Tutorial 6: Import Table

We will learn about import table in this tutorial. Let me warn you first. This tutorial is a long and difficult one for those who aren't familiar with the import table. You may need to read this tutorial several times and may even have to examine the related structures under a debugger.

Download **the example**.

Theory:

First of all, you should know what an import function is. An import function is a function that is not in the caller's module but is called by the module, thus the name "import". The import functions actually reside in one or more DLLs. Only the information about the functions is kept in the caller's module. That information includes the function names and the names of the DLLs in which they reside.
Now how can we find out where in the PE file the information is kept? We must turn to **the data directory** for the answer. I'll refresh your memory a bit. Below is the PE header:

```
IMAGE_NT_HEADERS STRUCT
   Signature dd ?
   FileHeader IMAGE_FILE_HEADER <>
   OptionalHeader IMAGE_OPTIONAL_HEADER <>
IMAGE_NT_HEADERS ENDS
```

The last member of the optional header is the data directory:

```
IMAGE_OPTIONAL_HEADER32 STRUCT
   ....
   LoaderFlags dd ?
   NumberOfRvaAndSizes dd ?
   DataDirectory IMAGE_DATA_DIRECTORY 16 dup(<>)
IMAGE_OPTIONAL_HEADER32 ENDS
```

The data directory is an array of **IMAGE_DATA_DIRECTORY** structure. A total of 16 members. If you remember the section table as the root directory of the sections in a PE file, you should also think of the data directory as the root directory of the logical components stored inside those sections. To be precise, the data directory contains the locations and sizes of the important data structures in the PE file. Each member contains information about an important data structure.

Member	Info inside
0	Export symbols
1	Import symbols
2	Resources
3	Exception
4	Security
5	Base relocation
6	Debug
7	Copyright string
8	Unknown
9	Thread local storage (TLS)
10	Load configuration
11	Bound Import
12	Import Address Table
13	Delay Import

14	COM descriptor

Only the members painted in gold are known to me. Now that you know what each member of the data directory contains, we can learn about the member in detail. Each member of the data directory is a structure called **IMAGE_DATA_DIRECTORY** which has the following definition:

```
IMAGE_DATA_DIRECTORY STRUCT
  VirtualAddress dd ?
  isize dd ?
IMAGE_DATA_DIRECTORY ENDS
```

VirtualAddress is actually the relative virtual address (RVA) of the data structure. For example, if this structure is for import symbols, this field contains the RVA of the **IMAGE_IMPORT_DESCRIPTOR** array.
isize contains the size in bytes of the data structure referred to by **VirtualAddress**.

Here's the general scheme on finding important data structures in a PE file:

1. From the DOS header, you go to the PE header
2. Obtain the address of the data directory in the optional header.
3. Multiply the size of **IMAGE_DATA_DIRECTORY** with the member index you want: for example if you want to know where the import symbols are, you must multiply the size of **IMAGE_DATA_DIRECTORY** (8 bytes) with 1.
4. Add the result to the address of the data directory and you have the address of the **IMAGE_DATA_DIRECTORY** structure that contains the info about the desired data structure.

Now we will enter into the real discussion about the import table. The address of the import table is contained in the **VirtualAddress** field of the second member of the data directory. The import table is actually an array of **IMAGE_IMPORT_DESCRIPTOR** structures. Each structure contains information about a DLL the PE file imports symbols from. For example, if the PE file imports functions from 10 different DLLs, there will be 10 members in this array. The array is terminated by the member which contain all zeroes. Now we can examine the structure in detail:

