Understanding the Import Address Table

Import Libraries are dlls that an executable image are bound to. Much of windows core functionailty is found in Dlls that MS provides and is how applications interact with the base windows services.

Function addresses in the binary file of a dll are not static, as new versions come out they are destined to change, so applications cannot be built using a hardcoded function address.

When an executable is first loaded, the Windows loader is responsible for reading in the files PE structure and loading the executable image into memory. One of the other steps it takes is to load all of the dlls that the application uses and map them into the process address space.

The executable also lists all of the functions it will require from each dll. Because the function addresses are not static a mechanism had to be developed that allowed for the these variables to be changed without needing to alter all of the compiled code at runtime.

This was accomplished through the use of an import address table. This is a table of function pointers filled in by the windows loader as the dlls are loaded.

When the application was first compiled, it was designed so that all API calls will not use direct hardcoded addresses but rather work through a function pointer. Conventially this pointer table can be accessed in several ways. Either directly by a call[pointer address] or via a jmp thunk table. Below are examples of each

```
Jmp Thunk Table:


     ...inline app code...
     00401002  |. E8 7B0D0000    CALL 00401D82               ; \GetModuleHandleA
     ...thunk table...
     00401D82   $-FF25 4C204000 , JMP DWORD PTR DS:[40204C]  ;  KERNEL32.GetModuleHandleA
     ...memory address value of pointer...
     40204C > FC 3D 57 7C   ;little endian pointer value



JMP DWORD PTR DS:[40204C] is the same as jmp 7C573DFC which is GetModuleHandleA fx address.
that was filled into this address by the windows loader.

The pointer values can also be used directly in the compiled code without the use of a thunk
table:


     0040103A  |. FF15 7A204000  CALL DWORD PTR DS:[40207A]               ; \MessageBoxA
```

By using the pointer table, the loader does not need to fixup all of the places in the code that want to use the api call, all it has to do is add the pointer to a single place in a table

and its work is done.

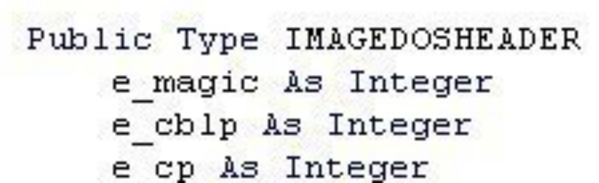
When it comes to packed executables, they almost invariably mess with the processes import table in order to make the executable smaller and make it harder for people to unpack and get running again.

Packed programs were still generated with standard compilers, and of course they were still designed to work this fixed mechanism (which is a very efficient way to handle the problem anyway)

