© The CodeBreakers-Journal, Vol. 1, No. 1 (2004)
http://www.CodeBreakers-Journal.com

Adding Imports By Hand

Author: Eduardo Labir

Abstract

This is a tutorial about the imports i wrote "by the way" while solving a harder problem. It focus on 2 particular problems: Building up all the imports information and adding imports (both by hand). It does NOT touch the problem of reconstructing a dumped imports section but, hopefully, i will release tutorials on the later as i tackle more well-known packers. Please, send me your comments - either positives or negatives - to LordOfHavok@hotmail.com or simply look for me at board.anticrack.de. Your feedback is always wellcome. While elaborating this tutorial i worked on Win98 and WinXP, but everything should be a trivial translation for other OSes (excluding section 5, but this deal with the differences between Win2k and the rest of Oses).

Keywords: *Reverse Code Engineering, Adding Functionality*

*The author **Eduardo Labir** has his main research focus on Virus Analysis and Reverse Code Engineering (RCE). He is author of several publications covering the analysis of viral functionality as an integral part. He is Associate Editor of the CodeBreakers-Journal.*

I. Introduction

Understanding the imports is a must for anyone wanting to be a good reverser. When i started studying this topic on my own i saw the lack of tutorials on this subject, in particular touching how to add yourself new imports to a program. Yes, it's true that you can use Snippet Creator or any other tool and never need this information but is preferable, by far, to have the background. This tutorial is particularly focused on how to add new imports but you will hopefully find accurate and detailed explainations about more or less everything you might need.

We'll work with the following examples:

```
notepad.exe
wordpad.exe
main.exe (source code below)
cplayer.exe (look at c:\windows)
```

Tools you need: A debugger, an hex editor and procdump (or another PE editor).

II. Understanding the Imports

The different APIs we use can be "linked" by the Window's loader. The best way of understanding this it's to examine different cases and not to get lost into useless definitions and attempts to formalize this stuff. Our first example, Notepad, shows that sometimes our conceptions about how the APIs get linked are totally wrong. Let's see what i mean...

1) AN STRANGE SURPRISE: If you dissamble NOTEPAD.exe you can see the following calls:

```
call dword ptr [4063E0]   ; call to kernel32.GetCommandLineA
...
call dword ptr [406390]     ; call to kernel32.GetStartupInfoA
...
```

At those offsets we have the following:

```
offset    hex

4063E0: 83092EB8
406390: 83092D78
...
```

So we're actually doing "call 83092EB8", "call 83092D78". But, what can we find there?. This is the answer:

```
83092EB8:    68 DAC5F8BF             PUSH KERNEL32.GetCommandLineA
83092EBD:   -E9 0829F03C             JMP KERNEL32.BFF957CA

83092D78:    68 AD77F7BF             PUSH KERNEL32.GetStartupInfoA
83092D7D:   -E9 482AF03C             JMP KERNEL32.BFF957CA
```

What are those strange "push KERNEL32.GetCommandLineA; JMP KERNEL32.BFF957CA"?. This is only generated when we open the app with a debugger, in real life you would find this:

```
83092EB8:   jmp KERNEL32.GetCommandLineA
83092D78:   jmp KERNEL32.GetStartupInfoA
```

We can derive a nice anti-debug trick from this... as you see the address of the API you wanted to call (when we debug) is pushed and then we have a jump to that address. Beware...

Starting again :)

In order to save workload, when you do a call, say call kernel32.ExitProcess, this gets compiled as follows:

```
           call dword ptr [xxxxxxxxh] ; call kernel32.ExitProcess
...
xxxxxxxxh: yyyyyyyyh
```

This way, yyyyyyyyh redirects all calls to that API to one "door" leading to it. Obviously, this is much faster than going thru each call and fixing it, as you can well imagine. But, What can we find at yyyyyyyyh ?. That yyyyyyyyh is a "door" to the API we want to call, the loader fills this information just after mapping the executable in memory and before jumping to the entry point. This "door" can be constructed in several ways, for example:

```
 ; this only when we debug

83092EB8:   push offset kernel32.ExitProcess
83092D78:   jmp kernel32.debugging

; a direct jump to the API

83092EB8:    jmp kernel32.ExitProcess

; the door can be the offset of the API so we're directly calling it

4063E0:     DD BFF8D4F8 ; the address of kernel32.ExitProcess
```

So, we're simply redirected to call the API. All these "doors" are called the IAT, Import Address Table. As we see, all calls are redirected by the IAT.