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

The first member of this structure is a union. Actually, the union only provides the alias for **OriginalFirstThunk**, so you can call it "Characteristics". This member contains the the RVA of an array of **IMAGE_THUNK_DATA** structures.
What is **IMAGE_THUNK_DATA**? It's a union of dword size. Usually, we interpret it as the pointer to an **IMAGE_IMPORT_BY_NAME** structure. Note that **IMAGE_THUNK_DATA** contains the pointer to an **IMAGE_IMPORT_BY_NAME** structure: not the structure itself.
Look at it this way: There are several **IMAGE_IMPORT_BY_NAME** structures. We collect the RVA of those structures (**IMAGE_THUNK_DATAs**) into an array, terminate it with 0. Then we put the RVA of the array into **OriginalFirstThunk**.
The **IMAGE_IMPORT_BY_NAME** structure contains information about an import function. Now let's see what **IMAGE_IMPORT_BY_NAME** structure looks like:

```
IMAGE_IMPORT_BY_NAME STRUCT
  Hint dw ?
  Name1 db ?
IMAGE_IMPORT_BY_NAME ENDS
```

Hint contains the index into the export table of the DLL the function resides in. This field is for use by the PE loader so it can look up the function in the DLL's export table quickly.This value is not essential and some linkers may set the value in this field to 0.

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

TimeDateStamp and **ForwarderChain** are advanced stuff: We will talk about them after you have firm grasp of the other members.

Name1 contains the RVA to the name of the DLL, in short, the pointer to the name of the DLL. The string is an ASCIIZ one.

FirstThunk is very similar to **OriginalFirstThunk**, ie. it contains an RVA of an array of **IMAGE_THUNK_DATA** structures(a different array though).
Ok, if you're still confused, look at it this way: There are several **IMAGE_IMPORT_BY_NAME** structures. You create two arrays, then fill them with the RVAs of those **IMAGE_IMPORT_BY_NAME** structures, so both arrays contain exactly the same values (i.e. exact duplicate). Now you assign the RVA of the first array to **OriginalFirstThunk** and the RVA of the second array to **FirstThunk**.

OriginalFirstThunk		IMAGE_IMPORT_BY_NAME		FirstThunk
|				|
IMAGE_THUNK_DATA	--->	Function 1	<---	IMAGE_THUNK_DATA
IMAGE_THUNK_DATA	--->	Function 2	<---	IMAGE_THUNK_DATA
IMAGE_THUNK_DATA	--->	Function 3	<---	IMAGE_THUNK_DATA
IMAGE_THUNK_DATA	--->	Function 4	<---	IMAGE_THUNK_DATA
...	--->	...	<---	...
IMAGE_THUNK_DATA	--->	Function n	<---	IMAGE_THUNK_DATA

Now you should be able to understand what I mean. Don't be confused by the name **IMAGE_THUNK_DATA**: it's only an RVA into **IMAGE_IMPORT_BY_NAME** structure. If you replace the word **IMAGE_THUNK_DATA** with RVA in your mind, you'll perhaps see it more clearly. The number of array elements in **OriginalFirstThunk** and **FirstThunk** array depends on the functions the PE file imports from the DLL. For example, if the PE file imports 10 functions from kernel32.dll, **Name1** in the **IMAGE_IMPORT_DESCRIPTOR** structure will contain the RVA of the string "kernel32.dll" and there will be 10 **IMAGE_THUNK_DATA**s in each array.
The next question is: why do we need two arrays that are exactly the same? To answer that question, we need to know that when the PE file is loaded into memory, the PE loader will look at the **IMAGE_THUNK_DATA**s and **IMAGE_IMPORT_BY_NAME**s and determine the addresses of the import functions. Then it replaces the **IMAGE_THUNK_DATA**s in the array pointed to by **FirstThunk** with the real addresses of the functions. Thus when the PE file is ready to run, the above picture is changed to:

OriginalFirstThunk		IMAGE_IMPORT_BY_NAME	FirstThunk
|			|
IMAGE_THUNK_DATA	--->	Function 1	Address of Function 1
IMAGE_THUNK_DATA	--->	Function 2	Address of Function 2
IMAGE_THUNK_DATA	--->	Function 3	Address of Function 3
IMAGE_THUNK_DATA	--->	Function 4	Address of Function 4
...	--->	...	...
IMAGE_THUNK_DATA	--->	Function n	Address of Function n