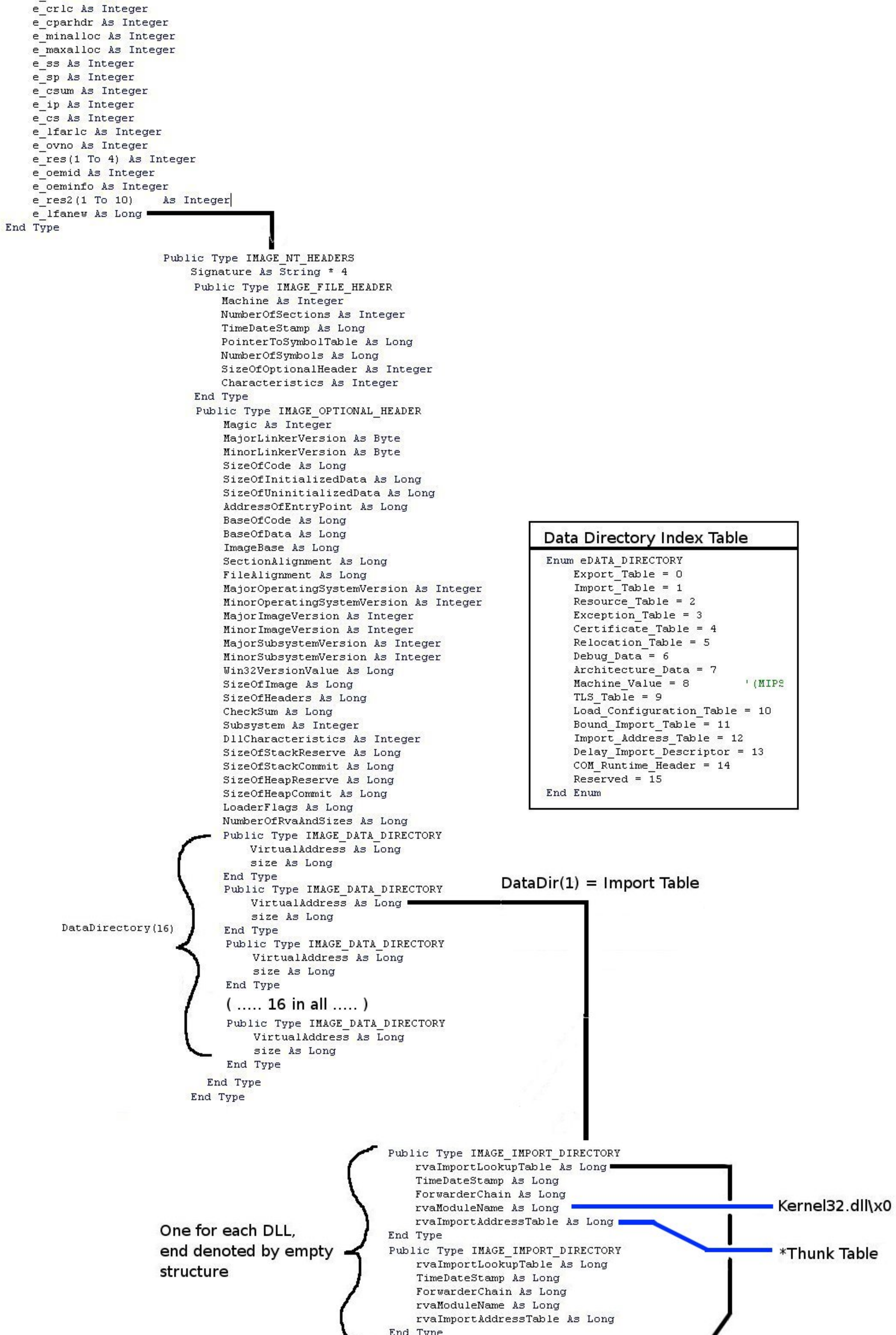
If a packer has destroyed the default import table mechanism, that simply means that it has agreed to take it upon themselves to figure out which dlls and functions to load and where to place the pointers so that the original program still operates as normal after it has done its decompression and restoration routines.

In order to understand how to restore a missing import table we will first have to start by understanding how the Import table is laid out and what work the Windows loader must do to parse it in the first place.

To understand this, the first thing you need to digest is the PE file format and how Import table information is held in the binary. I have created the following jpg image of the nested structures with annotations to help visualize it. It is a very complex structure and there is allot of stuff going on. I have only included the base information needed for understanding the import table in it. Structure definitions are in VB with a C style nesting for clarity.



```
Public Type IMAGEDOSHEADER
    e_magic As Integer
    e_cblp As Integer
    e_cp As Integer
```


DataDirectory(16)
(..... 16 in all .....)
DataDir(1) = Import Table
One for each DLL, end denoted by empty structure
Kernel32.dll\x0
*Thunk Table

