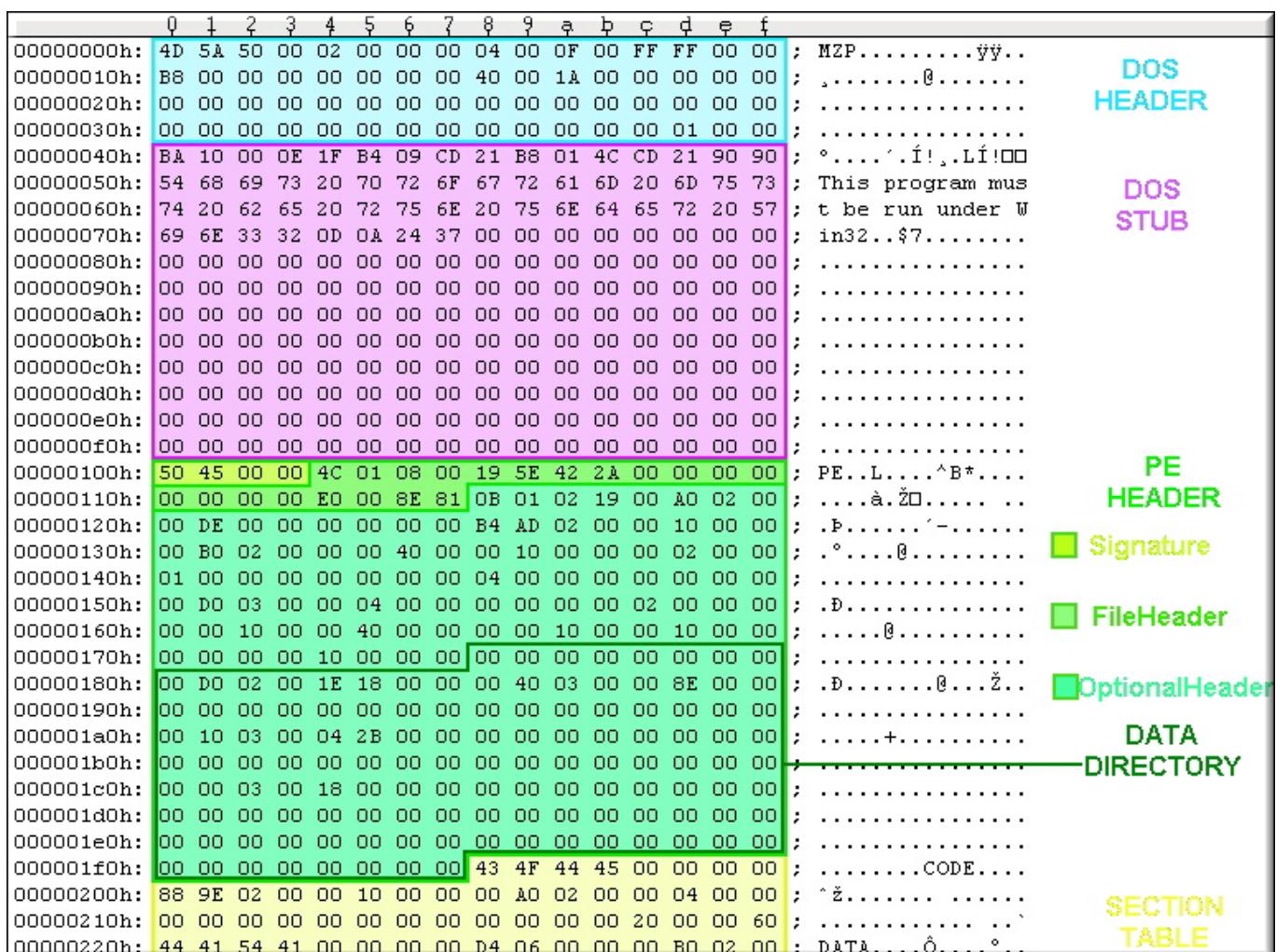
FileAlignment – Phần liên kết của các Section trong file. Lấy ví dụ : nếu giá trị cụ thể của trường này là 512 (200h), thì mỗi section tiếp theo sẽ phải bắt đầu tại vị trí mà sections trước đó cộng với 200h. Nếu section đầu tiên là tại offset 200h, và có kích thước là 10 bytes, vậy thì section tiếp theo sẽ được định vị tại địa chỉ offset là 400h : Không gian giữa file offsets 522 và 1024 là không sử dụng được/hoặc không được định nghĩa.

SizeOfImage - Toàn bộ kích thước của PE image trong bộ nhớ. Nó là tổng của tất cả các headers và sections được liên kết tới **SectionAlignment**.

SizeOfHeaders - Kích thước của tất cả các headers + section table.Nói tóm lại , giá trị này là bằng kích thước file trừ đi kích thước được tổng hợp của toàn bộ sections trong file. Bạn cũng có thể sử dụng giá trị này như một file offset của Section đầu tiên trong PE file.

DataDirectory – Một mảng của 16 **IMAGE_DATA_DIRECTORY** structures, mỗi một phần có liên quan tới một cấu trúc dữ liệu quan trọng trong PE File chẳng hạn như **import address table**. Cấu trúc quan trọng này sẽ được thảo luận chi tiết trong những phần tiếp theo.

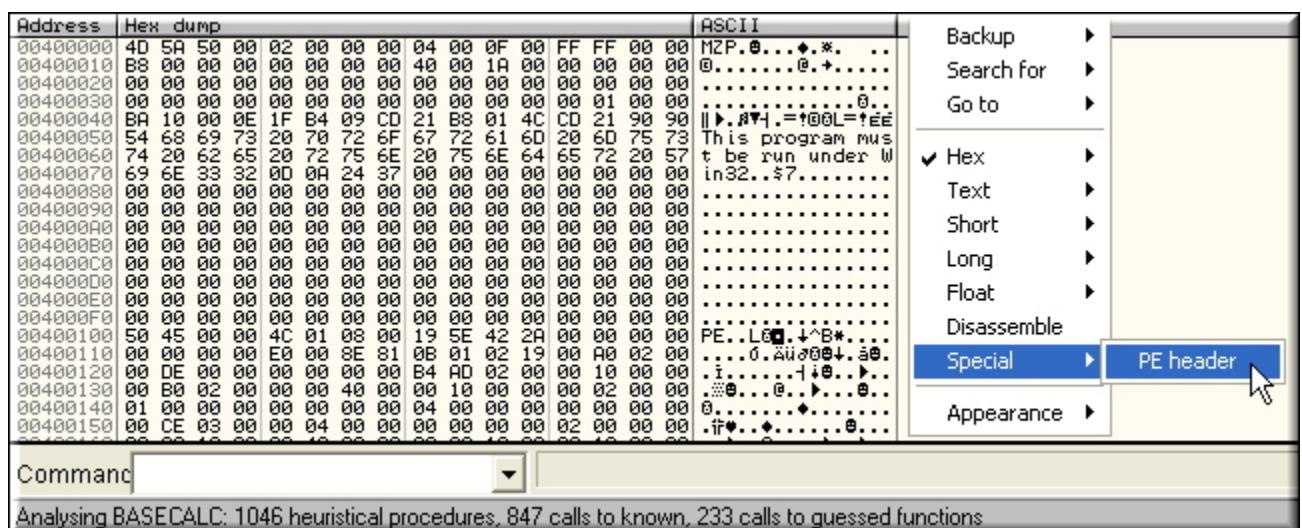
Cách bố trí mọi thứ của PE Header có thể được quan sát một cách trực quan thông qua hình ảnh minh họa sau đây trong chương trình HexEditor. Chú ý rằng **DOS Header** và phần của **PE Header** là luôn luôn cùng kích thước (and shape) khi được quan sát trong chương trình HexEditor. Phần **DOS Stub** có thể thay đổi theo kích thước :



Bên cạnh các công cụ PE đã được đề cập ở trên, chương trình debug được ưa thích là OllyDbg cũng có thể phân tích được PE Headers thông qua việc hiện thị thông tin một cách đầy đủ và có ý nghĩa. Dùng OllyDbg load file ví dụ của chúng ta vào trong Olly và nhấn **Alt + M** hoặc bấm vào nút **M** để mở cửa sổ Memory Map - cửa sổ này sẽ cho chúng ta thấy được PE File đã được nạp vào bộ nhớ.

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

Tiếp theo bạn nhấp chuột phải trên PE Header và chọn **Dump in CPU**. Sau đó trong cửa sổ Hex window , lại nhấp chuột phải một lần nữa và chọn **Special --> PE Header** .



Chúng ta sẽ có được thông tin như sau :

Address	Hex dump	Data	Comment
00400100	50 45 00 00	ASCII "PE"	PE signature (PE)
00400104	4C01	DW 014C	Machine = IMAGE_FILE_MACHINE_I386
00400106	0800	DW 0008	NumberOfSections = 8
00400108	195E422A	DD 2A425E19	TimeStamp = 2A425E19
0040010C	00000000	DD 00000000	PointerToSymbolTable = 0
00400110	00000000	DD 00000000	NumberOfSymbols = 0
00400114	E000	DW 00E0	SizeOfOptionalHeader = E0 (224.)
00400116	8E81	DW 818E	Characteristics = EXECUTABLE_IMAGE 32BIT_MA
00400118	0001	DW 010B	MagicNumber = PE32
0040011A	02	DB 02	MajorLinkerVersion = 2
0040011B	19	DB 19	MinorLinkerVersion = 19 (25.)
0040011C	00A00200	DD 0002A000	SizeOfCode = 2A000 (172032.)
00400120	000E0000	DD 0000E000	SizeOfInitializedData = DE00 (56832.)
00400124	00000000	DD 00000000	SizeOfUninitializedData = 0
00400128	B4AD0200	DD 0002ADB4	AddressOfEntryPoint = 2ADB4
0040012C	00100000	DD 00001000	BaseOfCode = 1000
00400130	000B00200	DD 0002B000	BaseOfData = 2B000
00400134	00004000	DD 00400000	ImageBase = 400000
00400138	00100000	DD 00001000	SectionAlignment = 1000
0040013C	00020000	DD 00000200	FileAlignment = 200
00400140	0100	DW 0001	MajorOSVersion = 1
00400142	0000	DW 0000	MinorOSVersion = 0
00400144	0000	DW 0000	MajorImageVersion = 0
00400146	0000	DW 0000	MinorImageVersion = 0
00400148	0400	DW 0004	MajorSubsystemVersion = 4
0040014A	0000	DW 0000	MinorSubsystemVersion = 0
0040014C	00000000	DD 00000000	Reserved
00400150	00CE0300	DD 0003CE00	SizeOfImage = 3CE00 (249344.)
00400154	00040000	DD 00004000	SizeOfHeaders = 400 (1024.)
00400158	00000000	DD 00000000	CheckSum = 0
0040015C	0200	DW 0002	Subsystem = IMAGE_SUBSYSTEM_WINDOWS_GUI
0040015E	0000	DW 0000	DLLCharacteristics = 0
00400160	00001000	DD 00100000	SizeOfStackReserve = 100000 (1048576.)
00400164	00400000	DD 00004000	SizeOfStackCommit = 4000 (16384.)
00400168	00001000	DD 00100000	SizeOfHeapReserve = 100000 (1048576.)
0040016C	00100000	DD 00001000	SizeOfHeapCommit = 1000 (4096.)
00400170	00000000	DD 00000000	LoaderFlags = 0
00400174	10000000	DD 00000010	NumberOfRvaAndSizes = 10 (16.)
00400178	00000000	DD 00000000	Export Table address = 0
0040017C	00000000	DD 00000000	Export Table size = 0
00400180	000002000	DD 0002D000	Import Table address = 20000
00400184	1E180000	DD 0000181E	Import Table size = 181E (6174.)
00400188	00400300	DD 00034000	Resource Table address = 34000

Command

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

4. The Data Directory :

Tóm tắt lại phần trước , chúng ta đã biết được rằng **Data Directory** là 128 bytes cuối cùng của **OptionalHeader** , và lần lượt là những thành phần cuối cùng của PE Header **IMAGE_NT_HEADERS**.

Như chúng ta đã từng nói, **Data Directory** là một mảng của 16 cấu trúc **IMAGE_DATA_DIRECTORY** structures, cứ mỗi 8 bytes thì mỗi phần lại có liên quan với một cấu trúc dữ liệu quan trọng trong PE File. Mỗi mảng tham chiếu tới một mục đã được định nghĩa trước , ví dụ như là **import table** . Cấu trúc của **Data Directory** có 2 thành phần mà bao gồm thông tin về vị trí và kích thước của cấu trúc dữ liệu trong những điều đã bàn đến :

```
IMAGE_DATA_DIRECTORY STRUCT
    VirtualAddress      DWORD      ?
    iSize                DWORD      ?
IMAGE_DATA_DIRECTORY ENDS
```

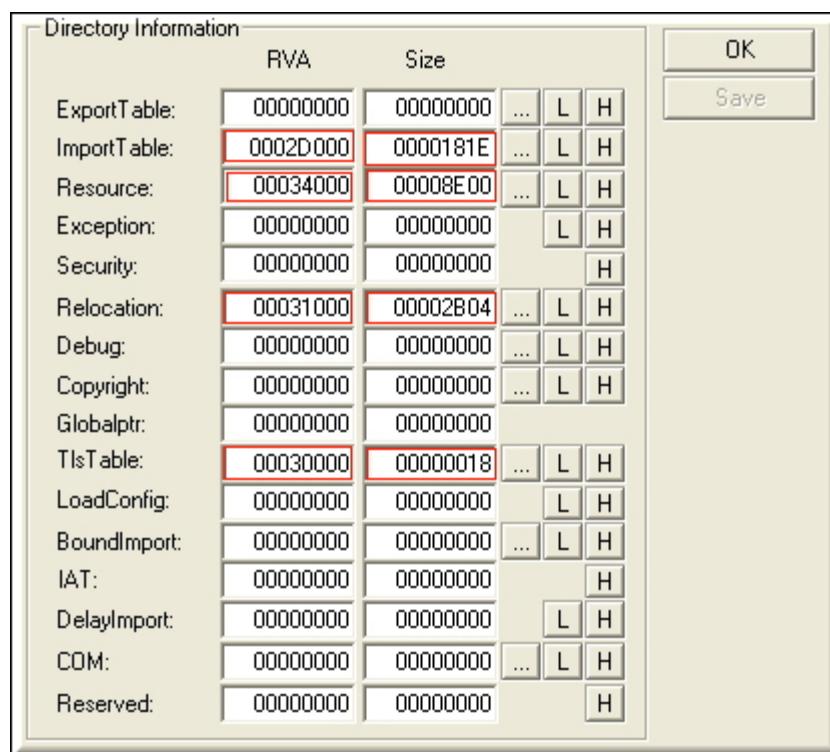
VirtualAddress là một địa chỉ ảo tương đối (relative virtual address) của cấu trúc dữ liệu (xem ở phần sau)

iSize bao gồm kích thước theo bytes của cấu trúc dữ liệu.

16 directories mà những cấu trúc này tham chiếu đến , bản thân chúng được định nghĩa trong file **window.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

Lấy ví dụ , chúng ta sử dụng chương trình LordPE. Trong LordPE , phần **Data Directory** cho file ví dụ của chúng ta chỉ chứa 4 thành phần (đã được tô khoanh màu đỏ trong hình vẽ). 12 thành phần còn lại không được sử dụng và được điền giá trị là 0 :



Như các bạn đã thấy trong hình minh họa ở trên, trường “import table” bao gồm thông tin về RVA và kích thước của **IMAGE_IMPORT_DESCRIPTOR** array – the Import Directory. Trong chương trình HexEditor, hình minh họa bên dưới đây chỉ cho chúng ta thấy PE Header với phần data directory được tô nét ngoài bằng màu đỏ. Mỗi một khu vực được khoanh này biểu diễn cho một cấu trúc **IMAGE_DATA_DIRECTORY**. Giá trị DWORD đầu tiên chính là **VirtualAddress** còn giá trị cuối cùng chính là **isize**.

000000100h:	50 45 00 00 4C 01 08 00 19 5E 42 2A 00 00 00 00 ; PE...L....^B*.....
000000110h:	00 00 00 00 E0 00 8E 81 0B 01 02 19 00 A0 02 00 ;à.Ž.....
000000120h:	00 DE 00 00 00 00 00 B4 AD 02 00 00 10 00 00 ; .P.....'-.....
000000130h:	00 B0 02 00 00 00 40 00 00 10 00 00 00 02 00 00 ; .°.....@.....
000000140h:	01 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 ;
000000150h:	00 D0 03 00 00 04 00 00 00 00 00 00 02 00 00 00 ; .D.....
000000160h:	00 00 10 00 00 40 00 00 00 00 10 00 00 10 00 00 ;
000000170h:	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 ;
000000180h:	00 D0 02 00 1E 18 00 00 00 40 03 00 00 8E 00 00 ; .D.....@.....ž.....
000000190h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
0000001a0h:	00 10 03 00 04 2B 00 00 00 00 00 00 00 00 00 00 ;
0000001b0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
0000001c0h:	00 00 03 00 18 00 00 00 00 00 00 00 00 00 00 00 ;
0000001d0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
0000001e0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
0000001f0h:	00 00 00 00 00 00 00 00 43 4F 44 45 00 00 00 00 ;
00000200h:	88 9E 02 00 00 10 00 00 00 A0 02 00 00 04 00 00 ; ^ž.....

The 16 Data Directories
Import Table
Resource
Relocation
TLS Table

Trong hình minh họa trên, thì Import Table được tô bằng màu hồng. 4 bytes đầu tiên là RVA 02D000h (NB reserver oder). Kích thước của **Import Table** là 181Eh bytes. Như chúng ta đã nói ở trên thì vị trí của những data directories từ phần đầu của PE Header là luôn luôn giống nhau. Ví dụ : giá trị DWORD 80 bytes từ phần đầu của PE Header luôn luôn là RVA của Import Table.

Để xác định được vị trí của một directory đặc biệt, bạn xác định rõ địa chỉ tương đối từ data directory. Sau đó sử dụng địa chỉ ảo để xác định section nào directory ở trong. Một khi bạn phân tích section nào chứa directory , thì Section Header cho section đó sau đó sẽ được sử dụng để tìm ra offset chính xác.

5. The Section Table :

Section Table là thành phần tiếp theo ngay sau PE Header.Nó là một mảng của những cấu trúc **IMAGE_SECTION_HEADER**, mỗi phần tử sẽ chứa thông tin về một section trong PE File ví dụ như thuộc tính của nó và offset ảo (virtual offset) . Các bạn hãy nhớ lại rằng số lượng các sections chính là thành phần thứ 2 của FileHeader (6 bytes từ chỗ bắt đầu của PE Header). Nếu có 8 sections trong PE File, thì sẽ có 8 bản sao của cấu trúc này trong table.Mỗi một cấu trúc Header (header structure) là 40 bytes và sẽ không có thêm “padding” giữa chúng (Padding ở đây có nghĩa là sẽ không chèn thêm các bytes có giá trị 00h vào).Cấu trúc này được định nghĩa trong file **windows.inc** như sau :

```

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

```

Xin nhắc lại một lần nữa , không phải tất cả các thành phần trên đều hữu ích. Tôi sẽ chỉ miêu tả những thành phần thực sự là quan trọng mà thôi.

Name1 - (NB this field is 8 bytes) Tên này chỉ là một nhãn và thậm chí là có thể để trống. Chú ý rằng đây không phải là một chuỗi ASCII vì vậy nó không cần phải kết thúc bằng việc thêm các số 0.

VirtualSize – (DWORD union) Kích thước thật sự của section's data theo bytes. Nó có thể nhỏ hơn kích thước của section trên đĩa (SizeOfRawData) và sẽ là những gì mà trình loader định rõ ví trí trong bộ nhớ cho section này.

VirtualAddress – RVA của section. Trình PE loader sẽ phân tích và sử dụng giá trị trong trường này khi nó ánh xạ section vào trong bộ nhớ. Vì vậy nếu giá trị trong trường này là 1000h và PE File được nạp tại địa chỉ 400000h , thì section sẽ được nạp tại địa chỉ là 401000h.

SizeOfRawData – Kích thước của section's data trong file trên đĩa, được làm tròn lên bởi số tiếp theo của sự liên kết file bởi trình biên dịch.

PointerToRawData – (Raw Offset) thành phần này thực sự rất hữu dụng bởi vì nó là offset từ vị trí bắt đầu của file cho tới phần section's data. Nếu nó có giá trị là 0 , thì section's data không được chứa trong file và sẽ không bị bó buộc vào thời gian nạp (load time). Trình PE Loader sẽ sử dụng giá trị trong trường này để tìm kiếm phần data trong section là ở đâu trong file.

Characteristics - Bao gồm các cờ ví dụ như section này có thể chứa executable code, initialized data , uninitialized data , có thể được ghi hoặc đọc (Xem thêm phần phụ lục)

NOTE : Khi bạn tiến hành tìm kiếm một section cụ thể nào đó , nó có thể phớt lờ toàn bộ PE Header và bắt đầu phân tích section headers bằng cách tìm kiếm section name trong cửa sổ ASCII của chương trình HexEditor của bạn.

Quay trở lại ví dụ của chúng ta , trong cửa sổ HexEditor file của chúng ta có 8 sections như chúng ta đã nhìn thấy trong section PE Header.

000001f0h:	00 00 00 00 00 00 00 00	43 4F 44 45 00 00 00 00	00 00 00 00 00 00 00 00	; ...CODE...
00000200h:	88 9E 02 00 00 10 00 00	A0 02 00 00 00 04 00 00	00 00 00 00 00 00 00 00	; ...DATA...
00000210h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 20 00 00 00 00 00 60	; ...BSS...
00000220h:	44 41 54 41 00 00 00 00	D4 06 00 00 00 00 00 00	B0 02 00 00 00 00 00 00	; ...idata...
00000230h:	00 08 00 00 00 A4 02 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...idata...
00000240h:	00 00 00 40 00 00 C0	42 53 53 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...idata...
00000250h:	78 06 00 00 00 C0 02 00	00 00 00 00 00 00 00 00	00 AC 02 00 00 00 00 00	; ...idata...
00000260h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...idata...
00000270h:	2E 69 64 61 74 61 00 00	1E 18 00 00 00 00 D0 02 00	00 00 00 00 00 00 00 00	; ...idata...
00000280h:	00 1A 00 00 00 AC 02 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...idata...
00000290h:	00 00 00 40 00 00 C0	2E 74 6C 73 00 00 00 00	00 00 00 00 00 00 00 00	; ...idata...
000002a0h:	08 00 00 00 F0 02 00 00	00 00 00 00 00 00 00 00	00 C6 02 00 00 00 00 00	; ...idata...
000002b0h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...idata...
000002c0h:	2E 72 64 61 74 61 00 00	18 00 00 00 00 00 00 03	00 00 00 00 00 00 00 00	; ...idata...
000002d0h:	00 02 00 00 00 C6 02 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...idata...
000002e0h:	00 00 00 40 00 00 50	2E 72 65 6C 6F 63 00 00	00 00 00 00 00 00 00 00	; ...idata...
000002f0h:	04 2B 00 00 00 10 03 00	00 2C 00 00 00 C8 02 00	00 00 00 00 00 00 00 00	; ...idata...
00000300h:	00 00 00 00 00 00 00 00	00 00 00 00 00 40 00 00	50 00 00 00 00 00 00 00	; ...idata...
00000310h:	2E 72 73 72 63 00 00 00	00 8E 00 00 00 00 40 03 00	00 00 00 00 00 00 00 00	; ...rsrc...
00000320h:	00 8E 00 00 00 F4 02 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...rsrc...
00000330h:	00 00 00 40 00 00 50	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...rsrc...
00000340h:	00 00 00 00 00 D0 03 00	00 00 00 00 00 00 00 00	82 03 00 00 00 00 00 00	; ...rsrc...
00000350h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 40 00 00 00 00 00 00	; ...rsrc...
00000360h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...rsrc...
00000370h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...rsrc...
00000380h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	; ...rsrc...

PointerToRawData
fields highlighted

Sau khi có được **Section Headers** chúng ta sẽ tìm kiếm các sections. Trong file ở trên đĩa, mỗi section bắt đầu tại một offset mà là bội số lần của giá trị **FileAlignment** được tìm thấy trong **OptionalHeader**. Giữa các section's data sẽ là các byte 00 được thêm vào.

Khi được nạp lên RAM, các sections luôn luôn bắt đầu trên một ranh giới trang (page boundary) vì vậy byte đầu tiên của mỗi section tương ứng với một trang bộ nhớ (memory page). Các trang trên những bộ vi xử lý x86 CPU là 4KB aligned, trong khi trên IA-64 là 8KB aligned. Giá trị liên kết (alignment value) này được lưu trữ trong **SectionAlignment**, và cũng được lưu trong **OptionalHeader**.

Lấy một ví dụ, nếu như **OptionalHeader** kết thúc tại file offset 981 và FileAlignment là 512, thì section đầu tiên sẽ bắt đầu tại byte 1024. Chú ý rằng bạn có thể tìm những section thông qua PointerToRawData hoặc là VirtualAddress, vì vậy không cần phải lo ngại băn khoăn về alignments.

Trong hình minh họa ở trên, **ImportData Section (.idata)** sẽ bắt đầu tại offset 0002AC00h (highlighted pink, NB reverse byte order) từ vị trí bắt đầu của file. Kích thước của nó, do đã được qui định là DWORD nên nó sẽ là 1A00h bytes.

6. The PE File Sections :

Là những sections chứa nội dung chính của file, bao gồm code, data, resources và những thông tin khác của file thực thi. Mỗi section có một Header và một body (dữ liệu thô – raw data : là dữ liệu chưa được xử lý hoặc chưa được định khuôn thức, nó chưa được sắp xếp, biên tập sửa chữa hoặc chưa được biểu diễn lại dưới dạng dễ truy tìm và phân tích). Những **Section Headers** thì được chứa trong **Section Table** nhưng những Section Bodies lại không có một cấu trúc file cứng rắn. Chúng có thể được sắp xếp hầu như theo bất kì cách nào khi một trình linker muốn tổ chức chúng, với điều kiện là Header được điền thông tin đầy đủ để có thể giải mã dữ liệu.

Một chương trình ứng dụng đặc thù trên hệ điều hành Windows NT có 9 sections được định nghĩa trước có tên là **.text**, **.bss**, **.rdata**, **.rsrc**, **.edata**, **.idata**, **.pdata** và **.debug**. Một vài chương trình không cần phải có đủ tất cả các sections này, trong khi một số chương trình ứng dụng khác lại định nghĩa thêm nhiều sections khác để phù hợp với những yêu cầu riêng biệt của chúng.