The array of RVAs pointed to by **OriginalFirstThunk** remains unchanged so that if the need arises to find the names of import functions, the PE loader can still find them.
There is a little twist on this *straightforward* scheme. Some functions are exported by ordinal only. It means you don't call the functions by their names: you call them by their positions. In this case, there will be no **IMAGE_IMPORT_BY_NAME** structure for that function in the caller's module. Instead, the **IMAGE_THUNK_DATA** for that function will contain the ordinal of the function in the low word and the most significant bit (MSB) of **IMAGE_THUNK_DATA** set to 1. For example, if a function is exported by ordinal only and its ordinal is 1234h, the **IMAGE_THUNK_DATA** for that function will be 80001234h. Microsoft provides a handy constant for testing the MSB of a dword, **IMAGE_ORDINAL_FLAG32**. It has the value of 80000000h.

Suppose that we want to list the names of ALL import functions of a PE file, we need to follow the steps below:

1. Verify that the file is a valid PE
2. From the DOS header, go to the PE header
3. Obtain the address of the data directory in **OptionalHeader**
4. Go to the 2nd member of the data directory. Extract the value of **VirtualAddress**
5. Use that value to go to the first **IMAGE_IMPORT_DESCRIPTOR** structure
6. Check the value of **OriginalFirstThunk**. If it's not zero, follow the RVA in **OriginalFirstThunk** to the RVA array. If **OriginalFirstThunk** is zero, use the value in **FirstThunk** instead. Some linkers generate PE files with 0 in **OriginalFirstThunk**. This is considered a bug. Just to be on the safe side, we check the value in **OriginalFirstThunk** first.
7. For each member in the array, we check the value of the member against **IMAGE_ORDINAL_FLAG32**. If the most significant bit of the member is 1, then the function is exported by ordinal and we can extract the ordinal number from the low word of the member.
8. If the most significant bit of the member is 0, use the value in the member as the RVA into the **IMAGE_IMPORT_BY_NAME**, skip **Hint**, and you're at the name of the function.
9. Skip to the next array member, and retrieve the names until the end of the array is reached (it's null -terminated). Now we are done extracting the names of the functions imported from a DLL. We go to the next DLL.
10. Skip to the next **IMAGE_IMPORT_DESCRIPTOR** and process it. Do that until the end of the array is reached (**IMAGE_IMPORT_DESCRIPTOR** array is terminated by a member with all zeroes in its fields).

Example:

This example opens a PE file and reads the names of all import functions of that file into an edit control. It also shows the values in the **IMAGE_IMPORT_DESCRIPTOR** structures.

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib

IDD_MAINDLG equ 101
IDC_EDIT equ 1000
IDM_OPEN equ 40001
IDM_EXIT equ 40003

DlgProc proto :DWORD,:DWORD,:DWORD,:DWORD
ShowImportFunctions proto :DWORD
ShowTheFunctions proto :DWORD,:DWORD
AppendText proto :DWORD,:DWORD

SEH struct
PrevLink dd ? ; the address of the previous seh structure
CurrentHandler dd ? ; the address of the new exception handler
SafeOffset dd ? ; The offset where it's safe to continue execution
PrevEsp dd ? ; the old value in esp
PrevEbp dd ? ; The old value in ebp
SEH ends
```

```
.data
AppName db "PE tutorial no.6",0
ofn OPENFILENAME <>
FilterString db "Executable Files (*.exe, *.dll)",0,"*.exe;*.dll",0
             db "All Files",0,"*.*",0,0
FileOpenError db "Cannot open the file for reading",0
FileOpenMappingError db "Cannot open the file for memory mapping",0
FileMappingError db "Cannot map the file into memory",0
NotValidPE db "This file is not a valid PE",0
CRLF db 0Dh,0Ah,0
ImportDescriptor db 0Dh,0Ah,"==================[ IMAGE_IMPORT_DESCRIPTOR
]=============",0
IDTemplate db "OriginalFirstThunk = %lX",0Dh,0Ah
           db "TimeDateStamp = %lX",0Dh,0Ah
           db "ForwarderChain = %lX",0Dh,0Ah
           db "Name = %s",0Dh,0Ah
           db "FirstThunk = %lX",0
NameHeader db 0Dh,0Ah,"Hint Function",0Dh,0Ah
           db "-----------------------------------------",0
NameTemplate db "%u %s",0
OrdinalTemplate db "%u (ord.)",0

.data?
buffer db 512 dup(?)
hFile dd ?
hMapping dd ?
pMapping dd ?
ValidPE dd ?