A. The IAT (Imports Address Table)

In our program we have lots of calls to different APIs imported from several DLLS, for example we can have:

```
00400300: call [xxxxxxxxh]        ; user32.MessageBoxA
...
0040041A: call [xxxxxxxxh]        ; user32.MessageBoxA, called many times
...
00400421:   call [yyyyyyyyh]      ; kernel32.ExitProcess
....
00400437: call [zzzzzzzzh]        ; Advapi32.RegOpenKeyA
...
```

All our calls are redirected thru those addresses: xxxxxxxxh, yyyyyyyyh, zzzzzzzzh,... This addresses are used to link all the calls to the same api with the IAT.

```
xxxxxxxxh: door to user32.MessageBoxA
yyyyyyyyh: door to jmp kernel32.ExitProcess
zzzzzzzzh: door to Advapi32.RegOpenKeyA
...
```

For example, if you have a look at main.exe (its code is below) then you see this:

```
00401000   /$ 6A 00          PUSH 0                       ; /Style = MB_OK|MB_APPLMODAL
00401002   |. 6A 00          PUSH 0                       ; |Title = NULL
00401004   |. 68 00204000    PUSH MAIN.00402000           ; |Text = "Hello"
00401009   |. 6A 00          PUSH 0                       ; |hOwner = NULL
0040100B   |. E8 0D000000    CALL JMP.&USER32.MessageBoxA ; \MessageBoxA
00401010   |. 6A 00          PUSH 0                          ; /ExitCode = 0
00401012   \. E8 00000000    CALL JMP.&KERNEL32.ExitProcess  ; \ExitProcess

00401017    .-FF25 4C304000  JMP DWORD PTR DS:[&KERNEL32.ExitProcess]
0040101D    $-FF25 54304000  JMP DWORD PTR DS:[&USER32.MessageBoxA]
```

You see, all calls are redirected thru those jumps, this jumps lead to the addresses given by the IAT. Any other call to MessageBox appearing in the program will get compiled as call jmp.&user32.MessageBoxA. This way, when you load the program the windows' loader only has to fill the IAT with the correct values and the exe works fine. The way the program uses to redirect the calls to the IAT depends on the particular programming language and compiler.

So the loader is in charge of filling the IAT with the right doors before executing our code (at loading time). If this information is not correctly set up then our program crashes when it calls the (incorrectly set) APIs. All the information the loader needs to fill the IAT can be found at the Imports Table, let's examine how it works.

B. The Imports Table

The imports table has all the information Windows needs to link the APIs for your program. The imports table has a very simple structure: there's one header for each imported dll - there's also an extra one ,totally nulled, to mark its end - and each header contains all the information for one particular dll. For example if you import APIs from kernel32 and user32 you will find 3 headers, one for kernel32, a second one for user32 and an extra one to mark the end of the imports table. Windows, i.e. the Windows loader, reads the information from each header, and uses this information to fill the IAT (the IAT is made up of the IATs for each dll).

This header for each DLL is called `IMPORT_IMAGE_DIRECTORY`, the word 'IMAGE' recalls us thas all this stuff is done in memory and so all offsets are RVAs, and has the following structure:

```
   IMAGE_IMPORT_DESCRIPTOR struct
      OriginalFirstThunk      dd 0  ; RVA to original unbound IAT (table of names)
      TimeDateStamp           dd 0  ; not used here
      ForwarderChain          dd 0  ; not used here
      Name                    dd 0  ; RVA to DLL name sring
      FirstThunk              dd 0  ; RVA to IAT array (table of doors)
   IMAGE_IMPORT_DESCRIPTOR ends
```

At this point is where every tutorial i've read tries to make a formal definition therefore making this a mess. Let's try to understand this by means of our examples, it's much easier, the first one is Notepad:

If you open Notepad with procdump and go to the data directories you can see it has:

```
        Virtual Address    Virtual Size
Import    6000                 8C
IAT       62E0                 240
```

The import table tells the loader where to find all headers for all imported dlls. The virtual address of the import table has to be always correct, in our case 6000, the virtual size can be bigger (but only to a given bound), in our case the virtual size is 8C.

The values for the IAT are totally useless, indeed you can change them to what you want and the app still works. Note that we already have a pointer to the IAT into the `IMAGE_IMPORT_DESCRITOR`, the first thunk.