Executable Code Section :

Trong hệ điều hành Windows NT tất cả các đoạn mã (code segment) tập trung vào một sections đơn lẻ được gọi là **.text** hoặc là **CODE**. Từ khi hệ điều hành Windows NT chuyển sang sử dụng một hệ thống quản lý bộ nhớ ảo dựa trên trang, thì có một section code lớn dễ dàng hơn trong việc quản lý đối với hệ điều hành cũng như đối với những người phát triển ứng dụng. Section này cũng chứa điểm đột nhập (entry point) mà đã được đề cập ở phần trên và bảng jump thunk table trả tới IAT (xem thêm phần import theory)

Data Sections :

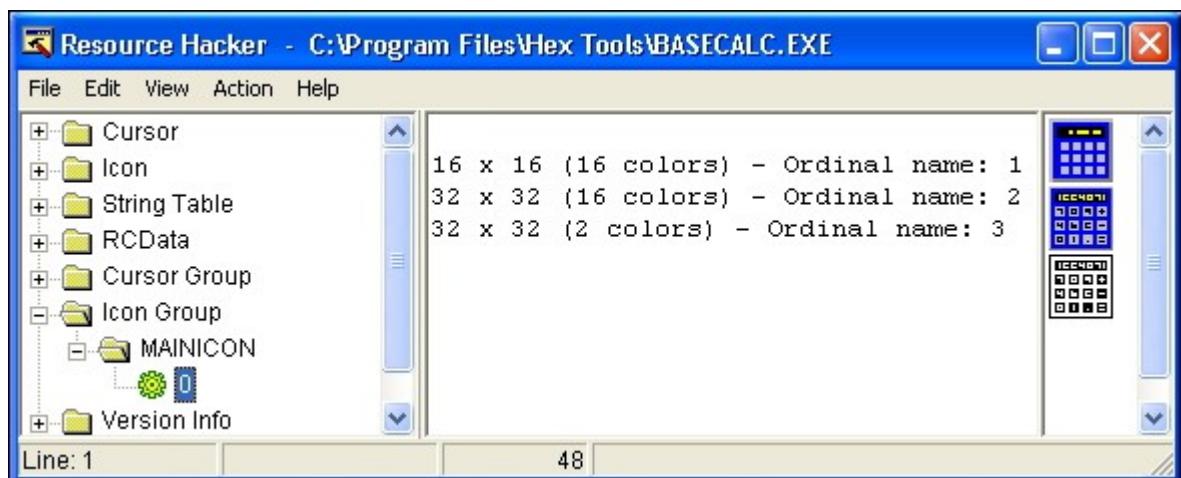
Section **.bss** biểu diễn dữ liệu không được khởi tạo cho ứng dụng, bao gồm toàn bộ các biến đã được khai báo là biến tĩnh trong một hàm hoặc là một module nguồn.

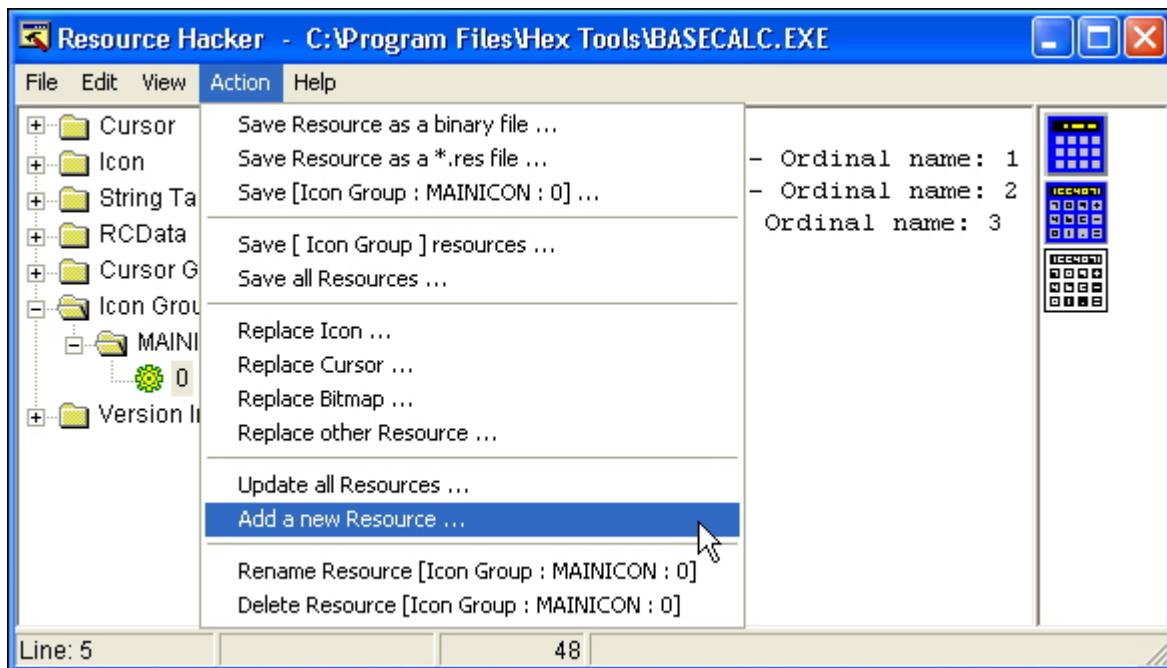
Section **.rdata** biểu diễn dữ liệu chỉ đọc ra (read – only), ví dụ như những chuỗi, các hằng, và thông tin thư mục debug.

Tất cả những biến khác (ngoại trừ những biến tự động, mà chỉ xuất hiện trên Stack) được lưu trữ trong Section **.data**. Đó là những ứng dụng hoặc là những biến toàn cục module.

Resources Section :

Section **.rsrc** chứa các thông tin resource cho một module. 16 bytes đầu tiên bao gồm một Header giống như những section khác, nhưng dữ liệu của Section này hơn nữa được cấu trúc vào trong một resource tree và được quan sát tốt nhất thông qua việc sử dụng một chương trình resource editor. Một chương trình khá nổi tiếng đó là **ResHacker**, đây là một chương trình miễn phí cho phép chỉnh sửa, thêm mới, xóa, thay thế và sao chép các Resources :





Đây là một chương trình rất mạnh phục vụ cho mục đích Cracking vì nó sẽ hiện thị một cách nhanh chóng các hộp thoại bao gồm cả những chi tiết về việc đăng ký sai cũng như các nag screens. Một ứng dụng shareware có thể thường bị Cracked chỉ bằng việc xóa bỏ resource hộp thoại nagscreen trong ResHacker.

Export Data Section :

Section **.edata** chứa Export Directory cho một chương trình ứng dụng hoặc file Dll. Khi biểu diễn, section này bao gồm các thông tin về tên và địa chỉ của những hàm exported functions. Chúng ta sẽ nói tiếp về vấn đề này sau, ở một phần rất quan trọng tiếp theo.

Import Data Section :

Section **.idata** chứa những thông tin khác nhau về những hàm imported functions bao gồm cả **Import Directory** và bảng **Import Address Table**. Chúng ta cũng sẽ nói tiếp về vấn đề này ở phần sau.

Debug Information Section :

Thông tin Debug được đặt ban đầu trong Section **.debug**. Định dạng PE File cũng hỗ trợ các file debug khác nhau (thường được nhận biết với phần mở rộng là .dbg) như là một cách thức của việc tập hợp thông tin debug tại một vị trí tập trung. Section debug chứa thông tin debug, nhưng những thư mục debug lại nằm trong Section **.rdata** như đã được đề cập ở phần trên. Mỗi một thư mục sẽ liên quan tới thông tin Debug trong Section **.debug**.

Base Relocations Section :

Khi mà trình linker tạo ra một file Exe, nó chuẩn bị một nơi mà tại đó file sẽ được ánh xạ vào trong bộ nhớ. Dựa trên điều này, trình linker sẽ đặt các địa chỉ thật của đoạn mã và những mục dữ liệu vào trong file thực thi. Nếu vì bất cứ lý do gì file thực thi kết thúc quá trình nạp ở một nơi nào đó nếu không trong phạm vi không gian địa chỉ ảo, thì những địa chỉ này sẽ bị trình linker đặt vào trong image không đúng. Thông tin được lưu trong Section **.reloc** cho phép trình PE loader fix những địa chỉ này trong loaded image vì vậy chúng sẽ lại chính xác. Mặt khác, nếu trình loader có thể nạp file tại những địa chỉ base address được thửa nhận bởi trình linker, thì dữ liệu Section **.reloc** là không cần thiết và bị lờ đi.

Các mục trong section .reloc được gọi bởi Base relocation vì sự sử dụng của chúng phụ thuộc vào địa chỉ base address của loaded image. Base Relocation đơn giản chỉ là một danh sách của các vị trí trong image mà yêu cầu một giá trị được thêm vào chúng. Định dạng của dữ liệu base relocation hơi phức tạp. Các mục base relocation được nén (packed) trong một chuỗi của các phần đệm dài biến đổi. Mỗi phần diễn tả các Relocation thay thế cho một trang 4KB trong image.

Hãy xem một ví dụ để hiểu cách hoạt động của base relocation. Một file thực thi được liên kết với một địa chỉ cơ sở của 0x10000. Tại offset 0x2134 bên trong image là một con trỏ chứa địa chỉ của một chuỗi. Chuỗi bắt đầu tại địa chỉ vật lý là 0x14002, vì vậy con trỏ sẽ chứa giá trị là 0x14002. Sau đó bạn nạp file, nhưng trình loader quyết định rằng nó cần phải ánh xạ image bắt đầu tại địa chỉ vật lý là 0x60000. Sự chênh lệch giữa trình linker dựa trên địa chỉ nạp và địa chỉ nạp thực sự được gọi là delta. Trong trường hợp ví dụ của chúng ta thì delta là 0x50000 bytes cao trong bộ nhớ, như vậy là chuỗi (bây giờ ở tại địa chỉ là 0x64002). Con trỏ tới chuỗi giờ đây không còn đúng nữa. File thực thi chứa một base relocation đại diện cho vị trí bộ nhớ (memory location) nơi mà con trỏ tới chuỗi trả về. Để giải quyết một base relocation, trình loader cộng thêm giá trị delta vào giá trị gốc ban đầu tại địa chỉ base relocation. Trong trường hợp của chúng ta, trình loader sẽ cộng giá trị delta là 0x50000 vào giá trị con trỏ ban đầu là (0x14002), và lưu kết quả trả lại là (0x64002) vào trong bộ nhớ của con trỏ. Vì thế chuỗi bây giờ sẽ có địa chỉ thực là tại 0x64002, vậy là mọi thứ đều tốt đẹp.

7. The Export Sections :

Section này có liên quan một cách đặc biệt tới các file DLLs. Phần thông tin được trích dưới đây từ **Win32 Programmer's Reference** sẽ giải thích tại sao :

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 easily. 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.

Các hàm có thể được exported bởi một DLL theo hai cách : “**by name**” hoặc “**by ordinal only**”. Một số thứ tự hay một chỉ số là một số 16-bit (WORD – sized) mà duy nhất chỉ ra một hàm trong một file DLL riêng biệt. Con số này là duy nhất chỉ bên trong file DLL nó tham chiếu tới. Chúng ta sẽ nói về exporting bằng số thứ tự ở phần sau.

Nếu như một hàm được exported bằng tên, khi các file DLL khác hoặc các file thực thi muốn gọi hàm này, chúng sẽ cùng sử dụng tên của hàm hoặc chỉ số của hàm trong hàm **GetProcAddress** mà trả về địa chỉ của hàm trong file DLL của nó. Tài liệu **Win32 Programmer's Reference** sẽ giải thích thêm về phương thức hoạt động của hàm **GetProcAddress** (Mặc dù trong thực tế thông tin về hàm này rất nhiều, không chỉ những tài liệu được viết bởi MS, những thông tin khác sẽ đề cập sau). Các bạn hãy chú ý đến những phần mà tôi đã đánh dấu bằng viền màu đỏ :

Contents Index Back <> Quick Info Overview Group

GetProcAddress

The **GetProcAddress** function returns the address of the specified exported dynamic-link library (DLL) function.

```
FARPROC GetProcAddress(
    HMODULE hModule,          // handle to DLL module
    LPCSTR lpProcName        // name of function
);
```

Parameters

hModule
Identifies the DLL module that contains the function. The [LoadLibrary](#) or [GetModuleHandle](#) function returns this handle.

lpProcName
Points to a null-terminated string containing the function name, or specifies the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero.

Return Values

If the function succeeds, the return value is the address of the DLL's exported function.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Remarks

The **GetProcAddress** function is used to retrieve addresses of exported functions in DLLs.

The spelling and case of the function name pointed to by *lpProcName* must be identical to that in the **EXPORTS** statement of the source DLL's module-definition (.DEF) file.

The *lpProcName* parameter can identify the DLL function by specifying an ordinal value associated with the function in the **EXPORTS** statement. **GetProcAddress** verifies that the specified ordinal is in the range 1 through the highest ordinal value exported in the .DEF file. The function then uses the ordinal as an index to read the function's address from a function table. If the .DEF file does not number the functions consecutively from 1 to *N* (where *N* is the number of exported functions), an error can occur where **GetProcAddress** returns an invalid, non-NULL address, even though there is no function with the specified ordinal.

In cases where the function may not exist, the function should be specified by name rather than by ordinal value.

Hàm **GetProcAddress** có thể làm được điều này bởi vì các tên và địa chỉ của những exported function được sắp xếp trong một cấu trúc được định nghĩa rất tốt trong **Export Directory**. Chúng ta có thể tìm thấy **Export Directory** bởi vì chúng ta biết nó là thành phần đầu tiên trong data directory và RVA của nó được chứa tại offset 78h từ nơi bắt đầu của PE Header. (Xin xem thêm phần phụ lục)