.code
start:
invoke GetModuleHandle,NULL
invoke DialogBoxParam, eax, IDD_MAINDLG,NULL,addr DlgProc, 0
invoke ExitProcess, 0

DlgProc proc hDlg:DWORD, uMsg:DWORD, wParam:DWORD, lParam:DWORD
.if uMsg==WM_INITDIALOG
  invoke SendDlgItemMessage,hDlg,IDC_EDIT,EM_SETLIMITTEXT,0,0
.elseif uMsg==WM_CLOSE
  invoke EndDialog,hDlg,0
.elseif uMsg==WM_COMMAND
  .if lParam==0
    mov eax,wParam
    .if ax==IDM_OPEN
      invoke ShowImportFunctions,hDlg
    .else ; IDM_EXIT
      invoke SendMessage,hDlg,WM_CLOSE,0,0
    .endif
  .endif
.else
  mov eax,FALSE
  ret
.endif
```

```
mov eax,TRUE
ret
DlgProc endp

SEHHandler proc C pExcept:DWORD, pFrame:DWORD, pContext:DWORD, pDispatch:DWORD
  mov edx,pFrame
  assume edx:ptr SEH
  mov eax,pContext
  assume eax:ptr CONTEXT
  push [edx].SafeOffset
  pop [eax].regEip
  push [edx].PrevEsp
  pop [eax].regEsp
  push [edx].PrevEbp
  pop [eax].regEbp
  mov ValidPE, FALSE
  mov eax,ExceptionContinueExecution
  ret
SEHHandler endp

ShowImportFunctions proc uses edi hDlg:DWORD
  LOCAL seh:SEH
  mov ofn.lStructSize,SIZEOF
  ofn mov ofn.lpstrFilter, OFFSET FilterString
  mov ofn.lpstrFile, OFFSET buffer
  mov ofn.nMaxFile,512
  mov ofn.Flags, OFN_FILEMUSTEXIST or OFN_PATHMUSTEXIST or OFN_LONGNAMES or
OFN_EXPLORER or OFN_HIDEREADONLY
  invoke GetOpenFileName, ADDR ofn
  .if eax==TRUE
    invoke CreateFile, addr buffer, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL
    .if eax!=INVALID_HANDLE_VALUE
      mov hFile, eax
      invoke CreateFileMapping, hFile, NULL, PAGE_READONLY,0,0,0
      .if eax!=NULL
        mov hMapping, eax
        invoke MapViewOfFile,hMapping,FILE_MAP_READ,0,0,0
        .if eax!=NULL
          mov pMapping,eax
          assume fs:nothing
          push fs:[0]
          pop seh.PrevLink
          mov seh.CurrentHandler,offset SEHHandler
          mov seh.SafeOffset,offset FinalExit
          lea eax,seh
          mov fs:[0], eax
          mov seh.PrevEsp,esp
          mov seh.PrevEbp,ebp
          mov edi, pMapping
          assume edi:ptr IMAGE_DOS_HEADER
          .if [edi].e_magic==IMAGE_DOS_SIGNATURE
            add edi, [edi].e_lfanew
            assume edi:ptr IMAGE_NT_HEADERS
```

```
            .if [edi].Signature==IMAGE_NT_SIGNATURE
              mov ValidPE, TRUE
            .else
              mov ValidPE, FALSE
            .endif
          .else
            mov ValidPE,FALSE
          .endif
FinalExit:
          push seh.PrevLink
          pop fs:[0]
          .if ValidPE==TRUE
            invoke ShowTheFunctions, hDlg, edi
          .else
            invoke MessageBox,0, addr NotValidPE, addr AppName, MB_OK+MB_ICONERROR
          .endif
          invoke UnmapViewOfFile, pMapping
      .else
          invoke MessageBox, 0, addr FileMappingError, addr AppName, MB_OK+MB_ICONERROR
      .endif
      invoke CloseHandle,hMapping
    .else
      invoke MessageBox, 0, addr FileOpenMappingError, addr AppName, MB_OK+MB_ICONERROR
    .endif
    invoke CloseHandle, hFile
   .else
   invoke MessageBox, 0, addr FileOpenError, addr AppName, MB_OK+MB_ICONERROR
   .endif
  .endif
 ret
ShowImportFunctions endp