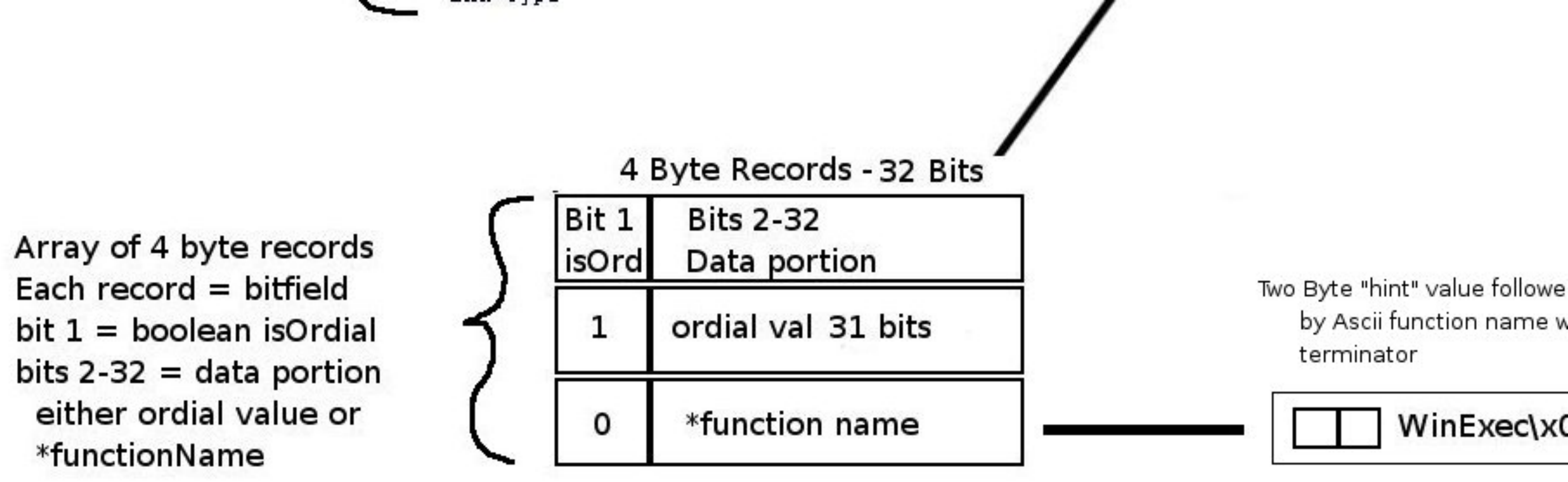
```
    e_crlc As Integer
    e_cparhdr As Integer
    e_minalloc As Integer
    e_maxalloc As Integer
    e_ss As Integer
    e_sp As Integer
    e_csum As Integer
    e_ip As Integer
    e_cs As Integer
    e_lfarlc As Integer
    e_ovno As Integer
    e_res(1 To 4) As Integer
    e_oemid As Integer
    e_oeminfo As Integer
    e_res2(1 To 10)    As Integer
    e_lfanew As Long
End Type
```

```
Public Type IMAGE_NT_HEADERS
    Signature As String * 4
    Public Type IMAGE_FILE_HEADER
        Machine As Integer
        NumberOfSections As Integer
        TimeDateStamp As Long
        PointerToSymbolTable As Long
        NumberOfSymbols As Long
        SizeOfOptionalHeader As Integer
        Characteristics As Integer
    End Type
    Public Type IMAGE_OPTIONAL_HEADER
        Magic As Integer
        MajorLinkerVersion As Byte
        MinorLinkerVersion As Byte
        SizeOfCode As Long
        SizeOfInitializedData As Long
        SizeOfUninitializedData As Long
        AddressOfEntryPoint As Long
        BaseOfCode As Long
        BaseOfData As Long
        ImageBase As Long
        SectionAlignment As Long
        FileAlignment As Long
        MajorOperatingSystemVersion As Integer
        MinorOperatingSystemVersion As Integer
        MajorImageVersion As Integer
        MinorImageVersion As Integer
        MajorSubsystemVersion As Integer
        MinorSubsystemVersion As Integer
        Win32VersionValue As Long
        SizeOfImage As Long
        SizeOfHeaders As Long
        CheckSum As Long
        Subsystem As Integer
        DllCharacteristics As Integer
        SizeOfStackReserve As Long
        SizeOfStackCommit As Long
        SizeOfHeapReserve As Long
        SizeOfHeapCommit As Long
        LoaderFlags As Long
        NumberOfRvaAndSizes As Long
        Public Type IMAGE_DATA_DIRECTORY
            VirtualAddress As Long
            size As Long
        End Type
        Public Type IMAGE_DATA_DIRECTORY
            VirtualAddress As Long
            size As Long
        End Type
        Public Type IMAGE_DATA_DIRECTORY
            VirtualAddress As Long
            size As Long
        End Type
        ( ..... 16 in all ..... )
        Public Type IMAGE_DATA_DIRECTORY
            VirtualAddress As Long
            size As Long
        End Type
    End Type
End Type
```