Cấu trúc export được gọi là **IMAGE_EXPORT_DIRECTORY**. Có 11 thành phần trong cấu trúc này nhưng có một số không quan trọng :

```

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 – Internal name của module. Trường này thực sự cần thiết bởi vì tên của file có thể bị thay đổi bởi người sử dụng . Nếu điều đó xảy ra , trình PE loader sẽ sử dụng Internal name này.

nBase – Bắt đầu của số thứ tự hay số chỉ số (Trường này được sử dụng để lấy những index trong address-of-function array – xem ở bên dưới).

NumberOfFunctions – Tổng số các hàm mà được exported bởi module.

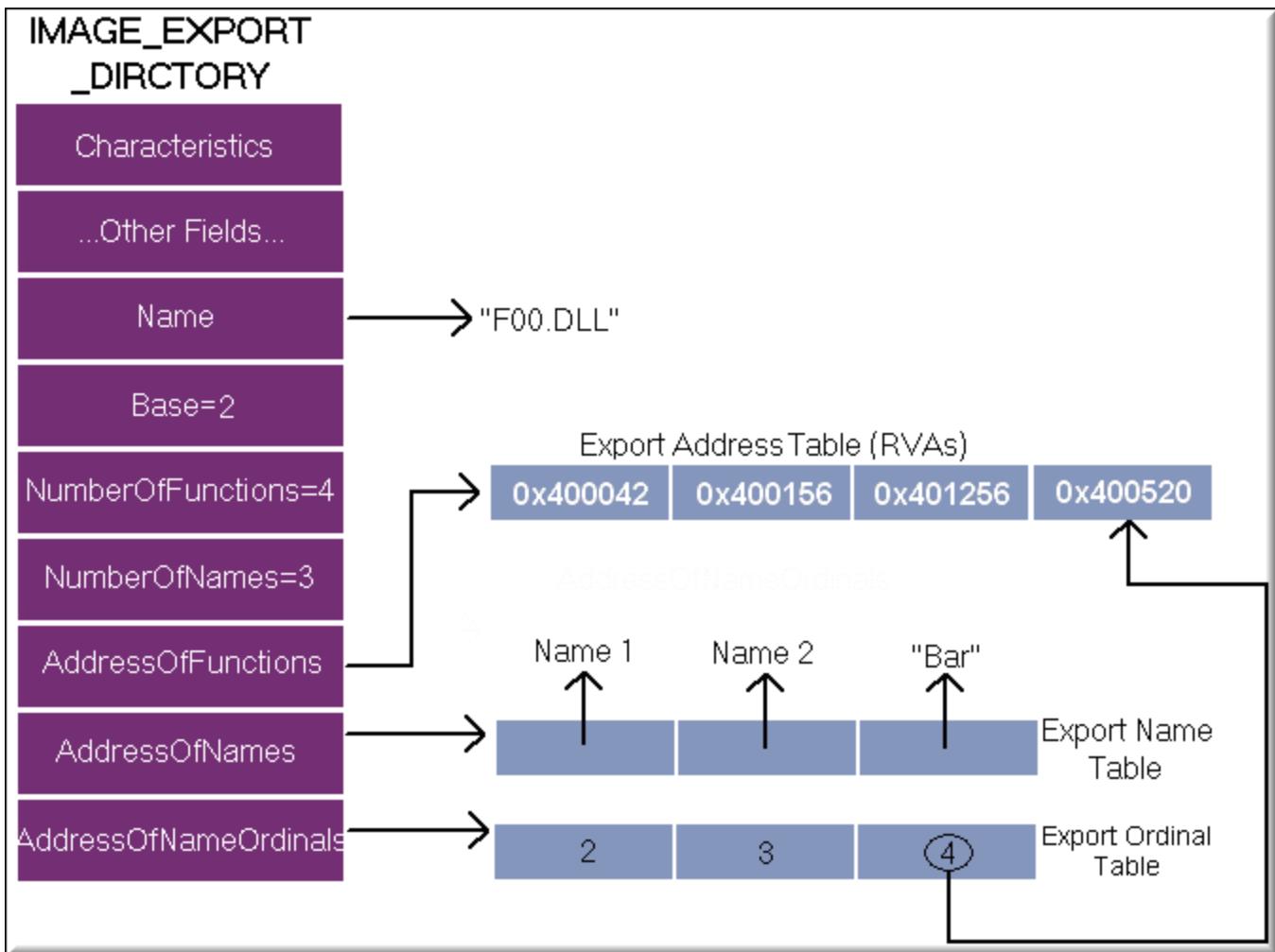
NumberOfNames – Số lượng các Symbols được exported bằng name. Giá trị này không phải là số lượng của tất cả các hàm/symbols trong module. Để lấy được con số này, bạn cần phải kiểm tra

NumberOfFunctions .Nó có thể là 0. Trong trường hợp ấy, module có thể export bằng ordinal only. Nếu không có hàm / symbol được exported trong trường hợp đầu tiên , thì RVA của bảng Export table trong data directory sẽ là 0.

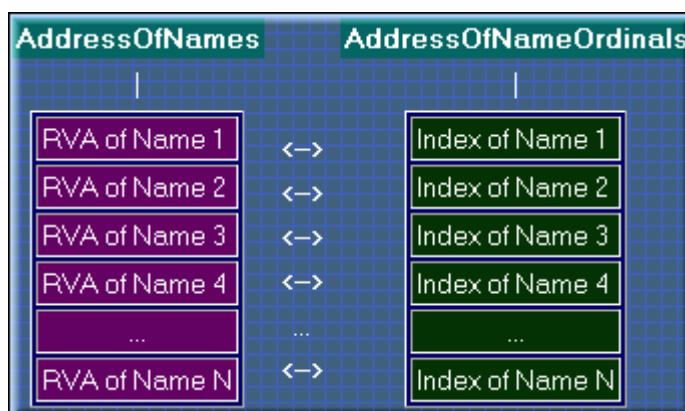
AddressOfFunctions – một RVA trả tới một mảng của các con trả tới các hàm trong module – **Export Address Table (EAT)**. Để sử dụng nó theo cách khác, những RVA trả tới các hàm trong module được giữ lại trong một mảng và trường này trả tới đầu của mảng đó.

AddressOfNames – một RVA trả tới một mảng các RVA của tên các hàm được lưu trong module – **Export Name Table (ENT)**.

AddressOfNameOrdinals – một RVA trả tới một mảng 16 bit mà chứa các ordinals của các named functions – **Export Ordinal Table (EOT)**.



Như vậy cấu trúc **IMAGE_EXPORT_DIRECTORY** trả tới 3 mảng và một bảng những chuỗi kí tự ASCII. **Mảng quan trọng là EAT**, vì nó là một mảng của các con trỏ hàm mà chứa địa chỉ của các exported functions. Hai mảng thứ hai là (ENT và EOT) chạy song song theo thứ tự sắp xếp tăng dần dựa trên tên của các hàm để một phép tìm kiếm nhị phân cho tên của hàm có thể được thực hiện và sẽ đưa kết quả là số thứ tự của hàm đó được tìm thấy vào trong một mảng khác. Số thứ tự chỉ đơn giản là một chỉ số bên trong EAT đối với hàm đó.



Trước đây mảng EOT tồn tại như là một liên kết giữa tên và địa chỉ, nó không thể chứa nhiều phần tử hơn mảng ENT. Ví dụ : mỗi một tên có thể có một và chỉ một địa chỉ tương ứng. Điều ngược lại là không đúng : một địa chỉ có thể có nhiều tên tương ứng với nó. Nếu đó là những hàm với “tên bí danh” tham chiếu đến cùng một địa chỉ thì ENT sẽ có nhiều phần tử hơn là EOT

Lấy ví dụ , nếu một file Dll export 40 hàm , thì nó phải có 40 thành phần trong mảng được trả bởi AddressOfFunctions (EAT) và trường NumberOfFunctions phải chứa 40 giá trị. Để tìm kiếm một hàm từ tên của nó, Hệ điều hành (OS) đầu tiên sẽ tìm những giá trị của **NumberOfFunction** và **NumberOfNames** trong Export Directory. Tiếp theo nó sẽ dạo qua các mảng được trả bởi **AddressOfNames (ENT)** và **AddressOfNameOrdinals (EOT)** một cách đồng thời, để tìm kiếm tên của hàm. Nếu như tên của hàm được tìm thấy trong ENT, thì giá trị tương ứng với phần tử trong EOT được trích xuất và sử dụng như là chỉ mục bên trong EAT.

Lấy ví dụ , trong file Dll 40 hàm của chúng ta ở trên chúng ta muốn tìm kiếm hàm X. Nếu chúng ta tìm tên hàm X(gián tiếp thông qua con trỏ khác) tại phần tử thứ 39 trong ENT , chúng ta nhìn vào phần tử thứ 39 của EOT và thấy 5 giá trị . Sau đó chúng ta xét phần tử thứ 5 của EAT để tìm kiếm RVA của hàm X.

Nếu như bạn đã sẵn có số thứ tự của một hàm , bạn có thể tìm thấy địa chỉ của nó bằng cách đi trực tiếp tới EAT. Mặc dù có được địa chỉ của một hàm thông qua số thứ tự của nó thì dễ dàng hơn và nhanh hơn rất nhiều so với việc sử dụng tên của hàm , thì ngược lại điều bất lợi là sẽ gặp khó khăn trong việc quản lý module. Nếu như file Dll được nâng cấp / cập nhật và số thứ tự của các hàm bị thay đổi, thì các chương trình khác mà chạy dựa trên file Dll này sẽ bị Break.

Exporting by Ordinal Only :

NumberOfFunctions phải ít nhất là bằng với **NumberOfNames**. Tuy nhiên thỉnh thoảng trong một số trường hợp thì **NumberOfNames** lại ít hơn **NumberOfFunctions** . Khi một hàm được Exported thông qua số thứ tự , nó không có danh sách trong cả hai mảng ENT và EOT – nó không có tên. Những hàm mà không có tên thì được Exported thông qua số thứ tự.

Lấy ví dụ như sau , nếu ta có 70 hàm nhưng chỉ có duy nhất 40 mục trong ENT , vậy thì có nghĩa là có 30 hàm trong module mà được Exported bằng số thứ tự. Vậy bây giờ làm thế nào để chúng ta tìm ra những hàm đó là gì? Điều này không dễ dàng. Bạn phải tìm ra bằng phương pháp loại trừ, lấy ví dụ : những mục trong EAT mà không được tham chiếu bởi EOT chứa RVAs của các hàm được Exported bằng số thứ tự.

Người lập trình viên có thể chỉ rõ số thứ tự bắt đầu trong một .def file. Lấy ví dụ , các bảng trong hình minh họa ở trên có thể bắt đầu tại 200. Để mà đối phó trước sự cần thiết cho 200 phần tử rỗng đầu tiên trong mảng , thành phần **nBase** lưu giữ giá trị bắt đầu và trình loader trừ các số thứ tự từ nó để thu được chỉ mục thật trong EAT.

Export Forwarding :

Đôi khi các hàm có vẻ được Exported từ một file Dll đặc thù, nhưng trên thực tế các hàm này lại nằm trong một file Dll hoàn toàn khác. Điều này được gọi là Export Forwarding . Lấy ví dụ , trong hệ điều hành WinNT , Win2k và WinXP, hàm trong **kernel32.dll** là **HeapAlloc** được forwarded từ hàm **RtlAllocHeap** được Exported bởi **ntdll.dll**. File **NTDLL.DLL** cũng chứa các API bẩm sinh mà tương tác trực tiếp với kernel windows . Forwarding được thực hiện tại thời điểm liên kết thông qua một câu lệnh đặc biệt trong **.DEF file**.

Forwarding là một kĩ thuật mà Microsoft sử dụng để đưa ra một tập hợp các API thông dụng và để che dấu sự khác biệt nền tảng giữa họ hệ điều hành NT với họ 9X. Các ứng dụng không có được nhiệm vụ để gọi các hàm trong tập hợp các API bẩm sinh vì điều này sẽ phá vỡ khả năng tương thích giữa Win9x và 2K/XP. Điều này có thể giải thích tại sao các file thực thi bị Packed có thể được unpacked và có bảng imports của chúng được xây dựng lại bằng tay trên một OS có thể không run được trên OS khác bởi hệ thống API forwarding hoặc một vài chi tiết khác đã bị chỉnh sửa.

Khi một Symbol (Hàm) được Forwarded RVA của nó một cách rõ ràng không thể là một đoạn code hoặc địa chỉ dữ liệu trong module hiện tại. Để thay thế , bảng EAT chứa một con trỏ tới một chuỗi ASCII của file DLL và tên hàm mà nó được Forwarded. Trong ví dụ trước nó sẽ là **NTDLL.RtlAllocHeap**.

Nếu vậy thì mục EAT cho một hàm trả tới một địa chỉ bên trong Export Section (ví dụ chuỗi ASCII) thay vì hơn là trả ra ngoài vào một file DLL khác, thì bạn biết rằng hàm đó đã được forwarded.

8. The Import Section :

Import Section (thường được biết dưới tên **.idata**) bao gồm thông tin về tất cả các hàm được imported bởi file thực thi từ các file Dlls. Thông tin này được lưu trữ trong một vài cấu trúc dữ liệu. Phần quan trọng nhất của section này là **ImportDirectory** và **ImportAddressTable** mà chúng ta sẽ nói đến tiếp theo đây. Trong một số file thực thi có thể cũng có các directories là **Bound_Import** và **Delay_Import**. **Delay_Import directory**, với chúng ta nó không quan trọng lắm nhưng chúng ta sẽ đề cập tới **Bound_Import directory** ở phần tiếp sau.

Trình Windows loader chịu trách nhiệm về việc **nạp** tất cả các file Dll mà ứng dụng sử dụng và ánh xạ chúng vào trong không gian địa chỉ process. Nó phải tìm địa chỉ của tất cả các imported functions trong các file Dlls khác nhau của chúng và sắp đặt chúng sẵn sàng để sử dụng cho các file thực thi được **c nạp**.

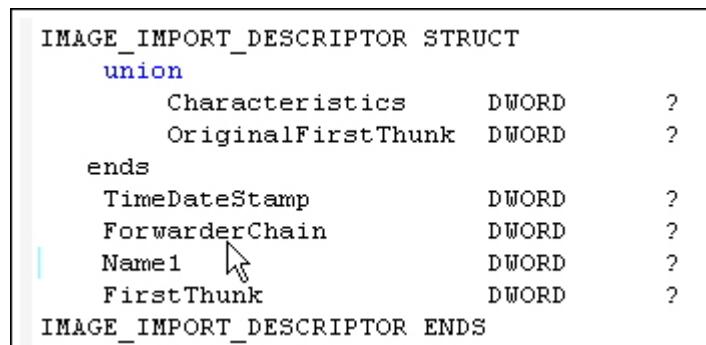
Địa chỉ của các hàm bên trong một file Dll không phải là những địa chỉ tĩnh mà thay đổi khi các phiên bản được cập nhật hóa của file Dll được released, vì vậy các ứng dụng không thể được xây dựng để sử dụng các địa chỉ hàm hardcoded. Bởi vì đó là một cơ chế được phát triển để cho phép những thay đổi mà không cần phải tạo ra nhiều sự thay đổi, chỉnh sửa đối với đoạn mã của file thực thi vào lúc chạy. Điều này đã được hoàn thành thông qua việc sử dụng một **Import Address Table (IAT)**. Đây là một bảng của những con trỏ tới các địa chỉ hàm mà đã được điền vào bởi trình Windows loader khi các file Dll được **nạp**.

Bằng việc sử dụng một bảng con trỏ, trình loader không cần phải thay đổi những địa chỉ của các imported functions trong đoạn mã lệnh mà chúng được gọi. Tất cả những thứ mà nó phải làm là thêm địa chỉ chính xác vào một nơi riêng lẻ trong bảng import và công việc của nó được hoàn tất.

The Import Directory :

Import Directory thực sự là một mảng của các cấu trúc **IMAGE_IMPORT_DESCRIPTOR**. Mỗi cấu trúc là **20 bytes** và chứa thông tin về một DLL mà PE file của chúng ta import các hàm vào. Lấy ví dụ, nếu PE file của chúng ta import các hàm từ 10 file DLL khác nhau, thì sẽ có 10 cấu trúc **IMAGE_IMPORT_DESCRIPTOR** trong mảng này. Không có trường nào chỉ cho ta biết số lượng của các cấu trúc trong mảng này. Để thay thế, cấu trúc cuối cùng sẽ có các trường được điền đầy các giá trị 0 (zeros).

Cùng với **Export Directory**, bạn có thể tìm thấy **Import Directory** ở đâu bằng việc quan sát tại **Data Directory** (80 bytes từ chỗ bắt đầu của PE Header). Trong đó thì thành phần đầu tiên và cuối cùng là quan trọng nhất :



Thành phần đầu tiên **OriginalFirstThunk**, là một DWORD union, có thể tại một thời điểm là một tập hợp của các cờ. Tuy nhiên, Microsoft đã thay đổi ý nghĩa của nó và không bao giờ lo lắng để cập nhật file **WINNT.H**. Trường này thực sự chứa RVA của một mảng các cấu trúc **IMAGE_THUNK_DATA**. [Tiện đây cũng nói luôn, từ union được đề cập ở trên chẳng qua chỉ là một sự định nghĩa lại của cùng một nơi của bộ nhớ. Từ **union** ở trên không chứa 2 DWORDS nhưng chỉ duy nhất một có thể chứa hoặc **OriginalFirstThunk** data hay **Characteristics** data mà thôi]

Thành phần tiếp theo là **TimeDateStamp** được đặt là 0 trừ khi file thực thi được giới hạn khi nó chứa -1 (xem ở bên dưới). Thành phần tiếp là **ForwarderChain** được sử dụng cho việc liên kết old-style và thành phần này sẽ không được đề cập đến ở đây.

Thành phần **Name1** chứa một con trỏ (RVA) tới chuỗi tên ASCII của file DLL.

Thành phần cuối cùng đó là **FirstThunk**, nó cũng chứa RVA của một mảng các cấu trúc **IMAGE_THUNK_DATA** – một bản sao của mảng đầu tiên. Nếu như hàm được miêu tả là một bound import (xem bên dưới) thì **FirstThunk** chứa địa chỉ thực sự của hàm thay vì một RVA tới một **IMAGE_THUNK_DATA**. Những cấu trúc này được định nghĩa như sau :

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

Mỗi **IMAGE_THUNK_DATA** là một DWORD union mà thực tế chỉ có một giá trị. Trong file trên đĩa nó chứa số thứ tự của imported function hoặc là một RVA tới một cấu trúc **IMAGE_IMPORT_BY_NAME**. Một khi đã được nạp một cấu trúc sẽ được trả tại bởi **FirstThunk** được viết đè lên bằng địa chỉ của các hàm imported function.- việc này trở thành **Import Address Table**.

Mỗi cấu trúc **IMAGE_IMPORT_BY_NAME** được định nghĩa như hình minh họa dưới đây :

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

Hint – Chứa chỉ mục(index) bên trong Export Address Table của file DLL các hàm hiện có trong đó. Trường này được sử dụng bởi trình PE Loader vì vậy nó có thể tìm kiếm hàm trong Export Address Table của DLL một cách nhanh chóng. Tên tại đó mà chỉ mục được dùng , và nếu nó không tương ứng thì một phép tìm kiếm nhị phân được thực hiện để tìm kiếm tên. Thông thường giá trị này không cần thiết và một vài trình linker đặt trường này là 0.

Name1 – bao gồm tên của imported function. Tên là một null-terminated ASCII string. Chú ý rằng kích thước của **Name1** được định nghĩa là một byte nhưng trên thực tế nó là một trường có kích thước thay đổi. Do đó không có phương pháp nào để biểu diễn một trường có kích thước thay đổi trong một cấu trúc. Cấu trúc mà được cung cấp để cho bạn có thể tham chiếu tới nó thông qua các tên miêu tả.

Những phần quan trọng nhất là các tên imported DLL và các mảng của các cấu trúc **IMAGE_THUNK_DATA**. Mỗi cấu trúc **IMAGE_THUNK_DATA** tương ứng với một imported function từ DLL. Các mảng được trả tới bởi **OriginalFirstThunk** và **FirstThunk** chạy song song và

được kết thúc bằng một Null DWORD. Đó là cặp phân tách của các mảng của các cấu trúc **IMAGE_THUNK_DATA** cho mỗi imported DLL.

Để sử dụng nó theo một cách khác, có nhiều các cấu trúc **IMAGE_IMPORT_BY_NAME**. Bạn tạo ra hai mảng , sau đó điền vào hai mảng này các RVAs của các cấu trúc **IMAGE_IMPORT_BY_NAME**, vì vậy cả hai mảng này cùng chứa các giá trị giống nhau. Bây giờ bạn có thể gán RVA của mảng đầu tiên cho OriginalFirstThunk và RVA của mảng thứ hai cho FirstThunk.

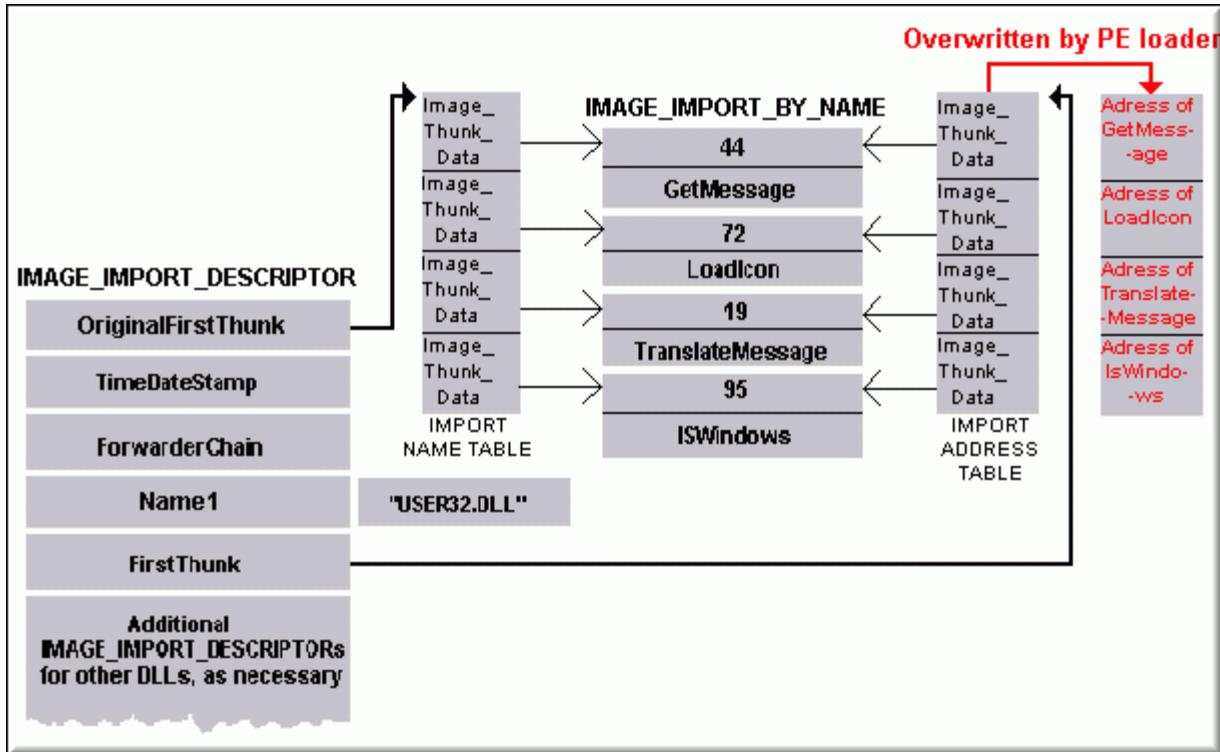
Số lượng các phần tử trong các mảng OriginalFirstThunk và FirstThunk phụ thuộc vào số lượng của các hàm được imported từ file DLL. Lấy ví dụ , nếu PE file import 10 hàm từ file dll là user32.dll, thì thành phần Name1 trong cấu trúc **IMAGE_IMPORT_DESCRIPTOR** sẽ chứa RVA của chuỗi “user32.dll” và sẽ là **10 IMAGE_THUNK_DATA** trong mỗi mảng.

Hai mảng song song , tương đương được gọi bởi các tên khác nhau nhưng cái tên chung nhất là **Import Address Table** (cho một được trả bởi FirstThunk) và **Import Name Table hay Import Lookup Table** (cho một được trả bởi OriginalFirstThunk).

Tại sao lại có hai mảng tương đương của các con trỏ tới những cấu trúc **IMAGE_IMPORT_BY_NAME**? Các **Import Name Table** được để nguyên và không bao giờ được chỉnh sửa. Các **Import Address Table** được viết lại với những địa chỉ hàm thực sự bởi trình loader. Trình loader lặp lại thông qua mỗi con trỏ tới các hàm và tìm kiếm địa chỉ của hàm mà mỗi cấu trúc tham chiếu tới. Trình loader sau đó sẽ viết lại con trỏ tới **IMAGE_IMPORT_BY_NAME** bằng địa chỉ của hàm. Các mảng của những RVAs trong **Import Name Tables** giữ nguyên không bị thay đổi vì vậy nếu cần thiết để tìm tên của các hàm imported , trình PE loader có thể vẫn tìm thấy chúng.

Mặc dù IAT được trả tới bởi entry number 12 trong Data Directory , một vài chương trình linkers không thiết lập danh sách thư mục này và tuy nhiên trình ứng dụng sẽ chạy. Trình loader chỉ sử dụng điều này để đánh dấu một cách tạm thời IAT khi read-write trong lúc import resolution và có thể giải quyết các import mà không cần nó.

Đó là cách thức mà trình Windows loader có thể viết lại IAT khi nó hiện có trong một section chỉ đọc (readonly – section). Tại thời điểm nạp hệ thống thiết lập một cách tạm thời các thuộc tính của các trang chứa dữ liệu import để đọc hoặc ghi. Khi import table được khởi tạo các trang được thiết lập trở lại với các thuộc tính được bảo vệ nguyên bản của chúng.



Các lời gọi tới các hàm được import xảy ra thông qua một con trỏ hàm trong IAT. Lấy ví dụ , hãy tưởng tượng rằng địa chỉ 00405030 tham chiếu tới 1 hàm của danh sách trong mảng FirstThunk mà đã được viết lại bởi trình loader bằng địa chỉ của hàm GetMessage trong file USER32.DLL.

Cách thức hiệu quả để gọi hàm GetMessage giống như dưới đây :

0040100C CALL DWORD PTR [00405030]

Còn cách thức kém hiệu quả là như sau :

0040100C CALL [00402200]

.....

.....

00402200 JMP DWORD PTR [00405030]

Lấy ví dụ , phương pháp thứ hai cũng thu được một kết quả tương tự nhưng sử dụng 5 byte thêm vào của code và mất thời gian lâu hơn để thực thi bởi vì extra jump.

Tại sao các lời gọi tới hàm được imported lại được thực hiện theo cách này? Chương trình biên dịch có thể không phân biệt giữa các lời gọi hàm thông thường trong cùng một module và các hàm được imported cho ra cùng một đầu ra giống nhau : **CALL [XXXXXXXXXX]**

Tại đây thì **XXXXXXXXXX** phải là một địa chỉ code thực sự (không phải là một con trỏ) được điền vào sau đó bởi chương trình linker. Trình linker không biết địa chỉ của hàm được imported và vì vậy phải cung cấp phần thay thế của đoạn mã (code) – The jump stub seen above.

Cách tối ưu được sử dụng là cách sử dụng trình **the _declspec(dllexport) modifier** để thông báo cho chương trình biên dịch rằng hàm hiện có bên trong một file DLL. Nó sẽ có kết quả là **CALL DWORD PTR [XXXXXXXXXX]**.

Nếu như **_declspec(dllexport)** không được sử dụng khi biên dịch một file thực thi thì sẽ có một tập hợp lớn của các jump stubs cho các hàm được imported để xác định lẫn nhau nằm ở đâu trong đoạn mã lệnh. Điều này được biết bởi các tên khác nhau ví dụ như "transfer area", "trampoline" or "jump thunk table".

Functions Exported by Ordinal Only:

Như chúng ta đã thảo luận trong phần về Export section, thì một số hàm được exported thông qua số thứ tự. Trong trường hợp này, sẽ không có cấu trúc IMAGE_IMPORT_BY_NAME cho hàm đó trong module của lời gọi (caller's module). Thay vào đó, IMAGE_THUNK_DATA cho hàm đó chứa số thứ tự của hàm.

Trước khi file thực thi được nạp, bạn có thể biết nếu một cấu trúc IMAGE_THUNK_DATA chứa một số thứ tự hoặc một RVA bằng cách xem xét bit có ý nghĩa quan trọng nhất (MSB) hay bit cao. Nếu được thiết lập thì 31 bits thấp hơn được xem như là một giá trị số thứ tự. Nếu không được set, thì giá trị là một RVA tới một IMAGE_IMPORT_BY_NAME. Microsoft cung cấp một hàng số có ích cho việc kiểm tra bit MSB của một DWORD, đó là **IMAGE_ORDINAL_FLAG32**. Nó có giá trị là 80000000h.

Lấy ví dụ, nếu một hàm được exported thông qua số thứ tự và số thứ tự của nó là 1234h, thì IMAGE_THUNK_DATA cho hàm đó sẽ là 80001234h.

Bound Imports :

Khi trình Loader nạp một PE file vào trong bộ nhớ, nó kiểm tra bảng import table và nạp các file DLLs được yêu cầu vào không gian địa chỉ xử lý. Sau đó nó dạo qua mảng được trả bởi **FirstThunk** và thay thế **IMAGE_THUNK_DATA** bằng những địa chỉ thực sự của các import functions. Giai đoạn này tốn khá nhiều thời gian. Nếu vì một lý do chưa biết người lập trình có thể dự đoán địa chỉ của các hàm một cách chính xác, trình PE loader không phải sửa các **IMAGE_THUNK_DATA** mỗi lần PE file thực thi y như địa chỉ chính xác là đã có rồi. Sự liên kết là kết quả của ý tưởng này.

Có một tiện ích được đặt tên là **bind.exe** đi kèm với các trình biên dịch của Microsoft, kiểm tra IAT (mảng FirstThunk) của một PE file và thay thế các **IMAGE_THUNK_DATA** Dword bằng địa chỉ của các import functions. Khi file được nạp, trình PE loader phải kiểm tra các địa chỉ đó có hợp lệ không. Nếu phiên bản của file DLL không khớp với một file trong PE file hoặc nếu các file DLLs cần phải được xây dựng lại, trình PE loader biết rằng các địa chỉ được liên kết là hết hiệu lực và nó dạo qua bảng Import Name Table (Original FirstThunk array) để tính toán các địa chỉ mới.

Bởi vậy mặc dù INT là không cần thiết cho một file thực thi để nạp, nếu nó không hiện diện file thực thi không thể được liên kết. Trong một thời gian dài trình linker của Borland là TLINK không tạo một INT vì vậy các file được tạo bởi Borland không thể được liên kết. Chúng ta sẽ xem xét tầm quan trọng khác của việc thiếu INT trong các section tiếp theo.

The Bound Import Directory

Thông tin trình loader sử dụng để xác định nếu địa chỉ được liên kết là hợp lệ được lưu giữ trong một cấu trúc là **IMAGE_BOUND_IMPORT_DESCRIPTOR**. Một bound executable chứa một danh sách các cấu trúc, một cho mỗi DLL được imported mà được liên kết :

IMAGE_BOUND_IMPORT_DESCRIPTOR STRUCT		
TimeStamp	DWORD	?
OffsetModuleName	WORD	?
NumberOfModuleForwarderRefs	WORD	?
IMAGE_BOUND_IMPORT_DESCRIPTOR ENDS		

Thành phần **TimeStamp** phải khớp với TimeStamp của exporting DLL's header. Nếu như không khớp, trình loader thừa nhận rằng binary được liên kết tới là “wrong” DLL và sẽ vá lại danh sách import. Điều này có thể xảy ra nếu phiên bản của exporting DLL không khớp hoặc nếu nó cần phải được sắp xếp lại trong bộ nhớ.

Thành phần **OffsetModuleName** chứa offset (không phải là RVA) từ **IMAGE_BOUND_IMPORT_DESCRIPTOR** đầu tiên cho tới tên của DLL trong null-terminated ASCII.

Thành phần **NumberOfModuleForwarderRefs** chứa số lượng các cấu trúc **IMAGE_BOUND_FORWARDER_REF** mà trực tiếp theo cấu trúc này. Cấu trúc này được định nghĩa như sau :

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

Như bạn có thể nhìn thấy chúng giống y hệt như cấu trúc bên trên ngoại trừ thành phần cuối cùng được để dành riêng trong bất kì tình huống nào. Lý do có hai cấu trúc tương tự nhau là khi để liên kết ngược lại một hàm mà được forwarded tới một file DLL khác, tính chất hợp lệ của forwarded DLL đó phải được kiểm tra cũng tại thời gian nạp. **IMAGE_BOUND_FORWARDER_REF** chứa thông tin chi tiết về các forwarded DLLs.

Lấy ví dụ như hàm HeapAlloc trong kernel32.dll được forwarded từ hàm **RtlAllocateHeap** trong file nt.dll. Nếu chúng ta tạo ra một ứng dụng mà import hàm HeapAlloc và được sử dụng bind.exe trong ứng dụng, đó sẽ là một **IMAGE_BOUND_IMPORT_DESCRIPTOR** cho kernel32.dll được theo bởi một **IMAGE_BOUND_FORWARDER_REF** cho nt.dll.

Chú ý : Tên của các hàm bản thân chúng không được bao gồm trong những cấu trúc này khi trình loader biết những hàm nào được liên kết từ **IMAGE_IMPORT_DESCRIPTOR** (xem ở trên).

9. The Windows Loader :

Phần viết này tuy là không cần thiết nhưng nó dành cho những ai muốn đi sâu nghiên cứu thêm về sự hoạt động của hệ điều hành (OS).

What The Loader Does

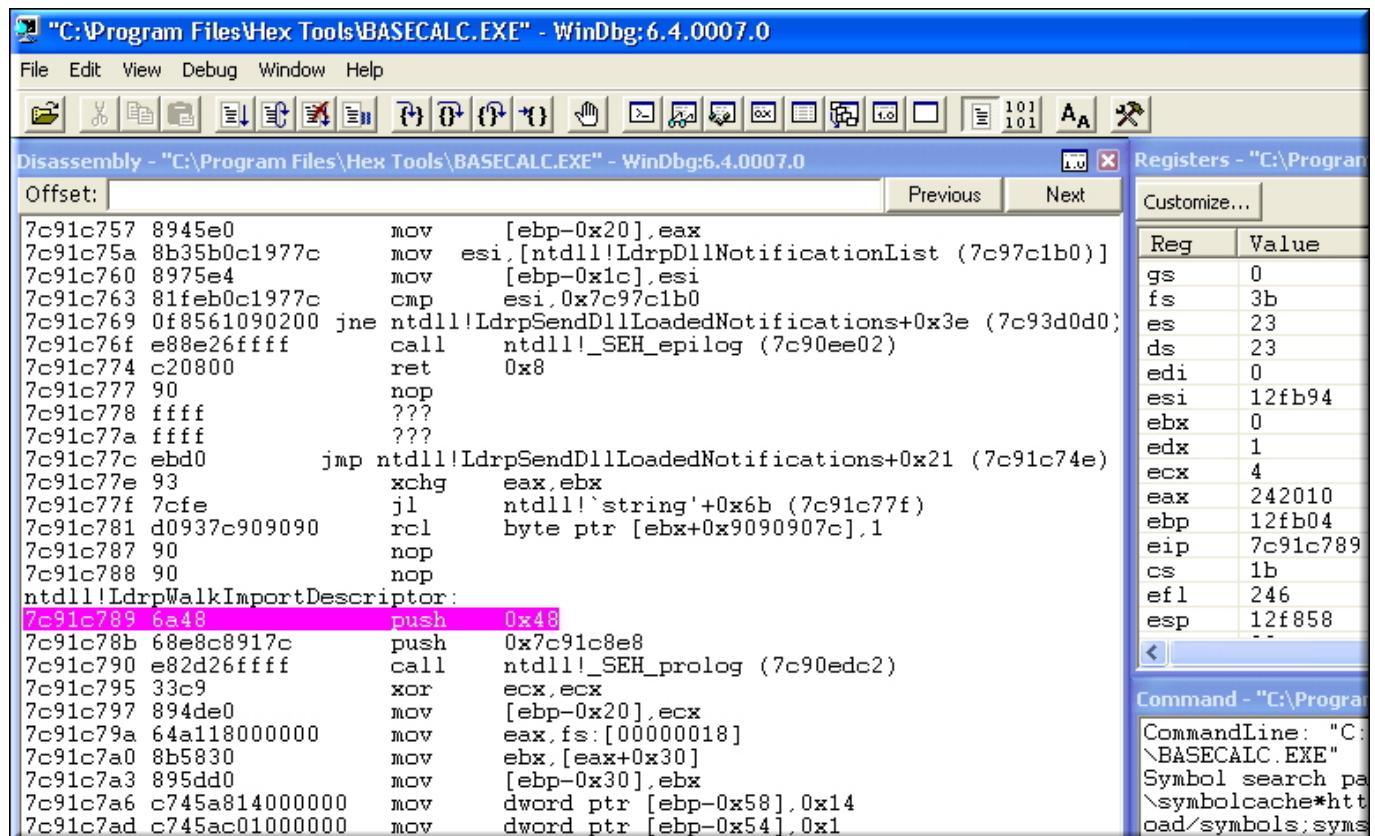
Khi một file thực thi chạy, trình windows loader sẽ tạo ra một không gian địa chỉ ảo cho process và ánh xạ executable module từ đĩa vào trong không gian địa chỉ của process. Nó cố gắng nạp image tại địa chỉ cơ sở được ưu tiên và ánh xạ các section vào trong bộ nhớ (memory). Trình loader sẽ xem xét tệp mảnh section table và ánh xạ mỗi section tại địa chỉ được tính toán bằng cách cộng thêm RVA của section với địa chỉ cơ sở. Các page attributes được thiết lập theo sự yêu cầu đặc điểm của section. Sau khi ánh xạ các section vào trong bộ nhớ, trình loader thực hiện bố trí các relocation nếu địa chỉ nạp không bằng với địa chỉ cơ sở được ưu tiên trong ImageBase.

Bảng import table sau đó được kiểm tra và bắt kịp file DLLs nào được yêu cầu sẽ được ánh xạ vào trong không gian địa chỉ của process. Sau đó tất cả DLL modules được định vị và ánh xạ vào, trình loader kiểm tra mỗi DLL's export section và sau đó IAT được chỉnh sửa để trả tới địa chỉ hàm được imported thực sự.

Nếu như symbol không tồn tại (đây là trường hợp rất hiếm gặp), trình loader sẽ thông báo lỗi. Một khi tất cả các module được yêu cầu đã được nạp sự thi hành được chuyển tới entry point của ứng dụng.

Phần quan trọng được ưa thích trong RCE đó chính là việc loading các file DLLs và giải quyết các imports. Process này bị làm phức tạp bởi rất nhiều các hàm internal (forwarded) và các routines tập trung trong file ntdll.dll mà không hề được chứng minh bằng tại liệu bởi Micro\$oft. Như chúng ta đã nói ở phần trước function forwarding là 1 cách cho M\$ để expose một tập Win32 API thông dụng, phổ biến và che dấu các hàm cấp thấp mà có thể khác nhau đối với từng phiên bản của hệ điều hành. Nhiều hàm kernel32 quen thuộc ví dụ như hàm **GetProcAddress** đơn giản chỉ bao bọc xung quanh các ntdll.dll exports ví dụ như LdrGetProcAddress (mà hàm này thực hiện công việc chính).

Để có thể thấy rõ những điều này bạn cần cài đặt chương trình **Windbg** và **Windows Symbol Package** (được cung cấp bởi M\$) hoặc một chương trình kernel-mode debugger giống như SoftIce. Bạn chỉ có thể xem những hàm này trong Olly nếu như bạn cấu hình Olly để sử dụng M\$ symbolserver, nếu không thì tất cả những gì bạn quan sát thấy chỉ là các pointers và các địa chỉ bộ nhớ mà không có tên của các hàm. Tuy nhiên Olly là một trình debugger trên user-mode và nó sẽ chỉ cho các bạn thấy được những gì đang xảy ra khi ứng dụng của bạn được nạp và nó sẽ không cho phép bạn quan sát thấy loading process. Mặc dù chức năng của chương trình Windbg còn hạn chế không thể so sánh với Olly nhưng nó tương thích tốt với hệ điều hành và sẽ cho ta thấy được quá trình loading process :



Như các bạn thấy có rất nhiều hàm APIs được liên kết cùng với quá trình nạp một file thực thi, tất cả tập trung trên hàm **LoadLibraryExW** trong kernel32.dll mà lần lượt dẫn đến hàm nội tại **LdrpLoadDll** trong ntdll.dll. Hàm này trực tiếp gọi 6 subroutines nữa là **LdrpCheckForLoadedDll**, **LdrpMapDll**, **LdrpWalkImportDescriptor**, **LdrpUpdateLoadCount**, **LdrpRunInitializeRoutines**, và **LdrpClearLoadInProgress** thực hiện những nhiệm vụ sau :

1. Kiểm tra để xem nếu module đã sẵn sàng để nạp vào.
2. Ánh xạ module và các thông tin hỗ trợ vào trong bộ nhớ.
3. Dạo qua bảng import descriptor table của module (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. Khởi tạo module

6. Xóa 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 (0x77f8beb9)
        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)
```

Một DLL có thể import các module khác mà bắt đầu một tầng của thư viện thêm vào. Trình loader sẽ cần phải lặp lại từ đầu đến cuối mỗi module , kiểm tra để xem nếu nó cần được nạp và sau đó kiểm tra những phụ thuộc của nó. Đó là lý do có sự xuất hiện của **LdrpWalkImportDescriptor** ở đây.