Let's make a little summary:

```
Import table: it consists of all IMAGE_IMPORT_DESCRIPTORs
              (headers for the dlls)
A given IMAGE_IMPORT_DESCRIPTOR gives the next information
about the imports of one Dll:
      name: name of the DLL.
      OrginalFirstThunk: names of the APIs we wanna import.
      FirstThunk: points to the IAT, the program redirects
                  all its calls to the IAT.
IAT: redirects all the calls made inside the program,
     they store the value of the doors
```

Back to the example (notepad):

For the imports, you can see that the virtual address is 6000 and the virtual size is 8C. Let's observe that this information refers to the `IMAGE_IMPORT_DESCRIPTORs`, therefore it means that all `IMAGE_IMPORT_DESCRIPTORs` (including the nulled terminating `IMAGE_IMPORT_DESCRIPTOR`) are mapped in memory at RVA 6000 and their total size is 8C bytes.

Now, you can go to the sections information and check which section are the imports inside. You will see that this section is called .idata, a common name for the imports section, and its raw offset is 6000 (simply compare the RVA of each section to the RVA of the imports). Observe that there's no need to have the imports into a detached section, many times you will find them below the code of the program. What matters is: they have to be mapped on memory and the loader has to have their RVA and size (to correctly set up the IAT).

If we wanna take a closer look at the imports table we need to open the file with an hex-editor, do it.This is what you will see, i.e. the imports table:

```
00006000  B0 61 00 00 CF 59 3B 37 FF FF FF FF 8A 65 00 00
00006010  F0 63 00 00 18 61 00 00 B3 C2 1F 37 FF FF FF FF
00006020  9C 67 00 00 58 63 00 00 CC 61 00 00 CA 59 3B 37
00006030  FF FF FF FF 92 6B 00 00 0C 64 00 00 B8 60 00 00
00006040  72 C2 3E 35 FF FF FF FF F6 6C 00 00 F8 62 00 00
00006050  C0 62 00 00 CF 59 3B 37 FF FF FF FF 7A 6D 00 00
00006060  00 65 00 00 A0 60 00 00 CA 59 3B 37 FF FF FF FF
00006070  DA 6D 00 00 E0 62 00 00 00 00 00 00 00 00 00 00
00006080  00 00 00 00 00 00 00 00 00 00 00 00
```

There you can find 7 consecutive `IMPORT_IMAGE_DESCRIPTORS`, one for each DLL and an extra one filled with zeroes. The size of the imports table, 8C, includes this last one null `IMPORT_IMAGE_DESCRIPTOR`.

Let's take for example the first `IMPORT_IMAGE_DESCRIPTOR`:

```
00006000  B0 61 00 00 CF 59 3B 37 FF FF FF FF 8A 65 00 00
00006010  F0 63 00 00
```

OriginalFirstThunk

The first dword, B0 61 00 00, is the OriginalFirstThunk. This gives the loader the information about where to find the names of the APIs to be imported from the current dll. If now we go to IMAGE_BASE + 000061B0, remember the address is kept in indian order, we can see the following:

```
004061B0  7A 65 00 00 68 65 00 00 20 65 00 00 4E 65 00 00
004061C0  3C 65 00 00 2E 65 00 00 00 00 00 00 86 6B 00 00
004061D0  8C 69 00 00 9E 69 00 00 BC 69 00 00 CC 69 00 00
```

This is an array of pointers, RVAs as usual, to the names of the imported APIS, for example:

First RVA:

```
0040657A  6C 00 53 68 65 6C 6C 45 78 65 63 75 74 65 41 00  l.ShellExecuteA.
```

Second RVA:

```
00406568  0F 00 44 72 61 67 41 63 63 65 70 74 46 69 6C 65  M.DragAcceptFile
00406578  73 00                                            s.
```

As you see, there's a word before the name of each API - 6c 00 in the first case and 0F 00 in the second one. This is the hint, most times the ordinal of the API. You're not compelled to inform it, indeed you will see many times all hints set to zero.

As you know, the APIs can be imported by name or by ordinal. In the second case what you will find, instead of the pointer to the hint, a dword of this kind: 80 00 00 3A, where 3A is the ordinal and 80 marks that the import is by ordinal. We'll see an example of importing by ordinals below, inside wordpad.exe.

TimeDateStamp, ForwarderChain: They are useless for us.