AppendText proc hDlg:DWORD,pText:DWORD
   invoke SendDlgItemMessage,hDlg,IDC_EDIT,EM_REPLACESEL,0,pText
   invoke SendDlgItemMessage,hDlg,IDC_EDIT,EM_REPLACESEL,0,addr CRLF
   invoke SendDlgItemMessage,hDlg,IDC_EDIT,EM_SETSEL,-1,0
   ret
AppendText endp

RVAToOffset PROC uses edi esi edx ecx pFileMap:DWORD,RVA:DWORD
   mov esi,pFileMap
   assume esi:ptr IMAGE_DOS_HEADER
   add esi,[esi].e_lfanew
   assume esi:ptr IMAGE_NT_HEADERS
   mov edi,RVA ; edi == RVA
   mov edx,esi
   add edx,sizeof IMAGE_NT_HEADERS
   mov cx,[esi].FileHeader.NumberOfSections
   movzx ecx,cx
   assume edx:ptr IMAGE_SECTION_HEADER
   .while ecx>0 ; check all sections
     .if edi>=[edx].VirtualAddress
       mov eax,[edx].VirtualAddress
       add eax,[edx].SizeOfRawData
```

```
      .if edi<eax ; The address is in this section
        mov eax,[edx].VirtualAddress
        sub edi,eax
        mov eax,[edx].PointerToRawData
        add eax,edi ; eax == file offset
        ret
      .endif
    .endif
    add edx,sizeof IMAGE_SECTION_HEADER
    dec ecx
  .endw
  assume edx:nothing
  assume esi:nothing
  mov eax,edi
  ret
RVAToOffset endp

ShowTheFunctions proc uses esi ecx ebx hDlg:DWORD, pNTHdr:DWORD
  LOCAL temp[512]:BYTE
  invoke SetDlgItemText,hDlg,IDC_EDIT,0
  invoke AppendText,hDlg,addr buffer
  mov edi,pNTHdr
  assume edi:ptr IMAGE_NT_HEADERS
  mov edi, [edi].OptionalHeader.DataDirectory[sizeof IMAGE_DATA_DIRECTORY].VirtualAddress
  invoke RVAToOffset,pMapping,edi
  mov edi,eax
  add edi,pMapping
  assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
  .while !([edi].OriginalFirstThunk==0 && [edi].TimeDateStamp==0 && [edi].ForwarderChain==0 &&
[edi].Name1==0 && [edi].FirstThunk==0)
    invoke AppendText,hDlg,addr ImportDescriptor
    invoke RVAToOffset,pMapping, [edi].Name1
    mov edx,eax
    add edx,pMapping
    invoke wsprintf, addr temp, addr IDTemplate, [edi].OriginalFirstThunk,[edi].TimeDateStamp,
[edi].ForwarderChain,edx,[edi].FirstThunk      invoke AppendText,hDlg,addr temp
    .if [edi].OriginalFirstThunk==0
      mov esi,[edi].FirstThunk
    .else
      mov esi,[edi].OriginalFirstThunk
    .endif
    invoke RVAToOffset,pMapping,esi
    add eax,pMapping
    mov esi,eax
    invoke AppendText,hDlg,addr NameHeader
    .while dword ptr [esi]!=0
     test dword ptr [esi],IMAGE_ORDINAL_FLAG32
     jnz ImportByOrdinal
     invoke RVAToOffset,pMapping,dword ptr [esi]
     mov edx,eax
     add edx,pMapping
     assume edx:ptr IMAGE_IMPORT_BY_NAME
     mov cx, [edx].Hint
     movzx ecx,cx
```