LdrpWalkImportDescriptor có hai subroutines đó là : **LdrpLoadImportModule** và **LdrpSnapIAT**.

Đầu tiên nó bắt đầu bằng hai lời gọi tới **RtlImageDirectoryEntryToData** để xác định vị trí Bound Imports Descriptor và các bảng Import Descriptor. Chú ý rằng trình loader sẽ kiểm tra bound imports đầu tiên- một ứng dụng khi thực thi nhưng không có một import directory có thể có các bound imports để thay thế.

Tiếp theo **LdrpLoadImportModule** xây dựng một Unicode string cho mỗi DLL được tìm thấy trong Import Directory và sau đó giao cho **LdrpCheckForLoadedDll** để nhận ra if they have already been loaded.

Tiếp nữa **LdrpSnapIAT** routine kiểm tra mỗi DLL được tham chiếu tới trong Import Directory thay thế cho 1 giá trị -1 (ie again checks for bound imports first). Sau đó nó thay đổi memory protection của IAT thành PAGE_READWRITE và tiến hành kiểm tra mỗi entry trong IAT trước khi chuyển tới **LdrpSnapThunk** subroutine.

LdrpSnapThunk sử dụng một chỉ số của hàm để xác định địa chỉ của nó và quyết định nó có được forward hay là không. Mặt khác nó gọi **LdrpNameToOrdinal** sử dụng một phép tìm kiếm nhị phân trên export table để xác định chỉ số một cách nhanh chóng. Nếu hàm không được tìm thấy thì nó trả về STATUS_ENTRYPOINT_NOT_FOUND, ngược lại nếu tìm thấy thì nó thay thế entry trong IAT bằng entry point của API và trả về cho **LdrpSnapIAT** khôi phục lại memory protection nó đã thay đổi tại lúc bắt đầu công việc của nó, gọi **NtFlushInstructionCache** để bắt buộc một cache refresh trên memory block có chứa IAT, và sau đó trả về lại cho **LdrpWalkImportDescriptor**.

Đó là một khác biệt đặc biệt giữa các hệ điều hành Window mà trong đó Win2k nhấn mạnh rằng ntdll.dll được nạp giống như một bound import hoặc trong import directory bình thường trước khi cho phép một

file thực thi được nạp, nhưng ngược lại hệ điều hành Win9x hay XP sẽ cho phép một ứng dụng không có imports nào được nạp

Phản khái quát ngắn gọn này đã được đơn giản hóa đi rất nhiều nhưng vẫn minh họa được làm thế nào một lời gọi tới LoadLibrary làm tăng lên một tầng của việc ẩn các subroutines nội tại which are deeply nested and recursive in places. Trình loader phải kiểm tra mỗi API được imported để tính toán một địa chỉ thực trong bộ nhớ và để kiểm tra nếu một API đã được imported. Mỗi DLL được imported có thể dẫn đến các modules thêm vào và process sẽ bị lặp lại hết lần này đến lần khác cho tới khi tất cả các phụ thuộc đều đã được kiểm tra.

10. Navigating Imports :

Navigating Imports on Disk

Nếu như các bạn muốn tìm kiếm thông tin về các hàm được imported từ file DLL ("foo" from DLL "bar"), đầu tiên các bạn tìm RVA của **Import Directory** từ **Data Directory**, tìm địa chỉ đó trong phần raw section data và bây giờ bạn có một mảng của các **IMAGE_IMPORT_DESCRIPTORs**. Lấy thành viên của mảng này mà liên quan tới bar.dll bằng cách kiểm tra các strings được trả tới bởi trường 'Name'. Khi bạn đã tìm thấy **IMAGE_IMPORT_DESCRIPTOR** đúng, follow 'FirstThunk' của nó và nám lấy con trả tới mảng các mảng **IMAGE_THUNK_DATA**s, kiểm tra kĩ các RVAs và tìm kiếm the function "foo".

Quay trở lại ví dụ của chúng ta trong chương trình Hexeditor, chúng ta sẽ tìm vị trí của bảng import table để quan sát những gì chúng ta cần tìm kiếm. Như chúng ta đã nói ở phần trước, RVA của **Import Directory** được lưu trong DWORD 80h bytes từ PE Header mà trong ví dụ của chúng ta là offset 180h và RVA là 2D000h (xem lại phần **Data Directory**). Bây giờ chúng ta phải chuyển đổi RVA đó sang một raw offset để nghiên cứu kĩ phạm vi chính xác của file của chúng ta trên đĩa. Kiểm tra **Section Table** để xem xét section nào mà địa chỉ của **Import Directory** nằm trong .Trong trường hợp của chúng ta, thì **Import Directory** bắt đầu tại nơi bắt đầu của .idata section và chúng ta biết rằng section table lưu giữ các raw offset trong **PointerToRawData** DWORD. Trong ví dụ của chúng ta thì offset là 2AC00h (xem phần section table). Bất kì một trình PE Editor nào cũng cho chúng ta kết quả như bên dưới đây. Ví dụ ta dùng LordPE, ta có như sau :

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

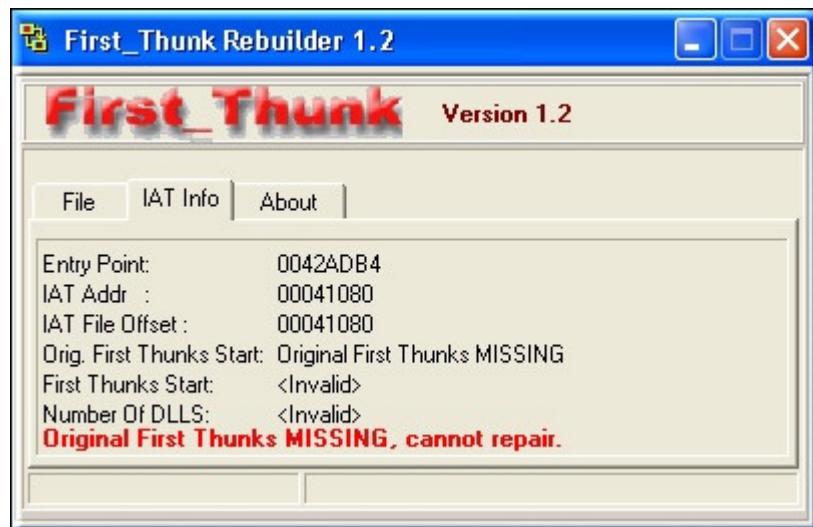
Sự khác biệt giữa RVA và Raw offset là 2D000h – 2AC00h = 2400h. Hãy chú ý tới điều này bởi vì nó sẽ có ích cho việc chuyển đổi các offsets. Xem thêm phần phụ lục để có thêm các thông tin về việc chuyển đổi các RVAs.

Tại Offset 2AC00h chúng ta có **Import Directory** – một mảng của các **IMAGE_IMPORT_DESCRIPTORs** mỗi mảng là 20 bytes và lặp lại cho mỗi import library (DLL) cho tới khi được kết thúc bởi 20 bytes có giá trị 00h. Trong chương trình HexEditor chúng ta quan sát thấy được như sau tại 2AC00h :

0002ac00h:	00 00 00 00 00 00 00 00 00 00 00 00 30 D5 02 00	;00..
0002ac10h:	B4 D0 02 00 00 00 00 00 00 00 00 00 00 00 00 00	; 'Đ.....
0002ac20h:	06 D7 02 00 24 D1 02 00 00 00 00 00 00 00 00 00 00	; .x..\$Ñ.....
0002ac30h:	00 00 00 00 20 D7 02 00 2C D1 02 00 00 00 00 00 00 00	;*.,Ñ....
0002ac40h:	00 00 00 00 00 00 00 00 8A D7 02 00 44 D1 02 00	;Š*x..DÑ..
0002ac50h:	00 00 00 00 00 00 00 00 00 00 00 00 FC D7 02 00	;üx..
0002ac60h:	60 D1 02 00 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 00 00	; NÚ..§Ñ.....
0002ac80h:	00 00 00 00 30 DE 02 00 D4 D2 02 00 00 00 00 00 00 00	;OP..ÔÔ.....
0002ac90h:	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 00	;
0002acb0h:	00 00 00 00 3E D5 02 00 56 D5 02 00 6E D5 02 00	;>Ö..VÖ..nÖ..
0002acc0h:	86 D5 02 00 A2 D5 02 00 B0 D5 02 00 C0 D5 02 00	; tõ..cõ..°Ö..ÀÖ..
0002acd0h:	CC D5 02 00 DA D5 02 00 F0 D5 02 00 FE D5 02 00	; iõ..úõ..gõ..þõ..

Mỗi một nhóm 5 DWORDs biểu diễn 1 **IMAGE_IMPORT_DESCRIPTOR**. Nhóm đầu tiên chỉ cho ta thấy rằng trong file PE này các thành phần **OriginalFirstThunk**, **TimeDateStamp** và **ForwarderChain** được thiết lập là 0. Cuối cùng là chúng ta đi đến một tập hợp của tất 5 DWORDs được thiết lập là 0.(trên hình được tô bằng màu đỏ) mà chỉ cho chúng ta biết đây là kết thúc của mảng.Chúng ta có thể thấy chúng ta đang import các hàm từ 8 DLLs

Chú ý quan trọng : Các trường **OriginalFirstThunk fields** trong ví dụ của chúng ta tất cả đều được set là 0. Đó là điển hình chung cho các file thực thi được tạo ra bằng trình compiler & linker của Borland và là điều đáng để ghi nhớ trong lí do sắp đề cập sau đây. Trong một file thực thi bị Packed thì các con trỏ **FirstThunk pointers** sẽ bị làm mất hiệu lực nhưng có thể thỉnh thoảng được xây dựng lại bằng cách sao chép lại bản sao OriginalFirstThunks(which many simple packers do not seem to bother removing). Đó thực sự là một điều có ích được gọi là **First_ Thunk Rebuilder** by Lunar_Dust mà sẽ thực hiện điều này. Tuy nhiên, với Borland khi đã tạo file thì điều này là không thể bởi vì **OriginalFirstThunks** tất cả đều là **Zero** và không có INT :



Lại quay trở lại ví dụ của chúng ta ở trên, trường **Name1** field của **IMAGE_IMPORT_DESCRIPTOR** đầu tiên chứa RVA 00 02 D5 30h (NB reverse byte order). Chuyển đổi giá trị này sang một raw offset bằng cách trừ đi giá trị 2400h (như đã nói ở trên) và chúng ta có là 2B130h. Nếu chúng ta quan sát trong PE file của chúng ta chúng ta sẽ thấy tên của DLL :

0002b110h:	86 E7 02 00 9C E7 02 00 B0 E7 02 00 C6 E7 02 00	; tç..æç..°ç..Æç..
0002b120h:	DE E7 02 00 F6 E7 02 00 0A E8 02 00 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	; kerne132.dll...
0002b140h:	44 65 6C 65 74 65 43 72 69 74 69 63 61 6C 53 65	; DeleteCriticalSection
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....

Tiếp tục , trường **FirstThunk** field chứa RVA 00 02 D0 B4h mà sau khi convert chúng sẽ có được Raw offset là 2ACB4h. Hãy ghi nhớ điều này đây là offset tới mảng của các cấu trúc DWORD-sized **IMAGE_THUNK_DATA** structures – IAT. Điều này sẽ khiến cho bit có ý nghĩa quan trọng nhất của nó được set (it will start with 8) và phần thấp hơn sẽ chứa số thứ tự của hàm được imported, hoặc nếu MSB không được set nó sẽ chứa RVA khác tới tên của hàm (**IMAGE_IMPORT_BY_NAME**).

Trong file của chúng ta , giá trị DWORD tại 2ACB4h là 00 02 D5 3E:

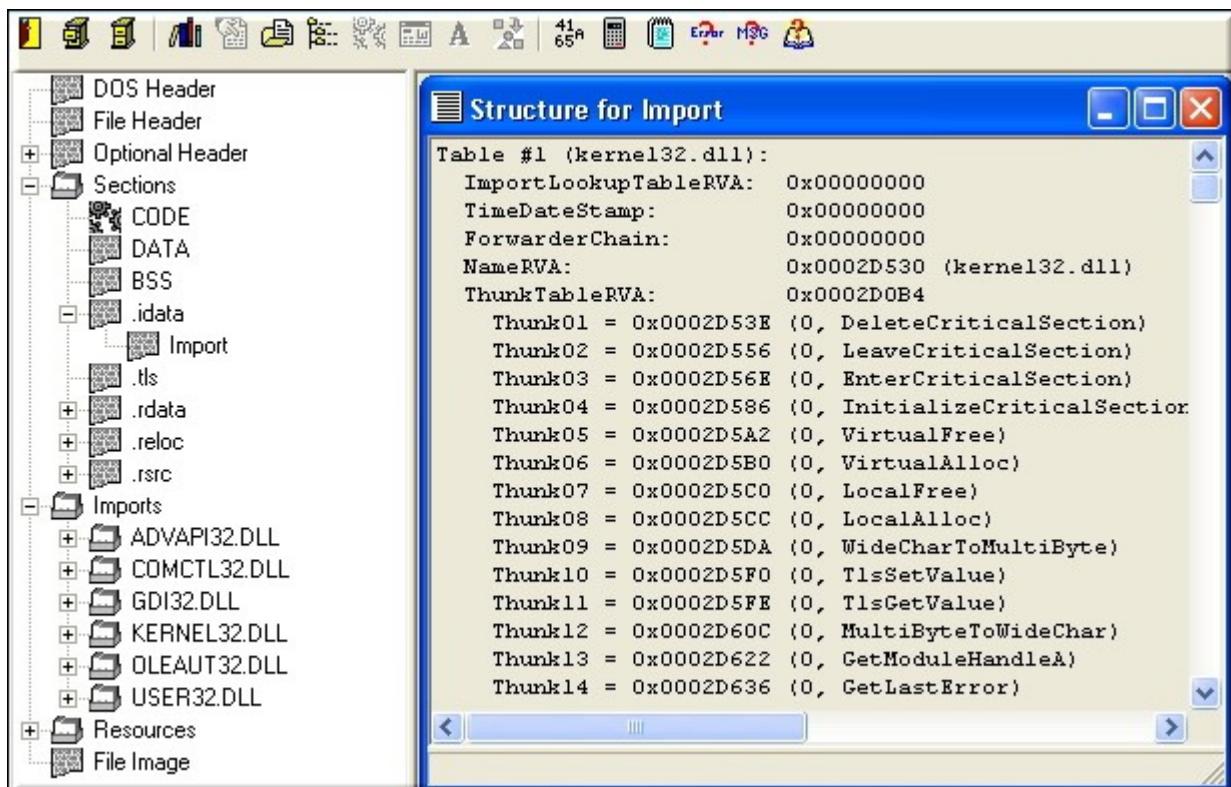
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0002ac90h:	00	00	00	00	00	00	00	F6	E6	02	00	FC	D4	02	00	;
0002aca0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00öæ..üô..
0002acb0h:	00	00	00	00	3E	D5	02	00	56	D5	02	00	6E	D5	02	00
0002acc0h:	86	D5	02	00	A2	D5	02	00	B0	D5	02	00	CO	D5	02	00
0002acd0h:	CC	D5	02	00	DA	D5	02	00	F0	D5	02	00	FE	D5	02	00

Đây là một RVA khác mà khi convert sang RAW offset là 2B13E. Thời điểm này nó sẽ là một null-terminated ASCII string. Như chúng ta quan sát thấy dưới đây :

	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	;
0002b130h:	6B	65	72	6E	65	6C	33	32	2E	64	6C	6C	00	00	00	kerne132.dll...
0002b140h:	44	65	6C	65	74	65	43	72	69	74	69	63	61	6C	53	65
0002b150h:	63	74	69	6F	6E	00	00	00	4C	65	61	76	65	43	72	69
0002b160h:	74	69	63	61	6C	53	65	63	74	69	6F	6E	00	00	00	ticalSection....

Vì vậy tên của của API đầu tiên được imported từ kernel32.dll là **DeleteCriticalSection**. Có thể bạn để ý đến 2 zero bytes trước tên của hàm. Đó là phần tử **Hint** element mà thường được set là 00 00.

Tất cả những điều này có thể được xác minh lại thông qua chương trình PE Browse Pro để phân tích IAT như hình minh họa dưới đây :

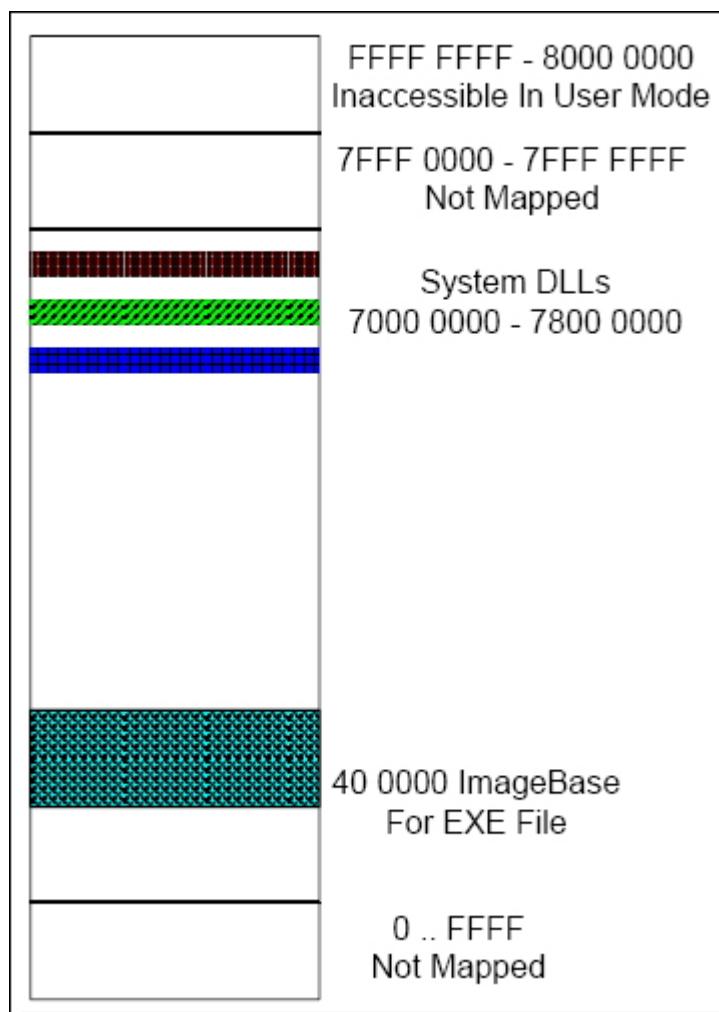


Nếu như file được loaded vào bộ nhớ, được dumped và kiểm tra bằng chương trình Hex editor thì giá trị DWORD tại RVA 2D0B4h mà contained 3E D5 02 00 trên đĩa sẽ được overwritten bởi trình loader bằng địa chỉ của hàm **DeleteCriticalSection** trong kernel32.dll :

	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
0002d0c0h:	A1	9F	80	7C	14	9B	80	7C	81	9A	80	7C	5D	99	80	7C
0002d0d0h:	BD	99	80	7C	C7	A0	80	7C	F5	9B	80	7C	50	97	80	7C
0002d0e0h:	AD	9C	80	7C	29	B5	80	7C	31	03	91	7C	8D	2C	81	7C

Allowing for reverse byte order this is 7C91188A.

Chú ý quan trọng : các hàm trong các DLLs hệ thống luôn luôn hướng về bắt đầu tại địa chỉ 7XXXXXXX và ở cùng tại chỗ giống nhau mỗi khi các chương trình được nạp. Tuy nhiên chúng hay thay đổi nếu bạn cài đặt lại OS của bạn và khác nhau giữa máy tính này và máy tính khác :



Các địa chỉ cũng khác nhau tùy theo từng hệ điều hành, lấy ví dụ :

OS	Base of kernel32.dll
Win XP SP1	77E60000H
Win XP SP2	7C000000H
Win 2000 SP4	79430000H

Trình Windows Updater cũng thỉnh thoảng thay đổi vị trí cơ sở của các DLLs hệ thống. Đó là lí do tại sao một số người thường chú ý đến việc dành thời gian để tìm cho được điểm đặt breakpoint nổi tiếng là **point-h** trên hệ thống của mình (it is prone to change unexpectedly since it is in a function inside user32.dll.)

Navigating Imports in Memory

Load file của chúng ta vào trong Olly và một lần nữa hãy quan sát cửa sổ **Memory Map** :

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

Chú ý rằng địa chỉ của .idata section là 42D000 tương ứng với RVA 2D000 mà chúng ta đã nói ở phần trước. Kích thước được làm tròn lên là 2000 để vừa khít với memory page boundaries.