Name:

This is the RVA to the name of the DLL the IMAGE_IMPORT_DESCRIPTOR corresponds to. For example, if we take this dword at the first IMAGE_IMPORT_DESCRIPTOR its value is (in indian order) 8A 65 00 00. Meaning the name of the DLL is at IMAGE_BASE + 8A 65 00 00 = 0040658A, where you can find this:

```
0040658A  53 48 45 4C 4C 33 32 2E 64 6C 6C 00              SHELL32.dll.
```

Where, obviously, ShellExecuteA is imported from.

FirstThunk (once the app is loaded in memory):

Points to, is an RVA to, the IAT.
For example, if we want to import ExitProcess, MessageBoxA and CreateFileA from kernel32 the first thunk would look as follows:

```
   First Thunk
Door to ExitProcess ; first API imported
Door to MessageBoxA ; second API imported
Door to CreateFileA ; third and last API imported
Null pointer ; terminating the list
```

Let's examine our example, the first `IMPORT_IMAGE_DESCRIPTOR` of notepad.exe has FirstThunk = F0 63 00 00 (in indian order). This means the IAT can be found at 004063F0. There we have the following:

```
004063F0 >FF 6A D1 7F 0F 28 CD 7F 66 16 CE 7F 7C 84 CB 7F
00406400 >EB C6 CC 7F 7B 42 D1 7F 00 00 00 00
```

As you can see (Win98) the doors are the offsets of the APIs inside SHELL32.dll (not redirected jumps):

```
FF 6A D1 7F => ShellExecuteA
0F 28 CD 7F => DragAcceptFilesA
66 16 CE 7F => ShellAboutA
7C 84 CB 7F => GHGetSpecialFolderPathA
EB C6 CC 7F => DragQueryFileA
7B 42 D1 7F => DragQueryPoint
```

An important note about the FirstThunk

If you recall, we've just said above that the FirstThunk contained the RVAs to the names of the APIs you wanna import from that particular dll... Well, is not always true. Exceptionally, it can contain harcoded offsets of the different APIs. For example, let's open NOTEPAD (the .exe file) and let's go to raw offset 6000, we see the following (just follow the pointers to the FirstThunk):

```
0000644F: BF 0D 58 F5 BF 1D 17 F5 BF DC 28 F5 BF A0 2F F5
0000645F: BF A4 20 F5 BF 04 18 F5 BF 28 58 F5 BF 36 4F F5
```

What's that? addresses of different APIs that got stamped there by the linker... Observe that this seldomly happen and moreover hasn't much influence in our work. We can simply overwrite the contents of the FirstThunk with the contents of the OriginalFirstThunk. Indeed, in terms of compatibility of applications, having that hardcoded offsets is VERY LAME. And should be always avoided.

C. The Windows Loader

In this section, I'm gonna try to explain the work the loader does to fill the IAT by means of the import table. First of all, you need to know two very importants APIs: LoadLibraryA and GetProcAddress. You can find detailed descriptions about them at win32.hlp but let me summarize what they do:

LoadLibrary loads a DLL, retuns a handle to the loaded DLL.

```
HINSTANCE LoadLibrary(

    LPCTSTR  lpLibFileName  // pointer to a zeroe
                               terminated string (name of the DLL)
   );
```

GetProcAddress returns the address of an API exported by a given DLL.

```
FARPROC GetProcAddress(

    HMODULE  hModule,// handle to DLL (returned by
                         LoadLibraryA or GetModuleHandleA)
    LPCSTR  lpProcName  // name of function
   );
```

Imagine you have available LoadLibraryA and GetProcAddress and you're given the next list of APIs to load:

```
user32:
      "MessageBoxA"
      "IsWindow"
```

How would you do it?. You should first call LoadLibraryA to load user32 and then, with the handle it returns, you could get the address of each imported API you want to call. For example:

```
_user32           dd 0                    ; hadle to user32
_MessageBoxA      dd 0                    ; address of MessageBoxA
szUser32          db 'user32.dll',0       ; name of dll
szMessageBoxA     db 'MessageBoxA',0      ; name of the API to import
szHello           db 'hello',0            ; text for calling MessageBoxA


      push offset szUser32 ; load user32
  call LoadLibraryA
      mov dword ptr [_user32], eax

      test eax, eax ; check error code
      je error ;

      push offset szMessageBoxA ; name of the API
      push dword ptr [_user32] ; handle (image base)
      call GetProcAddress ;
      mov dword ptr [_MessageBoxA], eax ; keep address

      ...

      push 0 ; now you can call MessageBoxA
      push 0 ; shows "error" in the message box title
      push offset szHello ;
      push 0 ;
      call dword ptr [MessageBoxA] ;
```