```
      invoke wsprintf,addr temp,addr NameTemplate,ecx,addr [edx].Name1
      jmp ShowTheText
ImportByOrdinal:
      mov edx,dword ptr [esi]
      and edx,0FFFFh
      invoke wsprintf,addr temp,addr OrdinalTemplate,edx
ShowTheText:
      invoke AppendText,hDlg,addr temp
      add esi,4
    .endw
    add edi,sizeof IMAGE_IMPORT_DESCRIPTOR
  .endw
  ret
ShowTheFunctions endp
end start
```

Analysis:

The program shows an open file dialog box when the user clicks Open in the menu. It verifies that the file is a valid PE and then calls **ShowTheFunctions**.

```
ShowTheFunctions proc uses esi ecx ebx hDlg:DWORD, pNTHdr:DWORD
  LOCAL temp[512]:BYTE
```

Reserve 512 bytes of stack space for string operation.

```
  invoke SetDlgItemText,hDlg,IDC_EDIT,0
```

Clear the text in the edit control

```
  invoke AppendText,hDlg,addr buffer
```

Insert the name of the PE file into the edit control. **AppendText** just sends **EM_REPLACESEL** messages to append the text to the edit control. Note that it sends **EM_SETSEL** with wParam=-1 and lParam=0 to the edit control to move the cursor to the end of the text.

```
  mov edi,pNTHdr
  assume edi:ptr IMAGE_NT_HEADERS
  mov edi, [edi].OptionalHeader.DataDirectory[sizeof IMAGE_DATA_DIRECTORY].VirtualAddress
```

Obtain the RVA of the import symbols. edi at first points to the PE header. We use it to go to the 2nd member of the data directory array and obtain the value of VirtualAddress member.

```
  invoke RVAToOffset,pMapping,edi
  mov edi,eax
  add edi,pMapping
```

Here comes one of the pitfalls for newcomers to PE programming. Most of the addresses in the PE file are RVAs and **RVAs are meaningful only when the PE file is loaded into memory by the PE loader**. In our case, we do map the file into memory but not the way the PE loader does. Thus we cannot use those RVAs directly. Somehow we have to convert those RVAs into file offsets. I write RVAToOffset function just for this purpose. I won't analyze it in detail here. Suffice to say that it checks the submitted RVA against the starting-ending RVAs of all sections in the PE file and use the value in **PointerToRawData** field in the **IMAGE_SECTION_HEADER** structure to convert the RVA to file offset.
To use this function, you pass it two parameters: the pointer to the memory mapped file and the RVA you want to convert. It returns the file offset in eax. In the above snippet, we must add the pointer to the memory mapped file to the file offset to convert it to virtual address. Seems complicated, huh? :)

```
  assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
```

```
.while !([edi].OriginalFirstThunk==0 && [edi].TimeDateStamp==0 && [edi].ForwarderChain==0 && [edi].Name1==0 && [edi].FirstThunk==0)
```

edi now points to the first **IMAGE_IMPORT_DESCRIPTOR** structure. We will walk the array until we find the structure with zeroes in all members which denotes the end of the array.

```
invoke AppendText,hDlg,addr ImportDescriptor
invoke RVAToOffset,pMapping, [edi].Name1
mov edx,eax
add edx,pMapping
```

We want to display the values of the current **IMAGE_IMPORT_DESCRIPTOR** structure in the edit control. Name1 is different from the other members since it contains the RVA to the name of the dll. Thus we must convert it to a virtual address first.

```
invoke wsprintf, addr temp, addr IDTemplate, [edi].OriginalFirstThunk,[edi].TimeDateStamp, [edi].ForwarderChain,edx,[edi].FirstThunk      invoke AppendText,hDlg,addr temp
```

Display the values of the current **IMAGE_IMPORT_DESCRIPTOR**.