Cửa sổ chính của Olly là CPU sẽ chỉ cho chúng ta thấy những địa chỉ CODE section (from 401000 to 42AFFF). Bạn cũng có thể kiểm tra IAT trong cửa sổ disassembly nếu nó nằm trong CODE section. Trong hầu hết các trường hợp nó sẽ nằm trong section riêng của nó . eg : .idata nhưng bạn có thể xem nó trong cửa sổ Hex-dump trong Olly bằng cách Right click vào và chọn Dump in CPU. Cửa sổ name (nhấn Ctrl + N) sẽ cho chúng ta thấy được các hàm được imported:

Names in BASECALC			
Address	Section	Type	Name
0042D04C0	.idata	Import	user32.DefMDIChildProcA
0042D04BC	.idata	Import	user32.DefWindowProcA
0042D00B4	.idata	Import	kernel32.DeleteCriticalSection
0042D029C	.idata	Import	gdi32.DeleteDC
0042D0298	.idata	Import	gdi32.DeleteEnhMetaFile
0042D04B8	.idata	Import	user32.DeleteMenu
0042D0294	.idata	Import	gdi32.DeleteObject
0042D04D4	.idata	Import	user32.DestroyCursor

Rightclicking bất kỳ một hàm nào và sau đó chọn Find References to Import sẽ cho bạn thấy jump thunk stub và the instances in the code nơi mà hàm đó được gọi (chỉ có 1 trong trường hợp của chúng ta):

References in BASECALC:CODE to kernel32.DeleteCriticalSection		
Address	Disassembly	Comment
00401314	JMP DWORD PTR DS:[<&kernel32.DeleteCriticalSection>]	ntdll.RtlDeleteCriticalSection
00401B12	CALL <JMP.&kernel32.DeleteCriticalSection>	

Chú ý : trong cột Comment bạn sẽ thấy rằng Olly đã xác định là hàm **DeleteCriticalSection** trong kernel32.dll là thực sự được forwarded tới **RtlDeleteCriticalSection** trong ntdll.dll. (xem phần giải thích Export Forwarding)

Tiếp tục Rightclicking và chọn Follow Import in Disassembler, Olly sẽ cho chúng ta thấy địa chỉ trong DLL thích hợp nơi mà code của hàm bắt đầu . Ví dụ : bắt đầu tại 7C91188A trong ntdll.DLL:

OllyDbg - BASECALC.EXE - [CPU - main thread, module ntdll]

```

C File View Debug Plugins Options Window Help
[<] [X] [>] [II] [F1] [F2] [F3] [F4] [F5] [F6] [F7] [F8] [F9] [F10] [F11] [F12] [L] [E] [M] [T] [W] [H] [C / K] [B] [R ... S] [Grid] [?]
7C91188A $ 6A 1C PUSH 1C
7C91188C . 68 0819917C PUSH ntdll.7C911908
7C911891 . E8 2CD5FFFF CALL ntdll.7C98EDC2
7C911896 . BB5D 08 MOV EBX,DWORD PTR SS:[EBP+8]
7C911899 . BB43 10 MOV EAX,DWORD PTR DS:[EBX+10]
7C91189C . 85C0 TEST EAX,EAX
7C91189E . 0F85 887E0000 JNZ ntdll.7C91972C
7C9118A4 > 8365 E4 00 AND DWORD PTR SS:[EBP-1C],0

```

Nếu chúng ta quan sát tại lời gọi tới hàm **DeleteCriticalSection** tại 00401B12 chúng ta sẽ thấy như sau:

OllyDbg - BASECALC.EXE - [CPU - main thread, module BASECALC]

```

C File View Debug Plugins Options Window Help
[<] [X] [>] [II] [F1] [F2] [F3] [F4] [F5] [F6] [F7] [F8] [F9] [F10] [F11] [F12] [L] [E] [M] [T] [W] [H] [C / K] [B] [R ... S] [Grid] [?]
00401AF5 : 68 1FB4000 PUSH BASECALC.00401B1F
00401AF8 > 803D 36C04200 CMP BYTE PTR DS:[42C0361],0
00401B01 .^ 74 0A JE SHORT BASECALC.00401B00
00401B03 : 68 20C44200 PUSH BASECALC.0042C420
00401B08 . E8 FFF7FFFF CALL <JMP.&kernel32.LeaveCriticalSection>
00401B0D > 68 20C44200 PUSH BASECALC.0042C420
00401B12 . E8 FDF7FFFF CALL <JMP.&kernel32.DeleteCriticalSection>
00401B17 : C3 RETN
00401B18 .^ E9 A3130000 JMP BASECALC.00402EC0
00401B1D .^ EB DB PUSH SHORT BASECALC.00401AFA
00401B1F > 5B POP EBX
00401B20 . 5D POP EBP
00401B21 . C3 RETN
00401314=<JMP.&kernel32.DeleteCriticalSection>

```

Address of jmp stub pointing to IAT

Như các bạn thấy trên hình minh họa có một lệnh "CALL 00401314" nhưng Olly sẽ thay thế bằng tên của hàm cho chúng ta. 401314 là địa chỉ của the jmp stub pointing to the IAT. Chú ý rằng nó là phần của một bảng jmp thunk table đã được nói đến ở phần trước :

OllyDbg - BASECALC.EXE - [CPU - main thread, module BASECALC]

```

C File View Debug Plugins Options Window Help
[<] [X] [>] [II] [F1] [F2] [F3] [F4] [F5] [F6] [F7] [F8] [F9] [F10] [F11] [F12] [L] [E] [M] [T] [W] [H] [C / K] [B] [R ... S] [Grid] [?]
004012EA : 8BC0 MOV EAX,EAX
004012EC $-FF25 C8004200 JMP DWORD PTR DS:[<&kernel32.VirtualAlloc>]
004012F2 : 8BC0 MOV EAX,EAX
004012F4 $-FF25 C4004200 JMP DWORD PTR DS:[<&kernel32.VirtualFree>]
004012FA : 8BC0 MOV EAX,EAX
004012FC $-FF25 C0004200 JMP DWORD PTR DS:[<&kernel32.InitializeCriticalSection>]
00401302 : 8BC0 MOV EAX,EAX
00401304 $-FF25 B0004200 JMP DWORD PTR DS:[<&kernel32.EnterCriticalSection>]
0040130A : 8BC0 MOV EAX,EAX
0040130C $-FF25 B8004200 JMP DWORD PTR DS:[<&kernel32.LeaveCriticalSection>]
00401312 : 8BC0 MOV EAX,EAX
00401314 $-FF25 B4004200 JMP DWORD PTR DS:[<&kernel32.DeleteCriticalSection>]
00401318 : 8BC0 MOV EAX,EAX

```

DS:[0042D0B4]=7C91188A (ntdll.RtlDeleteCriticalSection)
Local call from 00401B12

Tại đây chúng ta lại quan sát thấy có một lệnh nhảy "JMP DWORD PTR DS:[0042D0B4]" ,nhưng lại một lần nữa Olly đã thay thế bằng symbolic name cho chúng ta. Địa chỉ 0042D0B4 chứa cấu trúc **Image_Thunk_Data structure** trong IAT mà đã được overwritten bởi trình loader bằng địa chỉ thực sự của hàm trong kernel32.DLL: 7C91188A. Đó là những gì mà chúng ta đã tìm thấy thông qua việc rightclicking and selecting Follow Import in Disassembler và cũng từ dumped file ở phần trên.

11. Adding Code to a PE File :

Việc thêm code vào một PE file là một điều rất cần thiết để không những có thể crack một protection scheme mà còn có thể được ứng dụng trong việc thêm các chức năng vào trong PE file. Có 3 phương pháp chính để có thể add code vào trong một file thực thì là :

1. Thêm vào một section hiện tại khi có đủ chỗ cho đoạn code của bạn.
2. Mở rộng section hiện tại khi không đủ chỗ.
3. Thêm một section mới hoàn toàn.

Adding to an existing section

Chúng ta cần một section trong file mà được ánh xạ với các quyền thực thi trong bộ nhớ vì vậy đơn giản nhất chúng ta hãy thực hành với CODE section. Sau đó chúng ta cần một vùng chứa toàn byte 00 (00 byte padding) trong section này. Vùng này được gọi với một tên chung là “caves” Để có thể tìm được một “cave” phù hợp với những gì chúng ta mong đợi , chúng ta sẽ quan sát tại CODE section . Chi tiết thông qua chương trình LorPE :

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

Trong hình minh họa trên chúng ta quan sát thấy **VirtualSize** nhỏ hơn **SizeOfRawData**.Virtual size biểu diễn số lượng code thực sự. Còn kích thước của raw data xác định số lượng của không gian được sử dụng cho file trên đĩa cứng của bạn. Chú ý rằng virtual size trong trường hợp này là thấp hơn với virtual size trên đĩa cứng. Đó là bởi vì các trình compiler thường làm tròn kích thước lên để sắp xếp một section trên một vài ranh giới. Trong chương trình Hexeditor quan sát tại phía cuối của CODE section (phía trước của DATA section bắt đầu tại 2A400h) , chúng ta có được như sau :

0002a250h:	10 93 FD FF 8B E5 5D C3 FF FF FF FF OF 00 00 00 ; .``ýý<á] Áÿÿÿÿ... 0002a260h:	42 61 73 65 20 43 61 6C 63 75 6C 61 74 6F 72 00 ; Base Calculator. 0002a270h:	FF FF FF FF OC 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.0.8 0.à!0. 0002a410h:	34 25 40 00 32 1F 8B C0 32 13 8B C0 52 75 6E 74 ; 4%0.2.<à2.<àRunt 0002a420h:	69 6D 65 20 65 72 72 6F 72 20 20 20 20 20 61 74 ; imo error at 0002a430h:	20 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

Không gian thêm này là hoàn toàn không được sử dụng và không được ánh xạ vào trong bộ nhớ. Chúng ta cần phải bảo đảm chắc chắn rằng những câu lệnh mà chúng ta đặt vào không gian này sẽ được nạp vào trong bộ nhớ. Chúng ta thực hiện điều đó bằng cách chỉnh sửa thuộc tính size (Size attribute). Ngay bây giờ chúng ta thấy là kích thước ảo của Section này là 29E88, đó là bởi vì tất cả các trình compiler đều cần.Còn đối với chúng ta chúng ta phải cần tăng lên một chút nữa, vì vậy trong LordPE ta thay đổi virtual size của CODE section lên thành 29FFF , đó là kích thước lớn nhất mà chúng ta có thể sử dụng (Toàn bộ Raw size chỉ có 2A000). Để thực hiện được điều này , chúng ta chuột phải tại dòng CODE và chọn edit header, thực hiện thay đổi với giá trị trên và save lại .

Sau khi thực hiện xong chúng ta đã có một không gian thích hợp để lưu giữ đoạn patch code của chúng ta. Điều duy nhất mà chúng ta đã thay đổi là VirtualSize DWORD cho CODE section trong bảng Section Table. Chúng ta cũng có thể thực hiện được công việc này bằng tay thông qua chương trình HexEditor.

Để minh họa thêm nữa cho công việc này chúng ta sẽ tiến hành thêm vào chương trình ví dụ của chúng ta một chương trình ASM nhỏ thực hiện việc chiếm lấy điều khiển của entry point và sau đó chỉ trả về sự thực thi cho **OriginalEntryPoint**. Tất cả công việc này được làm thông qua Ollydbg.

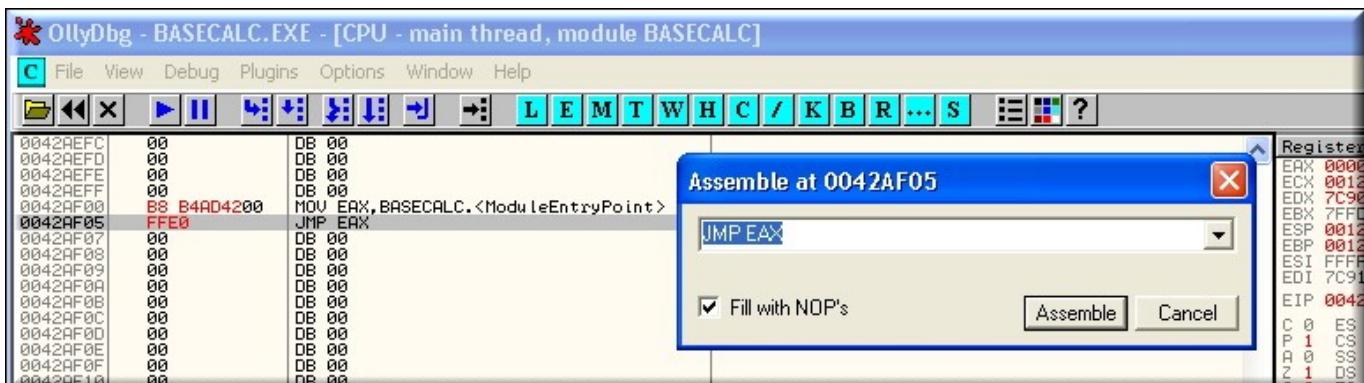
Đầu tiên chúng ta để ý trong LordPE thì EntryPoint là 0002ADB4 và ImageBase là 400000. Khi chúng ta load chương trình vào trong Olly thì EP sẽ là 0042ADB4. Chúng ta sẽ thêm một số dòng sau và sau đó thay đổi entry point tới dòng đầu tiên của đoạn code :

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

Chúng ta sẽ để các lệnh trên tại địa chỉ 0002A300h như chúng ta đã quan sát trên chương trình Hexeditor. Để convert RAW offset này sang một RVA để sử dụng cho Olly ta sẽ sử dụng công thức sau đây (Xem thêm phần phụ lục) :

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

Sau đó ta load chương trình vào trong Olly và nhảy tới target section của chúng ta (nhấn Ctrl + G và gõ vào giá trị đã tính toán được ở trên là **42AF00h**). Sau khi tới vị trí này, ta nhấn Space, gõ vào dòng đầu tiên của đoạn code trên sau đó nhấn assemble. Tiếp theo làm tương tự với dòng code thứ hai. Ta có được tương tự như hình minh họa dưới đây :



Tiếp theo nhấn chuột phải, chọn tùy chọn Copy to Executable and All modifications. Tiếp theo chọn Cpy all, một cửa sổ mới sẽ xuất hiện. Trên cửa sổ mới này tiếp tục nhấn chuột phải và chọn Save File v...v...Bây giờ chúng ta quay trở lại với LordPE (hay chương trình HexEditor) và thay đổi EntryPoint thành 0002AF00 (ImageBase Subtracted), chọn Save và nhấn OK. Chúng ta Run chương trình để kiểm tra và reopen nó trong Olly để xem New EntryPoint của chúng ta. Trong chương trình HexEditor chúng ta sẽ quan sát thấy như sau, chú ý đoạn được Highlight :

	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
0002a250h:	10	93	FD	FF	8B	E5	5D	C3	FF	FF	FF	FF	0F	00	00	00
0002a260h:	42	61	73	65	20	43	61	6C	63	75	6C	61	74	6F	72	00
0002a270h:	FF	FF	FF	OC	00	00	00	42	41	53	45	43	41	4C	43	;
0002a280h:	2E	48	4C	50	00	00	00	00	00	00	00	00	00	00	00	;
0002a290h:	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	;
0002a2b0h:	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	;
0002a2d0h:	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	;
0002a2f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;
0002a300h:	B8	B4	AD	42	00	FF	EO	00	00	00	00	00	00	00	00	;
0002a310h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;

Mặc dù đây chỉ là một đoạn tiny patch , nhưng chúng ta hoàn toàn có đủ không gian cho 386 bytes của New code.

Enlarging an Existing Section

Nếu như không có đủ không gian tại phía cuối của section .text thì chúng ta cần phải mở rộng nó. Điều này đưa ra một số vấn đề như sau :

1. Nếu section được followed bởi các section khác thì bạn sẽ cần phải dịch chuyển các following sections lên để tạo không gian.
2. Có rất nhiều các references khác nhau bên trong các file headers mà sẽ cần phải được điều chỉnh nếu bạn thay đổi kích thước của file.

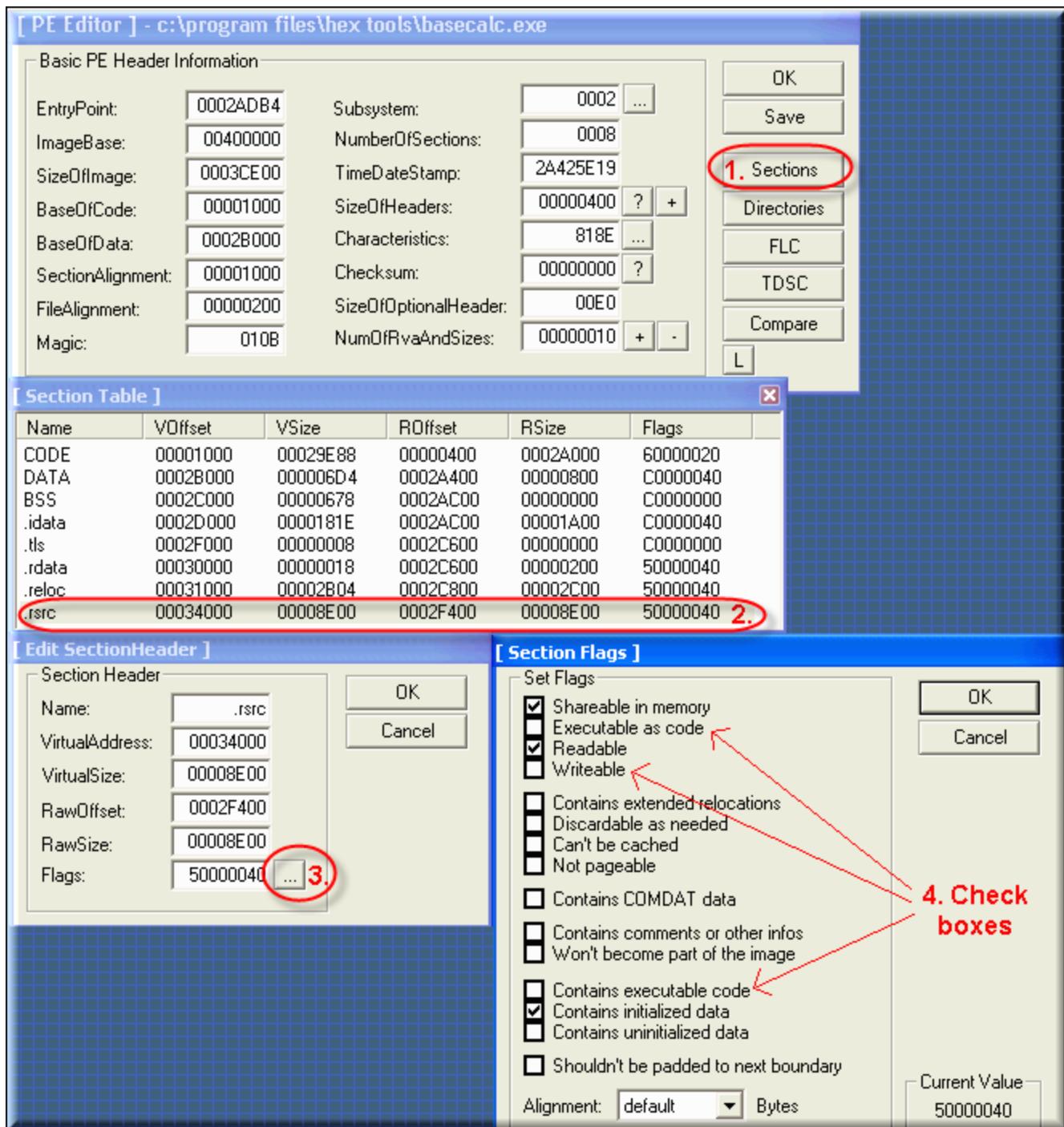
3. Các References giữa các sections khác nhau (ví dụ references tới data values từ code section) sẽ cần phải được điều chỉnh. Về thực tế là hầu như không thể thực hiện được nếu như thiếu việc re-compiling and re-linking file gốc.

Hầu hết các vấn đề nêu trên đều có thể tránh được bằng cách nối thêm và section cuối cùng trong file exe. Nó chẳng có liên quan gì tới section đó nếu như chúng ta có thể thay đổi khiến nó phù hợp với yêu cầu của chúng ta bằng cách thay đổi trường **Characteristic** trong **Section Table** bằng tay hoặc bằng LordPE.

Đầu tiên chúng ta tìm đến section cuối cùng và thay đổi nó sao cho nó thành **readable and executable**. Như chúng ta đã nói ở trên code section chỉ là ý tưởng cho một patch bởi vì các characteristics flags của nó là 60000020 , điều đó có nghĩa là đoạn mà có thể thực thi được và có thể đọc được (executable and readable) – (Xin xem thêm phần phụ lục). Tuy nhiên nếu chúng ta đặt đoạn mã và dữ liệu vào trong section này thì chúng ta sẽ nhận được một page fault vì nó không phải là writeable. Để thay đổi điều này chúng ta sẽ cần phải thêm flag 800000000 mà sẽ cho ta một giá trị mới là E0000020 cho **code, executable, readable and writable**.

Tương tự như vậy nếu section cuối cùng là .reloc thì flags thường sẽ là 42000040 cho initialized data, discardable and read-only. Để có thể sử dụng được section này chúng ta phải thêm code, executable and writable và chúng ta phải trừ discardable để đảm bảo chắc chắn rằng trình loader sẽ ánh xạ section này vào trong bộ nhớ. Điều này sẽ cho chúng ta một giá trị mới là E0000060.

Các công việc trên có thể thực hiện thành công bằng tay bằng cách thêm flags và chỉnh sửa lại trường **Characteristics** của Section header thông qua chương trình HexEditor hoặc LordPE. Trong ví dụ của chúng ta thì section cuối cùng là Resources :



Điều này sẽ cho chúng ta một giá trị **Characteristics** cuối cùng là F0000060. Như hình minh họa trên chúng ta quan sát thấy **RawSize** (on disk) của section này là 8E00h bytes nhưng tất cả chúng dường như đang được sử dụng (the **VirtualSize** cũng giống hệt). Bây giờ chúng ta chỉnh lại chúng và cộng 100h bytes vào cả hai để mở rộng section , giá trị mới chúng ta có được là 8F00h. Có một vài giá trị quan trọng khác cũng cần được thay đổi. Trường **SizeOfImage** trong PE Header cần phải được tăng lên bằng cách cộng thêm vào giá trị giống như chúng ta đã thêm để mở rộng cho section đó là 100h. Do đó giá trị **SizeOfImage** sẽ thay đổi 0003CE00h thành 0003CF00h.

Có 2 trường khác nữa mà không được thể hiện trong LordPE bởi vì chúng ít quan trọng đó là : **SizeOfCode** và **SizeOfInitialisedData** trong **Optional Header**. Ứng dụng sẽ vẫn thực thi mà không cần được chỉnh sửa nhưng có lẽ bạn nên thay đổi lại chúng để cho trọn vẹn.Chúng ta sẽ phải thay đổi lại chúng bằng tay. Cả hai đều là DWORDs tại các offset 1C và 20 từ điểm bắt đầu của PE header. (xem thêm phần phụ lục).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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
00000110h:	00	00	00	00	E0	00	8E	81	OB	01	02	19	00	A0	02	00
00000120h:	00	DE	00	00	00	00	00	00	B4	AD	02	00	00	10	00	00
00000130h:	00	B0	02	00	00	00	40	00	00	10	00	00	02	00	00	00

Các giá trị 0002A000 và 0000DE00 tương ứng với các vị trí xác định như các bạn đã thấy trên hình minh họa. Khi chúng ta cộng thêm 100h vào thì các giá trị này sẽ là 0002A100 và 0000DF00. Sau đó chúng ta sẽ đảo ngược thứ tự của các giá trị trên thành 00 A1 02 00 và 00 00 DF 00. Cuối cùng copy và paste 100h of 00 bytes (16 hàng trong trình Hexeditor) lên phía cuối của Section và lưu lại thay đổi. Chạy file để kiểm tra các lỗi.

Adding a New Section

Trong một vài tình huống bạn có thể cần phải tạo ra một bản sao của một section đang tồn tại để phá vỡ các self-checking procedures (Ví dụ như SafeDisk) hoặc tạo ra một section mới để lưu code khi các thông tin thuộc quyền sở hữu riêng được bổ sung thêm vào cuối của file (as in Delphi compiled apps).

Công việc đầu tiên cần làm là phải tìm đến trường **NumberOfSections** trong PE header và tăng trường này lên 1. Như đã nói trong những phần trước hầu hết mọi sự thay đổi có thể được thực hiện bằng chương trình LordPE hoặc bằng tay thông qua chương trình HexEditor. Bây giờ trong chương trình HexEditor của bạn hãy copy và paste 100h of 00 bytes (16 rows) lên phần cuối của file và đánh dấu offset của dòng mới đầu tiên. Trong trường hợp của chúng ta đó là 00038200h. Đó sẽ là nơi bắt đầu section mới của chúng ta và sẽ đi tới trường **RawOffset** field của **Section Header**. Khi chúng ta ở đây thì chắc chắn đó là thời điểm tốt để tăng **SizeOfImage** lên 100h như chúng ta đã làm ở trước.

Tiếp theo chúng ta sẽ tìm tới các section headers bắt đầu tại offset F8 từ PE header. It is not necessary for these to be terminated by a header full of zeros. Số lượng các headers được đưa ra bởi NumberOfSections và đó thường là một vài không gian tại phía cuối trước khi bản thân các sections bắt đầu. (aligned to the FileAlignment value). Tìm đến section cuối cùng và thêm một giá trị mới sau nó :

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000002c0h:	2E	72	64	61	74	61	00	00	18	00	00	00	00	03	00	;
000002d0h:	00	02	00	00	00	C6	02	00	00	00	00	00	00	00	00	;
000002e0h:	00	00	00	00	40	00	00	50	2E	72	65	6C	6F	63	00	00
000002f0h:	04	2B	00	00	00	10	03	00	00	2C	00	00	00	C8	02	00
00000300h:	00	00	00	00	00	00	00	00	00	00	00	40	00	00	50	;
00000310h:	2E	72	73	72	63	00	00	00	00	8E	00	00	00	40	03	00
00000320h:	00	8E	00	00	00	F4	02	00	00	00	00	00	00	00	00	00
00000330h:	00	00	00	00	40	00	00	50	00	00	00	00	00	00	00	00
00000340h:	00	00	00	00	00	D0	03	00	00	00	00	82	03	00	00	;
00000350h:	00	00	00	00	00	00	00	00	00	00	00	40	00	00	50	;
00000360h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000370h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000380h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000390h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Phần tiếp theo mà chúng ta phải làm là quyết định xem các thành phần Virtual Offset/Virtual Size/Raw Offset and Raw Size nào cần có. Để có thể quyết định được điều này chúng ta xem xét các giá trị sau :

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

This 40 byte section is incomplete and seems to relate to a non-existent section so we will make it our new section.