Data Directory Index Table

```
Enum eDATA_DIRECTORY
    Export_Table = 0
    Import_Table = 1
    Resource_Table = 2
    Exception_Table = 3
    Certificate_Table = 4
    Relocation_Table = 5
    Debug_Data = 6
    Architecture_Data = 7
    Machine_Value = 8        '(MIPS
    TLS_Table = 9
    Load_Configuration_Table = 10
    Bound_Import_Table = 11
    Import_Address_Table = 12
    Delay_Import_Descriptor = 13
    COM_Runtime_Header = 14
    Reserved = 15
End Enum
```

```
Public Type IMAGE_IMPORT_DIRECTORY
    rvaImportLookupTable As Long
    TimeDateStamp As Long
    ForwarderChain As Long
    rvaModuleName As Long
    rvaImportAddressTable As Long
End Type
Public Type IMAGE_IMPORT_DIRECTORY
    rvaImportLookupTable As Long
    TimeDateStamp As Long
    ForwarderChain As Long
    rvaModuleName As Long
    rvaImportAddressTable As Long
End Type
```



That graphic should explain the structure and nesting adequetly. If you need more of a reference search the msdn site for the full PE specification. They have a very comprehensive pdf on it although it is quite dry.

If you are to write any kind of import restoration tool you will have to understand it fully. To drill it into your mind, I would recommend finding a small target exe and opening it up in LordPE and a hexeditor at the same time. Use the hexeditor to navigate the structures and LordPE as reference on field values.

Even after all of that and writing a PE editor utility myself i still felt like i had to create that graphic in order to get a full sense of the nesting and layout. Dont worry it just takes time.

So, Lets say we have a packed executable... (with a packer that does not use API redirection for simplicity sake)

What information do we need to restore the import table of the app? To answer just this question i have put together a small download package consisting a very simple exe with only two imports. This file was compiled with fasm and simply displays a messagebox and then exits.

First thing we do is load it up in olly and search for all intermodular calls, we come up with the following:

```
0040103A   CALL DWORD PTR DS:[40207A]                 USER32.MessageBoxA

00401042   CALL DWORD PTR DS:[40205C]                 KERNEL32.ExitProcess
```

From this we really have everything that we need to build the import table from scratch. We know that the pointer for USER32.MessageBoxA must sit at offset 40207A and the one for KERNEL32.ExitProcess must be found at 40205C for this app to work properly. This seems simple enough, the only trick being that it is a rather complex structure to create to get the Windows loader to find all of the strings and load the dlls properly.