```
.if [edi].OriginalFirstThunk==0
  mov esi,[edi].FirstThunk
.else
  mov esi,[edi].OriginalFirstThunk
.endif
```

Next we prepare to walk the **IMAGE_THUNK_DATA** array. Normally we would choose to use the array pointed to by **OriginalFirstThunk**. However, some linkers errornously put 0 in **OriginalFirstThunk** thus we must check first if the value of **OriginalFirstThunk** is zero. If it is, we use the array pointed to by **FirstThunk** instead.

```
invoke RVAToOffset,pMapping,esi
add eax,pMapping
mov esi,eax
```

Again, the value in **OriginalFirstThunk**/**FirstThunk** is an RVA. We must convert it to virtual address.

```
invoke AppendText,hDlg,addr NameHeader
.while dword ptr [esi]!=0
```

Now we are ready to walk the array of **IMAGE_THUNK_DATAs** to look for the names of the functions imported from this DLL. We will walk the array until we find an entry which contains 0.

```
test dword ptr [esi],IMAGE_ORDINAL_FLAG32
jnz ImportByOrdinal
```

The first thing we do with the **IMAGE_THUNK_DATA** is to test it against **IMAGE_ORDINAL_FLAG32**. This test checks if the most significant bit of the **IMAGE_THUNK_DATA** is 1. If it is, the function is exported by ordinal so we have no need to process it further. We can extract its ordinal from the low word of the **IMAGE_THUNK_DATA** and go on with the next **IMAGE_THUNK_DATA** dword.

```
invoke RVAToOffset,pMapping,dword ptr [esi]
mov edx,eax
add edx,pMapping
assume edx:ptr IMAGE_IMPORT_BY_NAME
```

If the MSB of the IAMGE_THUNK_DATA is 0, it contains the RVA of IMAGE_IMPORT_BY_NAME structure. We need to convert it to virtual address first.

```
mov cx, [edx].Hint
```

```
        movzx ecx,cx
        invoke wsprintf,addr temp,addr NameTemplate,ecx,addr [edx].Name1
        jmp ShowTheText
```

Hint is a word-sized field. We must convert it to a dword-sized value before submitting it to wsprintf. And we print both the hint and the function name in the edit control

```
ImportByOrdinal:
        mov edx,dword ptr [esi]
        and edx,0FFFFh
        invoke wsprintf,addr temp,addr OrdinalTemplate,edx
```

In the case the function is exported by ordinal only, we zero out the high word and display the ordinal.

```
ShowTheText:
        invoke AppendText,hDlg,addr temp
        add esi,4
```

After inserting the function name/ordinal into the edit control, we skip to the next **IMAGE_THUNK_DATA**.

```
    .endw
    add edi,sizeof IMAGE_IMPORT_DESCRIPTOR
```

When all **IMAGE_THUNK_DATA** dwords in the array are processed, we skip to the next **IMAGE_IMPORT_DESCRIPTOR** to process the import functions from other DLLs.

Appendix:

It would be incomplete if I don't mention something about bound import. In order to explain what it is, I need to digress a bit. When the PE loader loads a PE file into memory, it examines the import table and loads the required DLLs into the process address space. Then it walks the **IMAGE_THUNK_DATA** array much like we did and replaces the **IMAGE_THUNK_DATAs** with the real addresses of the import functions. This step takes time. If somehow the programmer can predict the addresses of the functions correctly, the PE loader doesn't have to fix the **IMAGE_THUNK_DATAs** each time the PE file is run. Bound import is the product of that idea.
To put it in simple terms, there is a utility named **bind.exe** that comes with Microsoft compilers such as Visual Studio that examines the import table of a PE file and replaces the **IMAGE_THUNK_DATA** dwords with the addresses of the import functions.When the file is loaded, the PE loader must check if the addresses are valid. If the DLL versions do not match the ones in the PE files or if the DLLs need to be relocated, the PE loader knows that the precomputed addresses are not valid thus it must walk the array pointed to by **OriginalFirstThunk** to calculate the new addresses of import functions.
Bound import doesn't have much significance in our example because we use OriginalFirstThunk by default. For more information about the bound import, I recommmend LUEVELSMEYER's pe.txt.