RVA và raw offset của section mới của chúng ta phải được căn chỉnh với boundaries ở trên.RAW Offset của section là 00038200h như chúng ta đã nói ở trên (which luckily fits with FileAlignment). Để có được Virtual Offset của section của chúng ta thì chúng ta phải tính toán giá trị này : VirtualAddress of .rsrc + VirtualSize of .rsrc = 3CE00h. Vì SectionAlignment của chúng ta là 1000h chúng ta phải làm tròn giá trị này lên gần giống như 1000 tức là 3D000h. Vì vậy hãy điền vào header của section của chúng ta :

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)

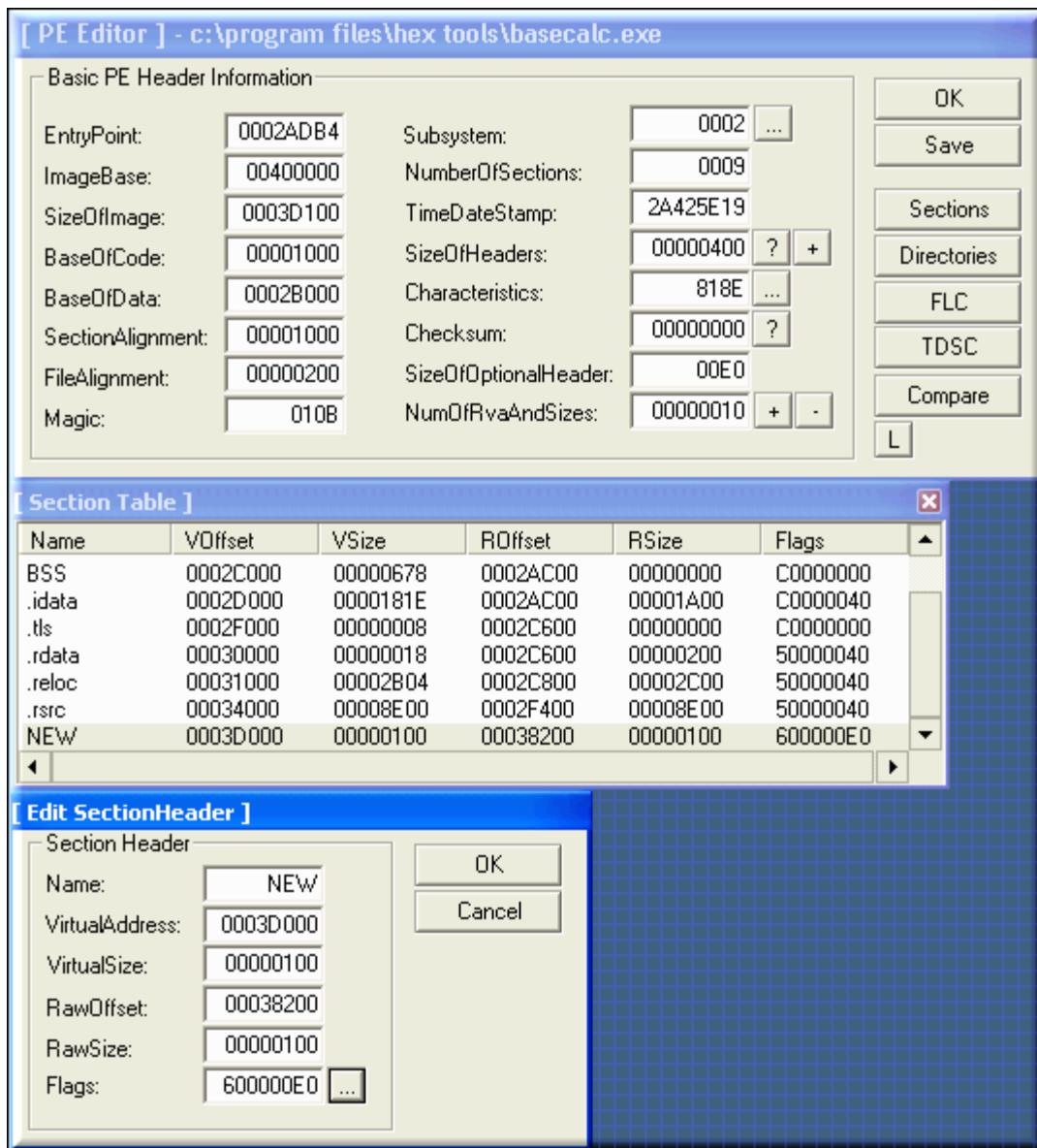
The next 12 bytes can be left null

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

Trong trình HexEditor chúng ta sẽ thấy như sau :

	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....ż...0..	
00000320h:	00	8E	00	00	00	F4	02	00	00	00	00	00	00	00	00	00	; .ż....ô.....	
00000330h:	00	00	00	00	40	00	00	50	4E	45	57	00	00	00	00	00	;0...PNEW.....	
00000340h:	00	01	00	00	00	D0	03	00	00	01	00	00	00	82	03	00	;Đ.....,	
00000350h:	00	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	00	;

Lưu lại thay đổi , chúng sẽ run chương trình và kiểm tra trong LordPE :



12. Adding Imports to an Executable :

Phương pháp này thường được sử dụng nhiều nhất trong trường hợp Patching một App khi mà chúng ta không có các hàm API mà chúng ta cần. Để thêm section mới, thì thông tin tối thiểu nhất được yêu cầu bởi trình loader để tạo ra một IAT hợp lệ là :

1. Mỗi Dll phải được khai báo với một **IMAGE_IMPORT_DESCRIPTOR (IID)**, nhớ kết thúc Import Directory bằng một null-filled.
2. Mỗi **IID** cần ít nhất 2 trường là Name1 và FirstThunk, phần còn lại có thể được set là 0(setting OriginalFirstThunk = FirstThunk i.e. duplicating the RVAs also works).
3. Mỗi entry của FirstThunk phải là một RVA tới một Image_Thunk_Data (the IAT) mà lần lượt chứa một further RVA tới API name.Tên phải là một chuỗi null terminated ASCII của độ dài có thể thay đổi và được đi trước bởi 2 bytes (**hint**) mà có thể được thiết lập là 0.

4. Nếu các IID đã được thêm thì trường **isize** của Import Table trong Data Directory có thể cần phải thay đổi. Các IAT entries trong Data Directory không cần phải được chỉnh sửa.

Việc viết import data mới trong một chương trình HexEditor và sau đó dán vào trong target của bạn có thể sẽ tốn rất nhiều thời gian. Có các công cụ có sẵn có thể thực hiện được một cách tự động quá trình này (Ví dụ : SnippetCreator, IIDKing, Cavewriter) nhưng việc tìm hiểu cách thực hiện công việc này bằng tay như thế nào vẫn là tốt hơn cả. Nhiệm vụ chính là để nối thêm một IID mới lên phần cuối của bảng Import Table – bạn sẽ cần có 20 bytes cho mỗi DLL được sử dụng, đừng quên 20 bytes dành cho null-terminator. Trong hầu hết tất cả các trường hợp sẽ không có không gian nào tại phía cuối của Import Table hiện hành vì vậy chúng ta sẽ tạo một bản sao và xây dựng lại nó ở một nơi nào đó.

Step 1 - create space for new a new IID

Công việc này liên quan đến các bước sau đây :

1. Dịch chuyển tất cả các IIDS tới một vị trí mà tại đó có đủ không gian. Vị trí này có thể ở bất kì đâu; phía cuối của section **.idata** hiện thời hoặc một section mới hoàn toàn.
2. Cập nhật RVA của Import Directory mới trong Data Directory của PE Header.
3. Nếu cần thiết, làm tròn kích thước của section nơi mà bạn đã đặt Import Table mới vì vậy mọi thứ đều được ánh xạ vào trong bộ nhớ (ví dụ : VirtualSize of the .idata section rounded up 1000h).
4. Chạy nó và nếu như nó làm việc thì chuyển tới bước 2. Nếu nó không kiểm tra các injected descriptors được ánh xạ vào trong bộ nhớ và RVA của Import Directory là chính xác.....

IMPORTANT NOTE: Các IIDS **FirstThunk** và **OriginalFirstThunk** chứa các RVAs- RELATIVE ADDRESSES – có nghĩa là các bạn có thể cắt và dán Import Directory (IIDs) ở bất kì đâu bạn muốn trong PE file (taking into account the destination has to mapped into memory) và thay đổi RVA (và kích thước nếu cần thiết) của Import Directory trong Data Directory sẽ khiến cho ứng dụng hoạt động một cách hoàn hảo.

Quay trở lại ứng dụng của chúng ta trong trình Hexeditor, IID đầu tiên và null terminator được tô bằng đường bao màu đỏ. Như bạn nhìn thấy trong hình vẽ dưới đây không có không gian trống nào sau null IID:

0002ac00h:	00 00 00 00 00 00 00 00 00 00 00 00 30 D5 02 00	;00
0002ac10h:	B4 D0 02 00 00 00 00 00 00 00 00 00 00 00 00 00	;	'D.....
0002ac20h:	06 D7 02 00 24 D1 02 00 00 00 00 00 00 00 00 00 00	;	.x...\$Ñ..
0002ac30h:	00 00 00 00 20 D7 02 00 2C D1 02 00 00 00 00 00 00	;x...Ñ..
0002ac40h:	00 00 00 00 00 00 00 8A D7 02 00 44 D1 02 00	;Šx..DÑ..
0002ac50h:	00 00 00 00 00 00 00 00 00 00 00 FC D7 02 00	;üx..
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 00	;	NÚ..šÑ..
0002ac80h:	00 00 00 30 DE 02 00 D4 D2 02 00 00 00 00 00 00	;OP..ÔÔ..
0002ac90h:	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	;>Ö..vÖ..nÖ..
0002acc0h:	86 D5 02 00 A2 D5 02 00 B0 D5 02 00 C0 D5 02 00	;	tÖ..çÖ..°Ö..ÀÖ..
0002acd0h:	CC D5 02 00 DA D5 02 00 F0 D5 02 00 FE D5 02 00	;	ìÖ..úÖ..sÖ..þÖ..

Tuy nhiên có một số lượng không gian lớn tại phần cuối của section **.idata** trước khi section **.rdata** bắt đầu. Chúng ta sẽ copy và paste các IIDS hiện thời được đưa ra phía trên tới offset 2C500h tại vị trí mới này :

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0002c4f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0002c500h:	00	00	00	00	00	00	00	00	00	00	30	D5	02	00	;0õ..
0002c510h:	B4	D0	02	00	00	00	00	00	00	00	00	00	00	00	00	00
0002c520h:	06	D7	02	00	24	D1	02	00	00	00	00	00	00	00	00	00
0002c530h:	00	00	00	00	20	D7	02	00	2C	D1	02	00	00	00	00	00
0002c540h:	00	00	00	00	00	00	00	00	8A	D7	02	00	44	D1	02	00
0002c550h:	00	00	00	00	00	00	00	00	00	00	00	00	FC	D7	02	00
0002c560h:	60	D1	02	00	00	00	00	00	00	00	00	00	00	00	00	00
0002c570h:	4E	DA	02	00	F0	D1	02	00	00	00	00	00	00	00	00	00
0002c580h:	00	00	00	00	30	DE	02	00	D4	D2	02	00	00	00	00	00
0002c590h:	00	00	00	00	00	00	00	00	F6	E6	02	00	FC	D4	02	00
0002c5a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0002c5b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0002c5c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0002c5d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0002c5e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Để convert một offset mới thành RVA (xem thêm phần phụ lục) :

$$\begin{aligned} \text{VA} &= \text{RawOffset} - \text{RawOffsetOfSection} + \text{VirtualOffsetOfSection} \\ &= 2\text{C}500 - 2\text{AC}00 + 2\text{D}000 = \text{2E}900h \end{aligned}$$

Vậy thay đổi địa chỉ ảo của import table trong Data Directory từ 2D000 thành 2E900. Bây giờ chỉnh sửa lại header của section .idata và thay đổi VirtualSize bằng với RawSize vì vậy trình loader sẽ ánh xạ toàn bộ section vào. Chạy thử ứng dụng của chúng ta để test.

Step 2 - Add the new DLL and function details

Công việc này bao gồm một số bước sau :

- Thêm null-terminated ASCII strings các tên của DLL của bạn và hàm vào không gian còn trống trong section .idata. Tên hàm sẽ thực sự là một cấu trúc Image_Import_By_Name được preceded bởi một null DWORD. (the hint field).
- Tính toán các RVAs của các string trên.
- Thêm RVA của tên DLL vào trường **Name1** của IID mới của bạn.
- Tìm DWORD sized space khác nữa và đặt vào nó RVA của hint/function name. Nó sẽ trở thành **Image_Thunk_Data** hoặc IAT của DLL mới của chúng ta.
- Tính toán RVA của **Image_Thunk_Data** DWORD ở trên và thêm nó vào trường FirstThunk của IID mới của bạn.
- Chạy ứng dụng để test ... API mới của bạn đã sẵn sàng được gọi...

Để điền vào IDD mới của chúng ta , chúng ta ít nhất phải có các trường là **Name1** và **FirstThunk** (các trường khác có thể nulled). Như chúng ta đã biết, trường Name1 chứa thông tin RVA tên của DLL trong null-terminated ASCII. Trường **FirstThunk** chứa RVA của một cấu trúc **Image_Thunk_Data** mà lần lượt chứa RVA khác nữa của tên hàm trong null-terminated ASCII. Tên tuy nhiên được đi trước bởi 2 bytes (Hint) mà được thiết lập là zero.

Lấy một ví dụ , chúng ta muốn sử dụng hàm **LZCopy** mà copy toàn bộ một file nguồn tới một file đích. Nếu file nguồn của chúng ta được nén bằng trình ứng dụng Microsoft File Compression Utility

(COMPRESS.EXE), thì hàm này tạo ra một file đích đã được giải nén. Nếu như file nguồn không bị nén , thì hàm này sẽ nhân đôi file gốc lên.

Hàm mà chúng ta nói ở trên nằm trong file dll là **Iz32.dll** mà hiện thời không được sử dụng bởi chương trình ứng dụng của chúng ta. Vì vậy đầu tiên chúng ta cần phải thêm strings cho các tên là “Iz32.dll” và “LZCopy”. Trong trình Hexeditor chúng ta cuộn lên trên từ chỗ bảng import table mới của chúng ta về phía cuối của dữ liệu đã tồn tại trước đó và thêm tên DLL sau đó là tên hàm lên phần cuối này . Chú ý, các bytes null sau mỗi string và null DWORD trước tên hàm :

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0002c3f0h:	65	49	63	6F	6E	00	00	00	49	6D	61	67	65	4C	69	73	
0002c400h:	74	5F	44	65	73	74	72	6F	79	00	00	00	49	6D	61	67	
0002c410h:	65	4C	69	73	74	5F	43	72	65	61	74	65	00	00	00	00	
0002c420h:	6C	7A	33	32	2E	64	6C	6C	00	00	00	00	00	00	00	00	
0002c430h:	00	00	4C	5A	43	6F	70	79	00	00	00	00	00	00	00	00	
0002c440h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c450h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c460h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c470h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c480h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c490h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c4a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c4b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c4c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c4d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c4e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c4f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c500h:	00	00	00	00	00	00	00	00	00	00	00	00	00	30	D5	02	00
0002c510h:	B4	D0	02	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002c520h:	06	D7	02	00	24	D1	02	00	00	00	00	00	00	00	00	00	

Chúng ta cần phải tính lại các RVA của chúng :

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

$$\text{RVA of DLL name} = 2\text{C420} - 2\text{AC00} + 2\text{D000} = 2\text{E820h} \quad (20 \text{ E8 } 02 \text{ 00 in reverse})$$

$$\text{RVA of function name} = 2\text{C430} - 2\text{AC00} + 2\text{D000} = 2\text{E830h} \quad (30 \text{ E8 } 02 \text{ 00 in reverse})$$

Giá trị đầu tiên có thể nằm trong trường Name1 của IDD mới của chúng ta nhưng giá trị thứ hai thì phải nằm trong một cấu trúc **Image_Thunk_Data structure**, với RVA của chúng, chúng ta sau đó có thể đặt vào trong trường **FirstThunk** (and OriginalFirstThunk) của IDD mới của chúng ta.Chúng ta sẽ đặt cấu trúc **Image_Thunk_Data structure** bên dưới tên hàm tại offset 2C440 và tính toán RVA mà chúng ta sẽ đặt vào FirstThunk.

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

Nếu chúng ta điền dữ liệu trong trình HexEditor chúng ta sẽ thấy như sau :

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0002c420h:	6C	7A	33	32	2E	64	6C	6C	00	00	00	00	00	00	00	00 ; lz32.dll.....
0002c430h:	00	00	4C	5A	43	6F	70	79	00	00	00	00	00	00	00	; ..LZCopy.....
0002c440h:	30	E8	02	00	00	00	00	00	00	00	00	00	00	00	00	; 0è ..
0002c450h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c460h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c470h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c480h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c490h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c4a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c4b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c4c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c4d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c4e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c4f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..
0002c500h:	00	00	00	00	00	00	00	00	00	00	00	30	D5	02	00	;ö..
0002c510h:	B4	D0	02	00	00	00	00	00	00	00	00	00	00	00	00	; 'Đ.....
0002c520h:	06	D7	02	00	24	D1	02	00	00	00	00	00	00	00	00	; ..x..\$Ñ..
0002c530h:	00	00	00	00	20	D7	02	00	2C	D1	02	00	00	00	00	;x.,Ñ..
0002c540h:	00	00	00	00	00	00	00	00	8A	D7	02	00	44	D1	02	00 ;Šx..DÑ..
0002c550h:	00	00	00	00	00	00	00	00	00	00	00	FC	D7	02	00	;üx..
0002c560h:	60	D1	02	00	00	00	00	00	00	00	00	00	00	00	00	; 'Ñ.....
0002c570h:	4E	DA	02	00	F0	D1	02	00	00	00	00	00	00	00	00	; NÚ..&Ñ..
0002c580h:	00	00	00	00	30	DE	02	00	D4	D2	02	00	00	00	00	;OP..öö..
0002c590h:	00	00	00	00	00	00	00	00	F6	E6	02	00	FC	D4	02	00 ;æ..üö..
0002c5a0h:	00	00	00	00	00	00	00	00	00	00	00	20	E8	02	00	; è..
0002c5b0h:	40	E8	02	00	00	00	00	00	00	00	00	00	00	00	00	; 0è..
0002c5c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	; ..

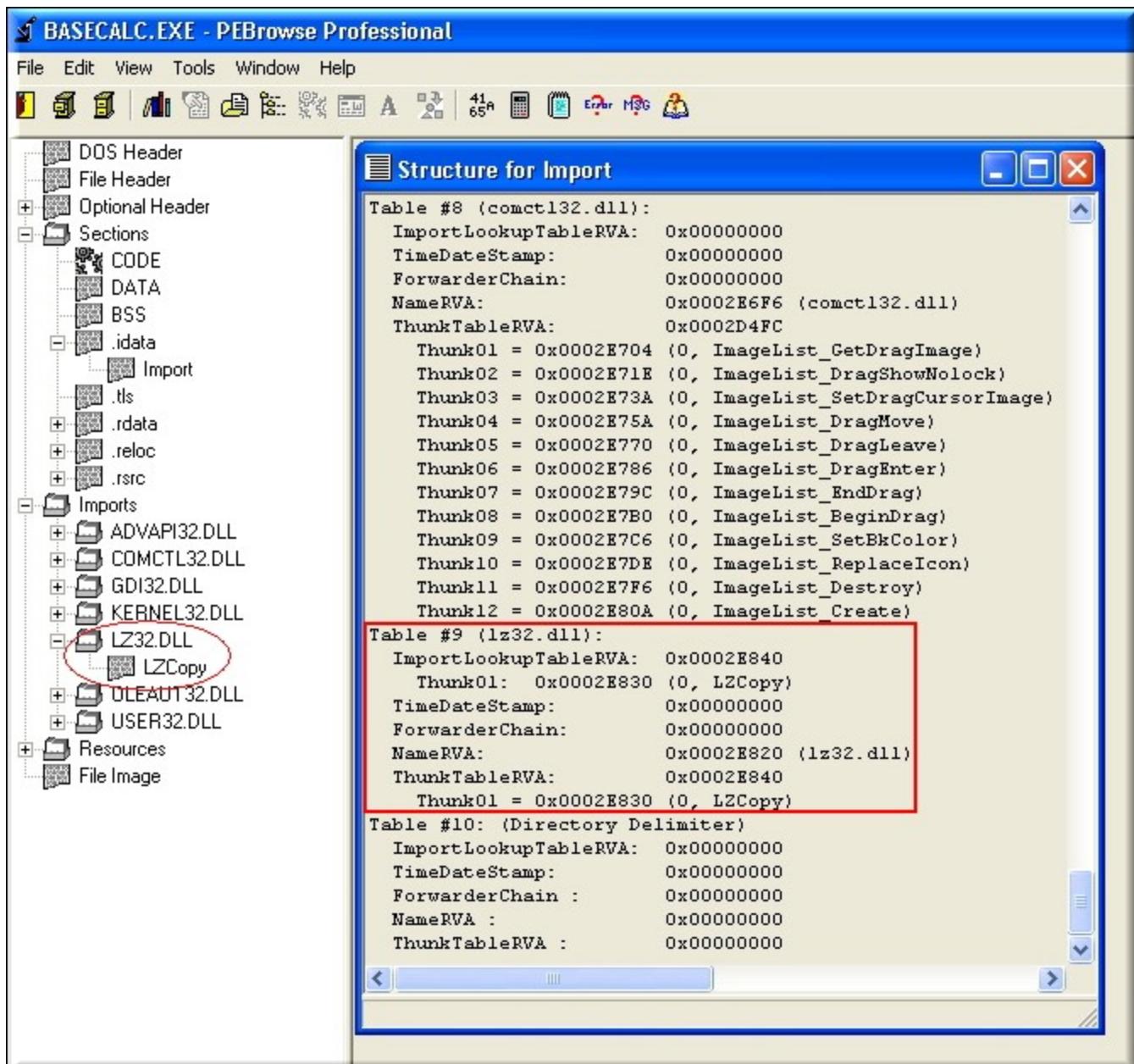
Cuối cùng chúng ta lưu lại những gì chúng ta đã thực hiện , chạy thử ứng dụng và load nó vào chương trình PEBrowse :

New

Image Thunk Data

Relocated Import Table

New IID



Để có thể gọi được hàm mới của chúng ta , chúng ta cần phải sử dụng đoạn code sau :

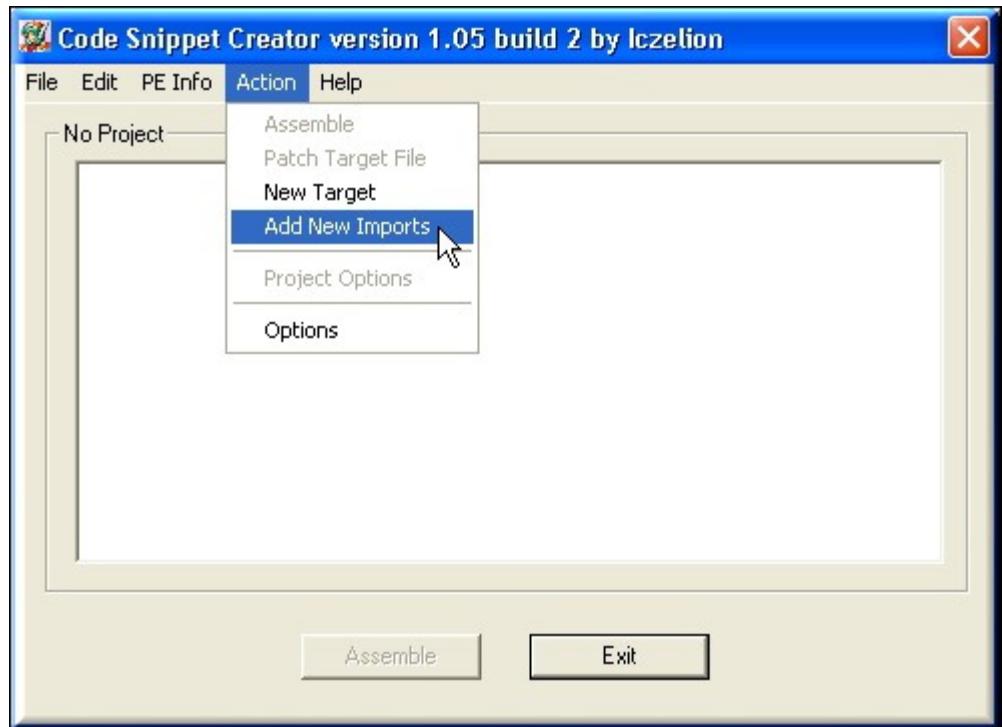
CALL DWORD PTR [XXXXXXXX] where XXXXXXXX = RVA of Image_Thunk_Data + ImageBase.

Trong ví dụ của chúng ta ở trên đối với hàm LZCopy, XXXXXXXX = 2E840 + 400000 = 42E840 vì vậy chúng ta sẽ viết là :

CALL DWORD PTR [0042E840]

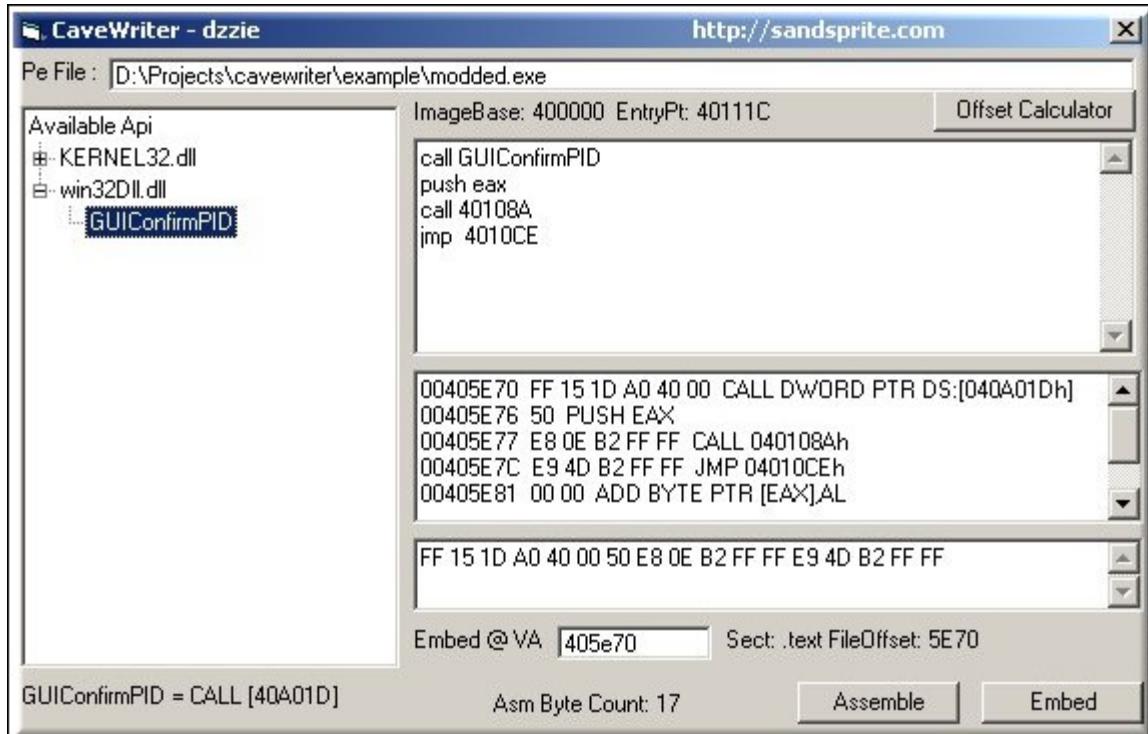
Chú ý cuối cùng : Dù là nếu chúng ta đã thêm một hàm được sử dụng bởi một DLL mà sẵn sàng được dùng trong kernel32.dll , chúng ta sẽ vẫn cần phải tạo ra một IDD mới cho nó để cho phép chúng ta có thể tạo một IAT mới tại một vị trí thuận lợi như ở trên.

Phần tiếp theo , đây chỉ là một phần được thêm vào trong section này. Sẽ có một cách tự động hoàn toàn thực hiện các công việc như đã nói ở trên :



Chú ý , Chương trình **SnippetCreator** thêm các jump-thunks stubs của các imports mới vào trong code của bạn trong khi với các chương trình khác bạn hoàn toàn phải thực hiện điều này bằng tay .