Lets now look at a packed version of this same app to see what the packer changed. The packer we used for this example was telock without using the API redirection option. The easiest way in, is to let the telocked sample run until it shows the messagebox and then attach to it. By this time, all debugger checks have already run and you know that it is paused in the main applications code. Once inside, you will be paused in ntdll, press ctrl-g and enter 401000 to jump to the main applications code. Now you can search for intermodular calls again and see all of the apis and how they are called:

```
0040103A   CALL DWORD PTR DS:[40207A]                 user32.MessageBoxA
00401042   CALL DWORD PTR DS:[40205C]                 kernel32.ExitProcess
```

This is exactly as it was in the original exe. If you open up the raw dump file provided in LordPE and list the Import table you will find that telock wiped the structure andit is not present. What we need now is a way to reconstruct the complex structure of the import table in the binary so that these dlls are loaded by the windows loader and the pointer values to the functions appear at the right offsets the linker was originally expecting.

Of course the best way to do this is to use that graphic i created and the PE specification from MS and manually recreate the structures in your trusty hex editor. Even for such a simple exe it is a lengthy process but trust me it is worth it and this is one of the simplest exes you will find as a trainer so give it a shot.

Seeing how manually rebuilding the tables in a hexeditor is to lengthy of a process to do manually it is time to seek out some automation to help you rebuild the import table. For this I create a simple tool that allows you to build an import table from scratch and control all aspects of it. There are some other automated tools out there for this task but I wanted to try to design my own. As a bonus, I am also releasing it open source so that you can see how it was done and modify it to suit your needs.

In the download package linked below you will find all exes mentioned throughout this paper as well as the impreb tool and source. The readme in the package will take you through the steps on how to use the tool to rewrite the import table of the sample dumped exe. This tool should also come in handy if you wanted to hack new functionality into a compiled exe. It should be able to be used similar to iidKing in this respect to add new dlls compiled exe. It should be able to be used similar to ilinking in this respect to add new dlls to the import table. Note this is a very bare bones first shot implementation of a tool for this purpose. There is probably more stuff to come as i get time.

A couple last notes i should probably throw in just for reference. This paper is primarily about understanding the import table layout and rewriting it from scratch. I did not highlight all of the steps necessary to take a memory dump and fix it up, that just provided a nice example for something that needed its import table rebuilt.

The last thing worth mentioning is that some packers use what is termed api redirection techniques. From the mechanism outlined above you now know that the linker only cared that a Call[ptr] value eventually leads to the right api. If the packer wants to hide the actual apis from you...all it has to replace that pointer so that the call leads execution into its own little messed up thunk which after some junk operations and math will usally decode the real pointer and jump there. This is an interesting topic so I will include a sample of telock api redirection mechanism below:

```
00740016  -FF20             JMP DWORD PTR DS:[EAX]                   ; user32.MessageBoxA
00740018   90               NOP

0040103A   FF15 7A204000    CALL DWORD PTR DS:[40207A]  ;40207A used to hold normal
                                                        ; pointer to Messageboxa
                                                        ;now jumps to 740000
00740000   EB 02            JMP SHORT 00740004
00740002   CD 20            INT 20
00740004   EB 02            JMP SHORT 00740008
00740006                    db B8                   ;anti-disasm junk
00740008   3D 8500FF20      CMP EAX,20FF0085        ;\
0074000D   B8 735ECF00      MOV EAX,0CF5E73         ; \__anti-heuristic junk ?
00740012   98               CWDE                    ; /
00740013   8BC5             MOV EAX,EBP             ;/
00740015   B8 21007400      MOV EAX,740025          ;address real pointer was moved to
0074001A   FF20             JMP DWORD PTR DS:[EAX]  ;DS:[00740025]=77E36544 (user32.MessageBoxA)
0074001C   90               NOP                         ; \
0074001D   B8 20000000      MOV EAX,20                  ; /--- junk
00740022   0000             ADD BYTE PTR DS:[EAX],AL    ;-
00740024   004465 E3        ADD BYTE PTR SS:[EBP-1D],AL ;\_ actual pointer val not opcodes
00740028   77 00            JA SHORT 00740026           ;/    cant disasm
```

Anyway...it will probably some time to wrap your brain around it all..but its fun stuff so just play with it :P

Downloads

-dzzie [dzzie@yahoo.com]