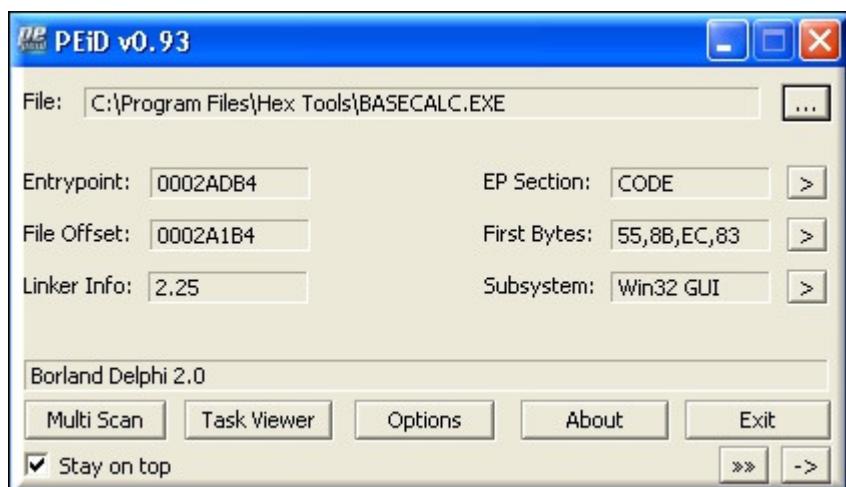




13. Introduction to Packers :

Trong phần này chúng ta sẽ mô tả sự tác động của một chương trình Packer đơn giản đối với ứng dụng của chúng ta và đề cập tới 2 phương pháp chính của việc Patching một file thực thi đã bị Packed – bằng cách Unpacking hoặc inline-patching. Chúng ta sẽ sử dụng Packer UPX 1.25 bởi vì đây thực sự là một chương trình nén file thực thi và không sử dụng bất kì một cơ chế bảo vệ cao cấp nào. Tác giả của chương trình này là Marcus & Laszlo.

Đầu tiên chúng ta dùng PeID để Scan file của chúng ta (file ban đầu chưa bị Packed) :



Tiếp theo chúng ta sẽ pack ứng dụng của chúng ta bằng chương trình UPX. Đây là chương trình sử dụng giao diện command line do đó chúng ta phải mở nó trong DOS , sau đó chúng ta gõ như sau : "**upx basecalc.exe**" :

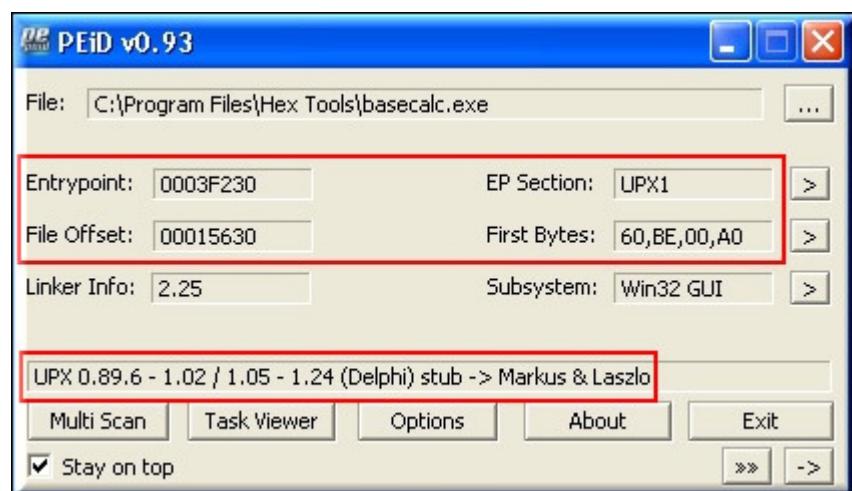
```
C:\WINDOWS\system32\cmd.exe
C:\Program Files\Hex Tools>upx basecalc.exe
      Ultimate Packer for eXecutables
Copyright <C> 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004
UPX 1.25w          Markus F.X.J. Oberhumer & Laszlo Molnar   Jun 29th 2004

  File size       Ratio     Format      Name
-----  -----  -----  -----
  229888 ->    92672    40.31%    win32/pe  basecalc.exe

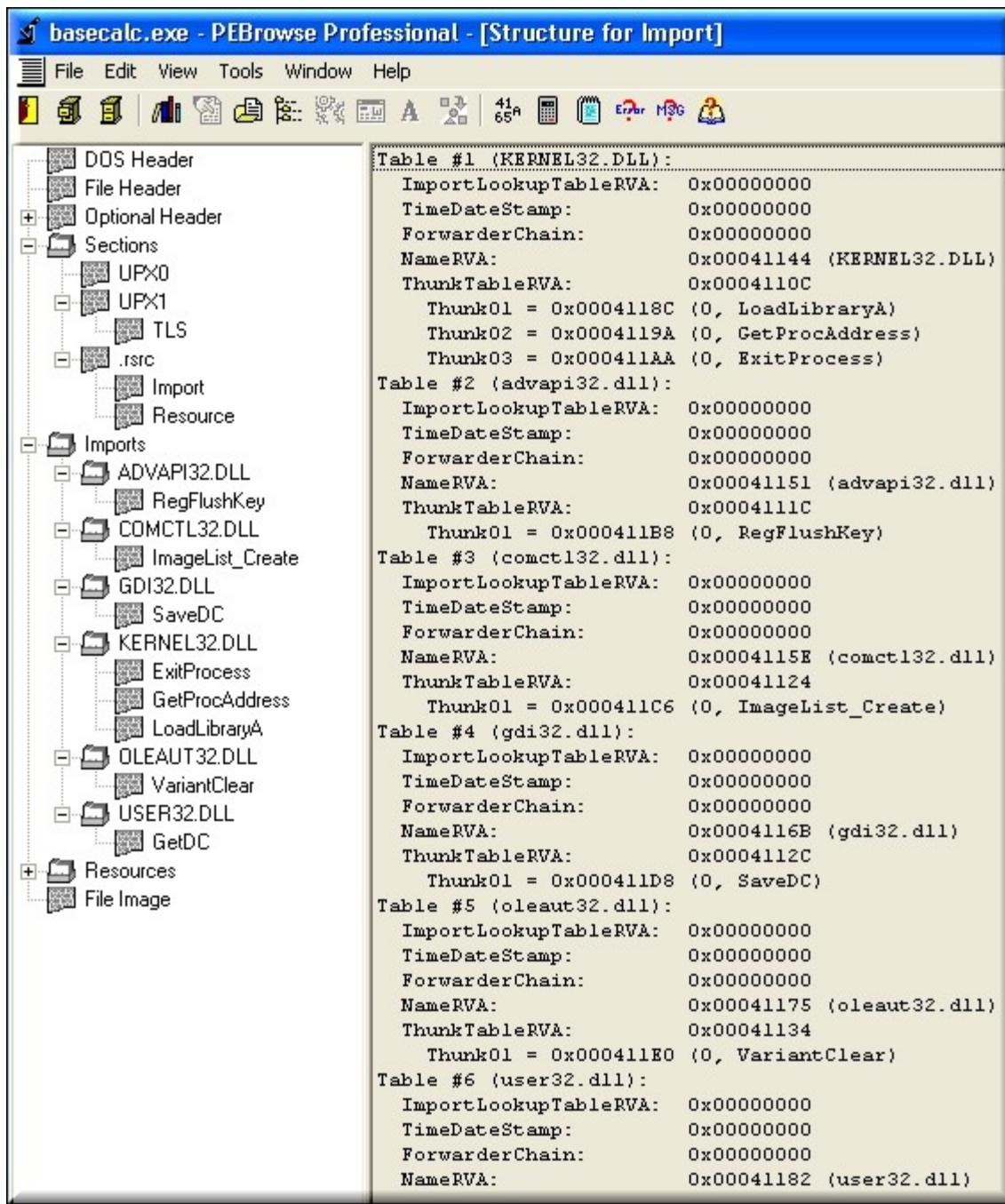
Packed 1 file.

C:\Program Files\Hex Tools>_
```

Sau đó chúng ta hãy để ý rằng kích thước chương trình của chúng ta đã giảm xuống từ 225kb xuống còn 91kb và trong PeID chúng ta quan sát thấy như sau :



Sử dụng chương trình PEBrowse Pro chúng ta quan sát thấy trình Packer sẽ thêm vào app của chúng ta 3 sections là **UPX0**, **UPX1** and **.rsrc**. Resource section bây giờ chứa import directory nhưng cho mỗi DLL thì chỉ có duy nhất một hoặc 2 hàm được imported – các hàm khác đã biến mất :



Chú ý rằng section **.rsrc** đã được giữ lại tên gốc của nó mặc dù thậm chí các phần khác đã bị thay đổi. Thú vị nữa là this dates back to a bug trong hàm **LoadTypeLibEx** trong oleaut32.dll in Win95 mà “rsrc” đã sử dụng để tìm kiếm và nạp resource section. Điều này gây ra một lỗi nếu section bị đổi tên. (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)

Bằng việc mở ứng dụng của chúng ta trong LordPE và nhấn vào Compare button chúng ta có thể mở bản gốc của ứng dụng và quan sát sự thay đổi của các headers :

Item To Compare	basecalc.exe	Copy of BASEC...
✗ filesize	16A00	38200
✓ e_lfanew	00000100	00000100
► FileHeader:		
✓ Machine	014C	014C
✗ NumberOfSections	0003	0008
✓ TimeDateStamp	2A425E19	2A425E19
✓ PointerToSymbolTable	00000000	00000000
✓ NumberOfSymbols	00000000	00000000
✓ SizeOfOptionalHeader	00E0	00E0
✗ Characteristics	818F	818E
► OptionalHeader		

Item To Compare	basecalc.exe	Copy of BASEC...
✓ MinorLinkerVersion	19	19
✗ SizeOfCode	00016000	0002A000
✗ SizeOfInitializedData	00002000	0000DE00
✗ SizeOfUninitializedData	00029000	00000000
✗ AddressOfEntryPoint	0003F230	0002ADB4
✗ BaseOfCode	0002A000	00001000
✗ BaseOfData	00040000	0002B000
✓ ImageBase	00400000	00400000
✓ SectionAlignment	00001000	00001000
✓ FileAlignment	00000200	00000200
✓ MajorOperatingSystemVersion	0001	0001

Item To Compare	basecalc.exe	Copy of BASEC...
► DataDirectories:		
✓ ExportTable-RVA	00000000	00000000
✓ ExportTable-Size	00000000	00000000
✗ ImportTable-RVA	00041080	0002D000
✗ ImportTable-Size	00000178	0000181E
✗ Resource-RVA	00040000	00034000
✗ Resource-Size	00001080	00008E00
✓ Exception-RVA	00000000	00000000
✓ Exception-Size	00000000	00000000
✓ Security-RVA	00000000	00000000
✓ Configuration-RVA	00000000	00000000

Khi chúng ta mở ứng dụng trong Olly , chúng ta sẽ nhận được một Message Box thông báo rằng file thực thi của chúng ta đã bị packed. Chỉ việc nhấn Ok và chúng ta sẽ ở tại EntryPoint :

C CPU - main thread, module BASECALC

0043F230	\$ 60	PUSHAD
0043F231	. BE 00A04200	MOV ESI, BASECALC.0042A000
0043F236	. 8DBE 0070FDFF	LEA EDI, DWORD PTR DS:[ESI+FFFD7000]
0043F23C	. C787 D4B30200	MOU DWORD PTR DS:[EDI+2B3D4], 8384888C
0043F246	. 57	PUSH EDI
0043F247	. 83CD FF	OR EBP, FFFFFFFF
0043F24A	↓EB 0E	JMP SHORT BASECALC.0043F25A
0043F24C	90	NOP
0043F24D	90	NOP
0043F24E	90	NOP
0043F24F	90	NOP
0043F250	> 8A06	MOV AL, BYTE PTR DS:[ESI]

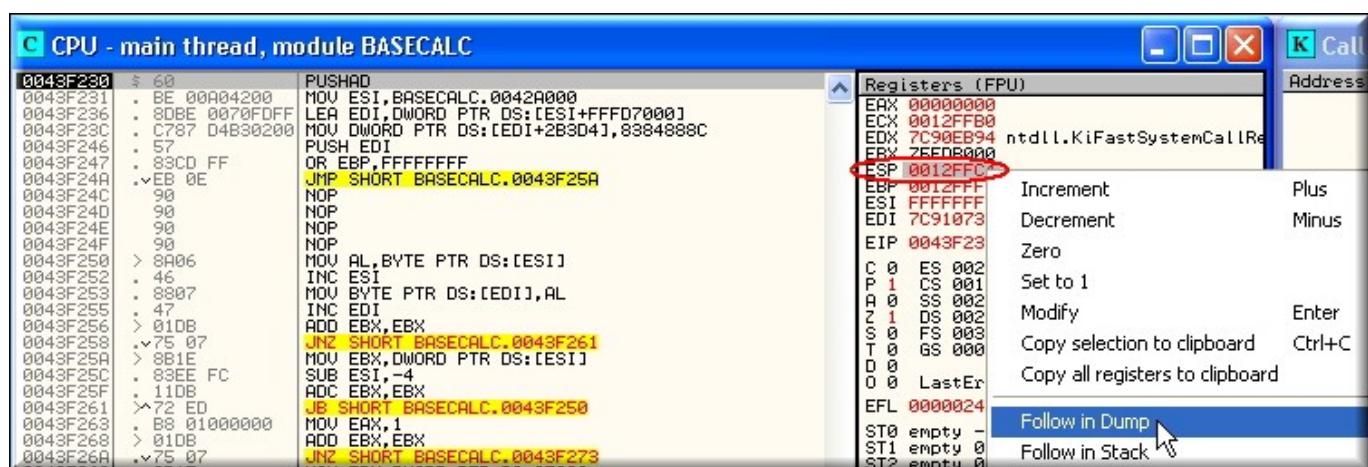
Trình Packer UPX đã nén ứng dụng của chúng ta và đã thêm code bằng một stub có chứa giải thuật để decompressEntryPoint của ứng dụng đã bị thay đổi để bắt đầu đoạn stub và sau đó khi stub thực hiện xong công việc của nó, hướng thực thi của chương trình sẽ nhảy về original entrypoint (OEP) để bắt đầu chương trình bây giờ đã được unpacked của chúng ta.

Lý do căn bản để đối phó với nó là để cho chương trình Sub decompress ứng dụng của chúng ta vào trong bộ nhớ và sau đó dump vùng nhớ này vào một file để có được bản sao của chương trình đã được unpacked. Tuy nhiên ứng dụng sẽ không thực thi theo đúng cách của nó đó là bởi vì file được dumped sẽ có các sections riêng của nó được aligned to memory page boundaries chứ không phải file alignment values, do đó entrypoint sẽ vẫn trỏ tới decompression stub và Import directory rõ ràng là sai và sẽ cần phải chỉnh sửa lại.

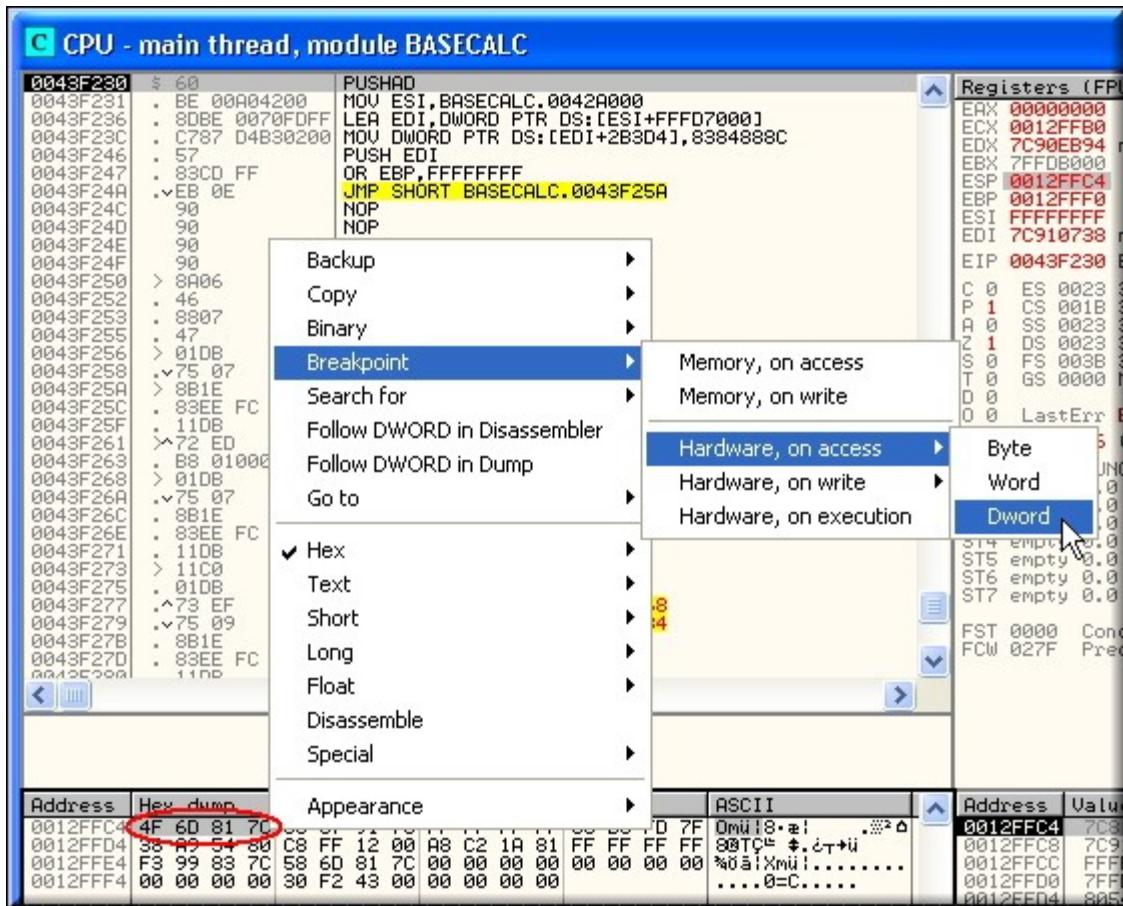
Chú ý rằng trong Olly entrypoint của chúng ta nằm tại câu lệnh đầu tiên là **PUSHAD**. Câu lệnh **PUSHAD** này là viết tắt của **PUSH ALL DOUBLE**, thực hiện việc lưu tất cả nội dung của các thanh ghi 32 bit vào trong Stack, bắt đầu từ EAX cho đến EDI. Theo đó Stub sẽ thực hiện công việc của nó và sau đó kết thúc bằng một câu lệnh **POPAD** trước lệnh nhảy tới OEP. **POPAD** sao chép lại nội dung của các thanh ghi từ Stack. Điều này có nghĩa là stub sẽ phải phục hồi lại mọi thứ và exited without trace trước khi thực sự Run ứng dụng. Vì vậy phương pháp này là ý tưởng cho nhiều packer thông dụng khác ví dụ như ASPack.

Từ thời điểm của câu lệnh **PUSHAD** đầu tiên, những nội dung của Stack tại level đó phải được hoàn toàn không được động tới cho tới khi gặp được câu lệnh **POPAD**. Nếu như chúng ta đặt một Hardware breakpoint lên 4 bytes đầu tiên của stack tại thời điểm thực hiện lệnh **PUSHAD** thì Olly sẽ break tại thời điểm khi mà 4 bytes này được truy cập tại câu lệnh **POPAD** và chúng ta sẽ ở tại đúng câu lệnh nhảy tới OEP của chúng ta.

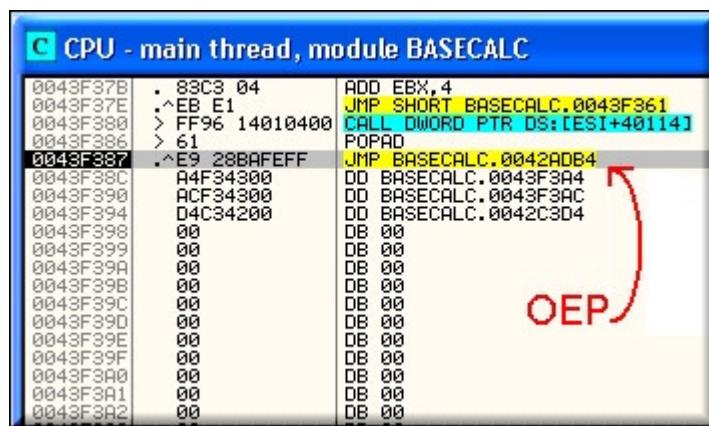
Đầu tiên chúng phải thực hiện câu lệnh **PUSHAD** bằng cách nhấn F7 một lần. Tiếp theo chúng ta sẽ đặt một BP của chúng ta. Thanh ghi ESP (Stack Pointer) luôn luôn trỏ tới của đỉnh Stack do đó Right click lên ESP và chọn Follow in Dump – Chúng ta sẽ có được như sau :



Tiếp theo Highlight DWORD đầu tiên của Stack trong cửa sổ Dump, chuột phải và chọn BP>HardWare on Access>DWORD:



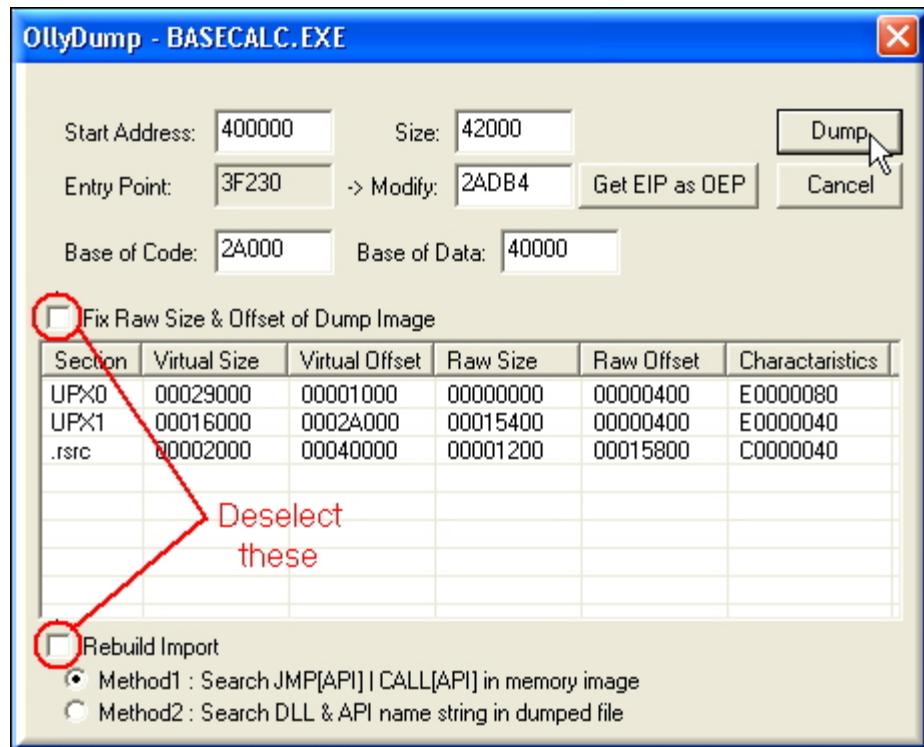
Tiếp theo nhấn **F9** để Run chương trình và Olly sẽ Break. Chúng ta quan sát sẽ thấy có lệnh JMP tới OEP. OEP mà chúng ta thấy ở đây có ImageBase là 400000h được cộng thêm vào , do đó chúng ta muốn tìm thấy Real OEP thì chúng ta phải trừ đi giá trị ImageBase ở trên. Cho nên ta có OEP là : 0002ADB4h.



Nếu như bạn muốn gian lận ở đây có một cách nhanh chóng mà luôn luôn có hiệu quả với UPX. Đơn giản chỉ là bạn cuộn chuột tới phía cuối của đoạn code trong màn hình CPU trong Olly và phía trước tất cả chỗ bắt đầu của zero padding thì bạn sẽ thấy được câu lệnh POPAD như ở trên.

NOTE: Các Packer khác mà cũng sử dụng cơ chế **PUSHAD/POPAD** để có thể nhảy tới OEP bằng cách sử dụng một lệnh PUSH để đẩy giá trị của OEP lên trên đỉnh của Stack được followed bởi câu lệnh RET. CPU sẽ nghĩ là đây là một return từ một hàm call và theo thói quen thì địa chỉ trả về được đặt lên đỉnh của Stack.

Bước tiếp theo chúng ta nhấn F7 để thực hiện lệnh JMP và chúng ta sẽ ở tại OEP. Tại đây chúng ta sẽ sử dụng Plugin của Olly là OllyDump để Dump file này. Chuột phải tại OEP sau đó chọn OllyDump, chúng ta sẽ có được màn hình như sau , thực hiện như hình minh họa :

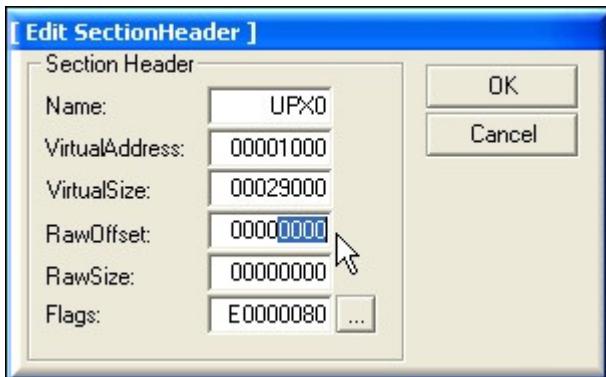


Note that OllyDump has already worked out the base address and size of image (which you could see by looking in the memory map window) and has offered to correct the entrypoint for us (although we could do this manually in the hexeditor). Nhấn Dump và save file với tên nào đó mà bạn muốn (eg as basecalc_dmp.exe). Giữ nguyên trạng thái của Olly sau khi đã thực hiện Dump.

Thật không may mắn khi chúng ta quan sát file được dump thì thấy nó bị mất icon và nếu như chúng ta cố tình Run file thì chúng ta sẽ nhận được thông báo như sau :



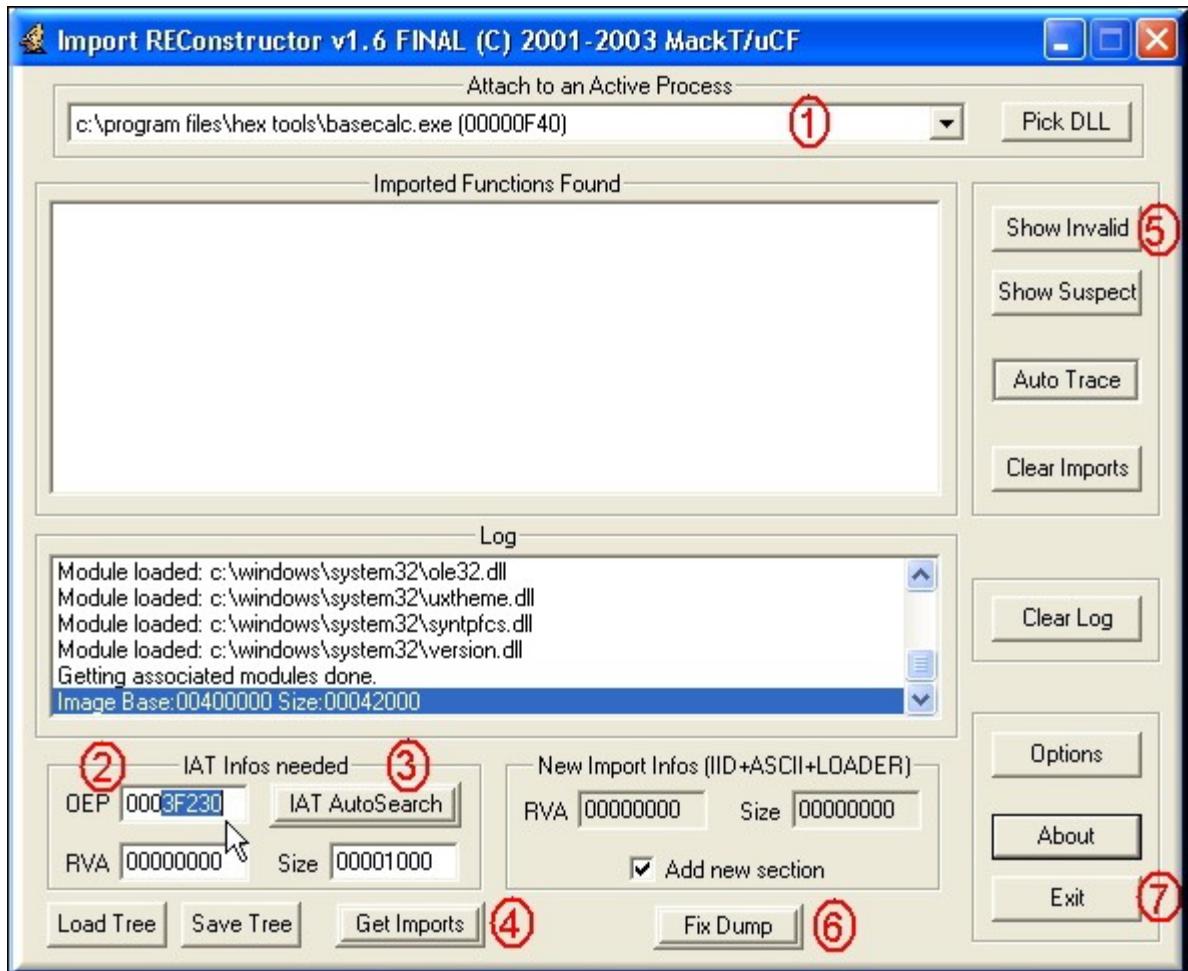
Chúng ta nhận được thông báo trên là bởi vì hậu quả của vấn đề alignment mà tôi đã đề cập ở trên – kích thước của file cũng đã tăng. Chúng ta mở app của chúng ta trong LordPE và quan sát tại các Sections. Các giá trị Raw offset và Raw Size đã sai. Chúng ta sẽ phải tạo các giá trị Raw bằng với giá trị các Virtual cho mỗi Section cho ứng dụng của chúng ta để cho nó hoạt động. Nhấn chuột phải tại UPX0 section và chọn edit header:



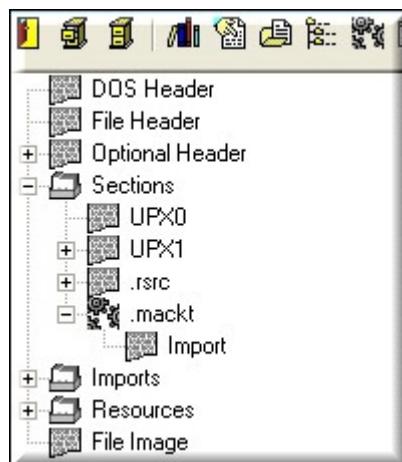
Bây giờ chúng ta sẽ làm cho RawOffset bằng VirtualAddress và RawSize bằng VirtualSize. Lặp lại thao tác này cho mỗi Sections sau đó nhấn Save và Exit (this is what the "fix raw size" checkbox in OllyDump does automatically). Bây giờ chúng ta quan sát thấy app đã có icon nhưng khi chạy ứng dụng của chúng ta , ta sẽ nhận được một lỗi khác là : "The application failed to initialize properly". Có lỗi này là bởi vì chúng ta chưa fix imports.

Việc Fix imports này chúng ta hoàn toàn có thể thực hiện được bằng tay . Tuy nhiên sẽ tốn rất nhiều thời gian và công sức nếu như chúng ta có nhiều hàm được imported v..v. Do đó ở đây chúng ta sẽ sử dụng chương trình ImpREC 1.6F by MackT để thực hiện một cách tự động. Chương trình ImpREC cần phải attach tới một process đang chạy và cũng cần packed file để tìm imports. Khởi động ImpREC và thực hiện theo các bước sau :

1. Chọn Basecalc.exe trong danh sách Attach (it should still be running in Olly).
2. Tiếp theo nhập OEP của chúng ta là 2ADB4 vào trong textbox OEP.
3. Nhấn nút “IAT AutoSearch” và nhấn OK trên messagebox.
4. Nhấn nút “Get Imports”.
5. Nhấn “Show Invalid” – trong trường hợp của chúng ta không có invalid nào.
6. Nhấn “Fix Dump” và chọn file mà chúng ta đã dump là basecalc_dmp.exe.
7. OkieThoát khỏi ImpREC.

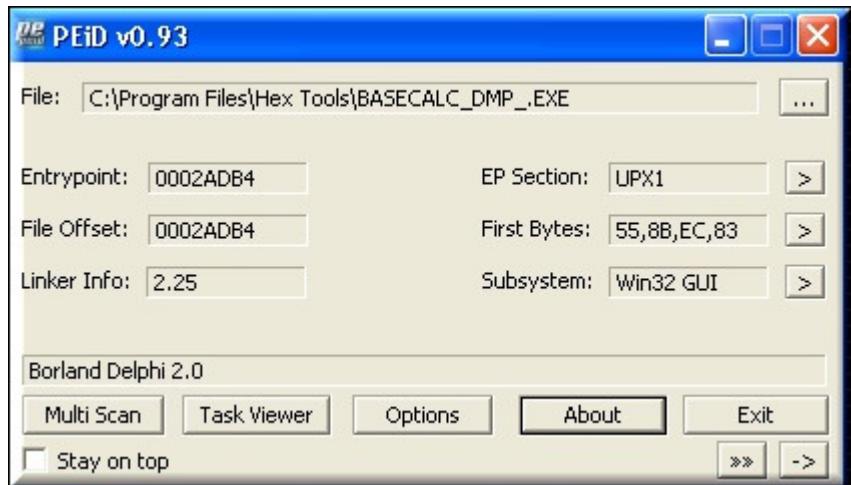


Chương trình ImpREC sẽ lưu file đã fix với tên như sau : **basecalc_dmp_.exe**. Chúng ta chạy thử file này để kiểm tra. Nếu như chúng ta phân tích file này chúng ta sẽ thấy kích thước của nó đã tăng lên và có thêm một section nữa có tên là “mackt” – đó là nơi mà ImpREC đưa import data mới :



Vì UPX chỉ là một chương trình nén, nó đơn giản chỉ là lấy existing import data và lưu nó lại trong resource section mà không encrypting or damaging it. Đó là lý do tại sao ImpREC có thể tìm được tất cả các valid imports mà không cần phải resorting to tracing or rebuilding – nó chỉ lấy import directory từ file thực thi đã bị packed trong bộ nhớ và transfer nó tới section mới trong file thực thi đã được unpacked.

Giờ chúng ta hãy Scan file đã được unpacked trong PEID xem :



Trên đây chỉ là phần minh họa các bước cần thiết cho việc thực hiện unpack một file thực thi đã bị packed bằng một packer đơn giản. Tuy nhiên có rất nhiều các packers cao cấp mà các packer này thêm rất nhiều các cơ chế bảo vệ khác nhau ví dụ như : antidebugging và anti-tampering tricks, encryption of code và IAT, stolen bytes, API redirection, etc mà trong phạm vi của bài viết này tôi không thể đề cập hết được, mong các bạn bỏ qua cho 😊.

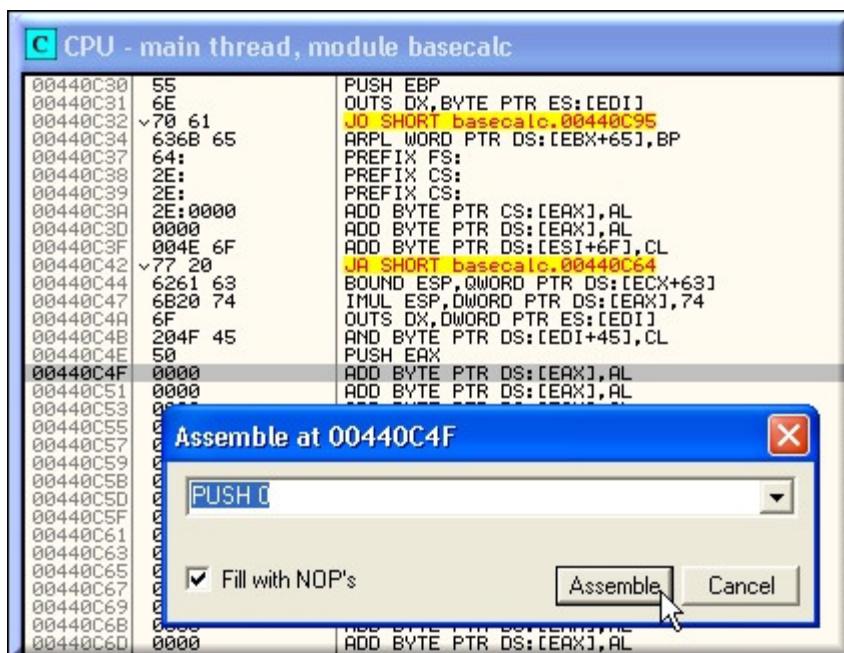
Trong một số trường hợp nếu như cần thiết chúng ta phải Patch một file đã bị packed , điều này giúp chúng ta có thể tránh được việc không cần phải unpacking file thì có một kĩ thuật được sử dụng đó là **“inline – patching”**. Nó liên quan đến việc patching code tại thời điểm runtime trong bộ nhớ sau khi quá trình decompression stub đã hoàn thành xong công việc của mình và cuối cùng nhảy tới OEP để thực thi ứng dụng. Nói cách khác ,chúng ta đợi cho đến khi ứng dụng của chúng ta đã được unpacked trong bộ nhớ , thì nhảy tới patching code mà chúng ta đã injected, cuối cùng sau đó nhảy trở về OEP.

Để minh họa cho kỹ thuật này chúng ta sẽ inject code vào trong file thực thi đã bị packed của chúng ta để bắn ra một thông báo và cho chúng ta biết khi ứng dụng đã được unpacked trong bộ nhớ. Sau đó khi chúng ta nhấn OK thì sẽ nhảy tới OEP và ứng dụng sẽ thực thi một cách bình thường.

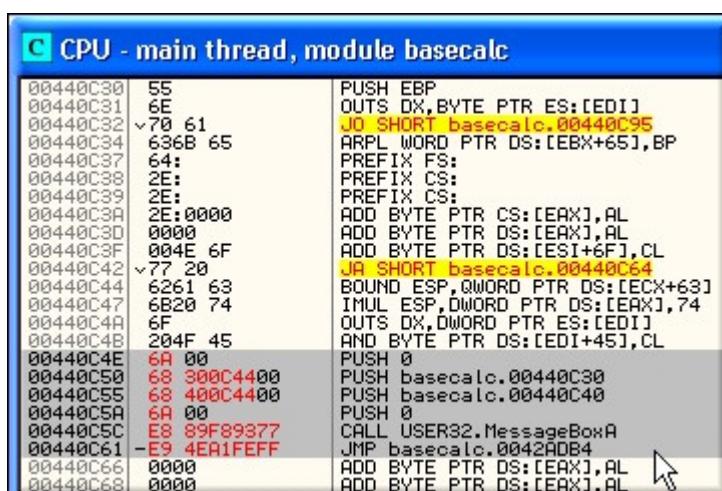
Nhiệm vụ đầu tiên là chúng ta phải tìm kiếm một nơi cho đoạn code của chúng ta vì vậy hãy mở packed app vào trong trình Hexeditor và tìm kiếm một khoảng không gian phù hợp còn gọi là suitable "cave". Khoảng không gian trống này nằm tại phía cuối của section là tốt hơn cả bởi vì nó ít được sử dụng bởi packer và có thể mở rộng được bằng cách nới rộng section nếu cần thiết (Xin xem lại phần **adding code to a PE file**.) Bạn có thể quan sát thấy hiệu quả của Packer UPX – khoảng không gian chúng ta cần là rất khó – tuy nhiên vẫn có một khoảng nhỏ (small cave) tồn tại ở đây.Bây giờ chúng ta thêm "Unpacked..." và "Now back to OEP" trong ASCII column của chương trình HexEditor. Tương tự như hình minh họa dưới đây :

Điều này sẽ đánh dấu dấu vết của chúng ta để patch trong Olly mà không cần phải lo lắng về việc tính toán các VAs. Lưu lại những thay đổi và mở ứng dụng của chúng ta trong Olly. Chuột phải tại cửa sổ Hex window và chọn search for binary string. Bây giờ nhập vào là "Unpacked" và để ý tới VA của 2 strings. Trong cửa sổ CPU Window, nhấn chuột phải và chọn Goto expression. Nhập địa chỉ của string đầu tiên và bạn sẽ quan sát 2 strings trong hexadecimal form. Olly đã không analysed nó một cách đúng đắn do đó nó hiện thị không ra thành một đoạn code không có ý nghĩa gì. Highlight đoạn code (the next free row underneath) và nhấn Space Bar để 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. Đoạn code của chúng ta sẽ trông như sau trong Olly :



Tiếp theo chuột phải và chọn **copy to executable, selection**. Trong cửa sổ mới xuất hiện , rightclick và chọn 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	ä	þ	ç	đ	é	í
00016420h:	77	77	77	77	00	00	00	00	00	00	00	00	00	00	00	00 ; www.....
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	; tø^wéK;þý.....
00016470h:	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	;
00016490h:	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	20	00	00	00	; .. OS...t.....

Cuối cùng chúng ta cần phải thay đổi lệnh JMP tại phía cuối của UPX stub để nhảy tới đoạn code của chúng ta. Tìm lệnh nhảy này như đã đề cập ở phần trên, doubleclick vào JMP instruction để assemble và thay đổi address thành 440C4E. Lưu lại thay đổi một lần nữa và run app của chúng ta để test :



Clicking OK resumes BaseCalc.!!!!!!!!

14. 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).

15. Complete PE Offset Reference :

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 :

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	
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 :

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 :

OFFSET	SIZE	NAME	EXPLANATION
00	DWORD	OriginalFirstThunk	RVA to Image_Thunk_Data
04	DWORD	TimeStamp	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

16. Relative Virtual Addressing Explained :

Trong một file thực thi hay một file DLL, thì **RVA** luôn luôn là địa chỉ của một item khi được nạp vào trong bộ nhớ, với base address (**ImageBase**) của imaging file được trừ đi : **RVA = VA – ImageBase** do đó **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.

Ví dụ : Nếu một PE file nạp tại địa chỉ 400000h trong virtual address (VA) space và chương trình bắt đầu thực thi tại virtual address 401000h, chúng ta có thể nói rằng chương trình bắt đầu thực thi tại RVA 1000h. A VA 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

Thuật ngữ "Virtual Address" được sử dụng bởi vì Windows tạo ra một không gian địa chỉ ảo riêng biệt cho mỗi process, không lệ thuộc vào physical memory. Cho hầu hết tất cả các mục đích, một virtual address được xem xét chỉ là một address. Như nói ở trên, một virtual address là không thể dự đoán trước được như một RVA, bởi vì trình loader không thể load the image at its preferred base address.

Tại sao PE file format lại sử dụng RVA? Đó là để làm giảm bớt quá trình nạp của trình loader. Đó là bởi vì nếu một module có thể được relocated bất kỳ vị trí nào trong không gian địa chỉ ảo, nó sẽ gây trở ngại cho trình loader để fix mọi hardcoded address trong module. Nhưng ngược lại, nếu tất cả relocatable items trong file use RVA, there is no need for the loader to fix anything: nó chỉ đơn giản relocates toàn bộ moduel tới một new starting VA.

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

Chuyển đổi các raw offsets (the one in a file you see in a HexEditor) thành virtual offsets (the one you see in a debugger) là cực kỳ hữu ích nếu như bạn làm việc với PE Header. Chính vì lý do này bạn cần phải biết một vài giá trị từ PE Header. Bạn cần phải biết **ImageBase**, the name of the section in which your offset lies. Dưới đây bạn sẽ xem một ví dụ của một PE Header từ điểm bắt đầu của file (where it is actually a MZ header until offset 80h) cho tới phần cuối định nghĩa các sections (offset 23Fh). Ví dụ này được minh họa bằng chương trình **notepad.exe**.

```

000000000 4D5A 9000 0300 0000 0400 0000 FFFF 0000 MZ.....
000000010 B800 0000 0000 0000 4000 0000 0000 0000 .....@....
000000020 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000030 0000 0000 0000 0000 0000 0000 8000 0000 .....
000000040 0E1F BAOE 00B4 09CD 21B8 014C CD21 5468 .....!..L.!Th
000000050 6973 2070 726F 6772 616D 2063 616E 6E6F is program canno
000000060 7420 6265 2072 756E 2069 6E20 444F 5320 t be run in DOS
000000070 6D6F 6465 2EOD ODOA 2400 0000 0000 0000 mode....$....
000000080 5045 0000 4C01 0500 6591 4635 0000 0000 PE..L...e.F5...
000000090 0000 0000 E000 0E01 0B01 030A 0040 0000 .....@...
0000000A0 0072 0000 0000 0000 CC10 0000 0010 0000 .r.....
0000000B0 0050 0000 0000 4000 0010 0000 0010 0000 .P....@...
0000000C0 0400 0000 0000 0000 0400 0000 0000 0000 .....
0000000D0 00E0 0000 0004 0000 D509 0100 0200 0000 .....
0000000E0 0000 1000 0010 0000 0000 1000 0010 0000 .....
0000000F0 0000 0000 1000 0000 0000 0000 0000 0000 .....
000000100 0060 0000 8C00 0000 0070 0000 E453 0000 .`.....p...S..
000000110 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000120 00D0 0000 3C09 0000 0000 0000 0000 0000 ....<...
000000130 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000140 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000150 0000 0000 0000 0000 E062 0000 4002 0000 .....b..@...
000000160 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000170 0000 0000 0000 0000 2E74 6578 7400 0000 .....text...
000000180 9C3E 0000 0010 0000 0040 0000 0010 0000 .>.....@....
000000190 0000 0000 0000 0000 0000 0000 2000 0060 .....
0000001A0 2E64 6174 6100 0000 4C08 0000 0050 0000 .data...L...P..
0000001B0 0010 0000 0050 0000 0000 0000 0000 0000 .....P...
0000001C0 0000 0000 4000 00C0 2E69 6461 7461 0000 .....@...idata..
0000001D0 E80D 0000 0060 0000 0010 0000 0060 0000 .....`...
0000001E0 0000 0000 0000 0000 0000 0000 4000 0040 .....@..0
0000001F0 2E72 7372 6300 0000 0060 0000 0070 0000 .rsrc...`...p..
000000200 0060 0000 0070 0000 0000 0000 0000 0000 .`...p.....
000000210 0000 0000 4000 0040 2E72 656C 6F63 0000 .....@..0.reloc..
000000220 9C0A 0000 00D0 0000 0010 0000 00D0 0000 ......
000000230 0000 0000 0000 0000 0000 0000 4000 0042 .....@..B

```

Ví dụ 1 - Converting raw offset 7800h to a virtual offset:

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

```

000000170 2E74 6578 7400 0000 .....text...
000000180 9C3E 0000 0010 0000 0040 0000 0010 0000 .>.....@....
000000190 0000 0000 0000 0000 0000 0000 2000 0060 .....
0000001A0 2E64 6174 6100 0000 4C08 0000 0050 0000 .data...L...P..
0000001B0 0010 0000 0050 0000 0000 0000 0000 0000 .....P...
0000001C0 0000 0000 4000 00C0 2E69 6461 7461 0000 .....@...idata..
0000001D0 E80D 0000 0060 0000 0010 0000 0060 0000 .....`...
0000001E0 0000 0000 0000 0000 0000 0000 4000 0040 .....@..0
0000001F0 2E72 7372 6300 0000 0060 0000 0070 0000 .rsrc...`...p..
000000200 0060 0000 0070 0000 0000 0000 0000 0000 .`...p.....
000000210 0000 0000 4000 0040 2E72 656C 6F63 0000 .....@..0.reloc..
000000220 9C0A 0000 00D0 0000 0010 0000 00D0 0000 ......
000000230 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** và các giá trị được đánh màu da cam trong output của trình Hexeditor ở trên là không quan trọng đối với quá trình chuyển đổi nhưng sẽ có những chức năng khác(see Section Table page).

Chúng ta muốn convert raw offset 7800h. Điều này dường như là rõ ràng bởi offset này nằm trong **.rsrc section** bởi vì **.rsrc** bắt đầu tại 7000h (Raw Offset) và 6000h bytes long (Raw Size). Offset 7800h is located 800h bytes sau chỗ bắt đầu của section trong file. Vì tất cả các sections đã được copy vào trong 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.

Công thức chung là :

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

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

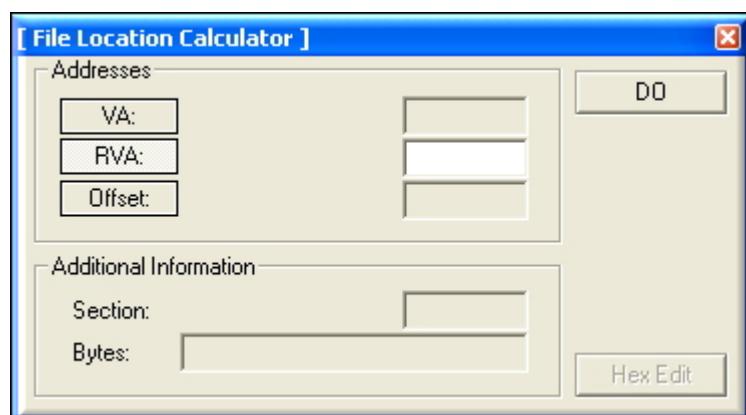
The conversion from a virtual offset to a raw offset just goes the other way round. The general formula is:

$$\text{Raw Offset} = \text{RVA_YouHave} - \text{ImageBase} - \text{VirtualOffsetOfSection} + \text{RawOffsetOfSection}$$

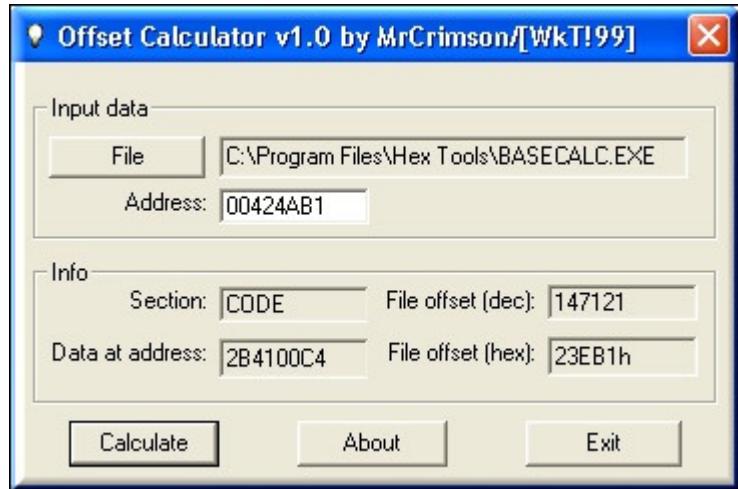
For 40A000 that is: $40A000 - 400000 - 7000 + 7000 = A000$

There are also automated tools to perform the above conversions. Pressing the "FLC" button on the PE

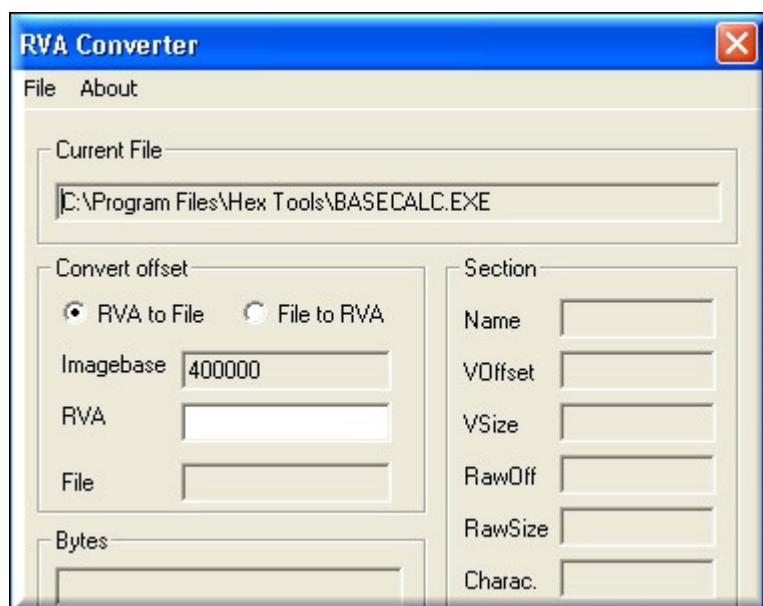
Trình Editor của LordPE sẽ cho phép chúng ta convert an RVA to an offset:



Chương trình **Offset Calculator** cũng cho phép một conversion one-way from RVA to Raw Offset.



Chuong trinh **RVA Calculator** cho phép onversion **both ways**:



...:[The End]:...



03/08/2005

--+---=[Greatz Thanks To]=---++-

- Thank to my family, Computer_Angel, Moonbaby , Zombie_Deathman, Littleboy, Benina, QHQCk, the_Lighthouse, Hoadongnoi, Nini ... all REA's members, HacNho,RongChauA,Deux, tlandn, dqtl, ARTEAM (especially Goppit) all my friend, and YOU.

--+---=[Special Thanks To]=---++-

- coruso_trac, patmsvn, trm_tr v..v.. and all brothers in VSEC

>>> If you have any suggestions, comments or corrections email me: [kienbigmummy\[at\]gmail.com](mailto:kienbigmummy[at]gmail.com)