Description of the process carried out by the loader:

For each `IMPORT_IMAGE_DESCRIPTOR` (but the last, nulled, one) the loader does a very similiar process to what we did in the previous section, let's see how it works:

Suppose you wanna import, from kernel32, the APIs ExitProcess, CreateProcess and LoadLibraryA. Then, in to the .exe file, in the hard drive, you have this information:

```
                                                                   IAT
"KERNEL32.DLL"
     ^
     |
     |                                             VVVVVVVVVVV
IMAGE_IMPORT_DESCRIPTOR for KERNEL32  -----------> RVA to the API name 'ExitProcess'
     |                                FirstThunk   RVA to the API name 'CreateProcess'
     | OriginalFirstThunk                          RVA to the API name 'LoadLibraryA'
     |                                             null pointer
     v
RVA to the API name 'ExitProcess'
RVA to the API name 'CreateProcess'
RVA to the API name 'LoadLibraryA'
null pointer
```

```
; somewhere lost in the .exe file, you have the name of the APIs

0000A000: 'ExitProcess', 0, 'CreateProcess', 0, 'LoadLibraryA',0
```

When the loader acceeds to the `IMPORT_IMAGE_DESCRIPTOR` it first checks the DLL it refers to by examining its name. Next, the loader loads this DLL and start constructing the IAT. Constructing the IAT is a bit tricky: First, the loader examine the OriginalFirstThunk, but this information is only used in case of trouble. Next, for each name pointed by the FirstThunk it calls GetProcAddress and substitutes the pointer to the name by the door (the offset of the API, a jump to it,...). If the API is not found, f.e. cos the name's wrong spelling, then it goes to the OriginalFirstThunk and try to get this information from there. If this final possibility still doesn't work then it crashes.

Therefore, in memory, you will see that all pointers into the FirstThunk contain doors to the APIs from the current DLL instead of RVAs to the names of the APIs. Note that the IAT construction is done AFTER the exe has been mapped in memory, this means that if a process is created as "suspended" then the IAT hasn't been constructed yet (if you hook the IAT this will be overwritten by the loader).

Now, everything makes a sense:

OriginalFirstThunk: back-up of the FirstThunk, used in case of trouble.

FirstThunk: array of pointers to the names of the APIs we need to import (actually to the hints). The loader, for each pointer, reads the name of the API and looks for its address. If it finds the address (or a door to it) then it substitutes the pointer to the name by it, otherwise it goes to the back-up (OriginalFirstThunk) and tries its luck. IAT: If the loading process has been correct, all the pointers into the FirstThunk has been substituted by the addresses of the APIs (now all that addresses are called IAT). All calls from the program have to be redirected to the IAT, the way of achieving this depends on how you compile + link the code. The addresses stamped into the the IAT by the loader can be:

1) the actual address of the API
2) a jump to the API
3) push offset API; jmp procedure to filter (when we debug)

Therefore:

On memory you will find the IAT informed with the addresses of the APIs.
On the file you will find the FirstThunk informed with the RVAs to the names of the APIs.

The program, when it's compiled + linked, redirects all its calls to the IAT with the hope of finding there a valid address for the API which is calling:

```
call dword ptr [xxxxxxxxh]          ; call CreateProcess
...
call dword ptr [zzzzzzzzh]          ; call LoadLibraryA
...
call dword ptr [yyyyyyyyh]          ; call ExitProcess
```

```
xxxxxxxxh: offset of IAT(2)
yyyyyyyyh: offset of IAT(1)
zzzzzzzzh: offset of IAT(3)
```

As you see, the order hasn't why to be preserved. This way, all calls are redirect thru the jumps to the different entries of the IAT.

Remark: All hints can be set to zeroe but if they are informed they will probably give the actual API.

D. What do wee need to have a correctly set up windows table?

1) First of all, the RVA of the Import Table, and its size, need to be set into the data directory for the imports. Otherwise, Windows is unable to find it and therefore the IAT won't be informed. There's no need of having the imports into a separated section.
2) You need to declare each DLL with a `IMAGE_IMPORT_DESCRIPTOR` and you have to close the Import Table with a totally nulled one.
3) The `IMAGE_IMPORT_DESCRIPTOR` needs to have the OriginalFirstThunk, FirstThunk and Name well informed. The rest of fields can be set to zeroe.
4) Each entry of the FirstThunk, IAT, must be an RVA to an API name (to the hint). If you have placed into IAT[N] the API GetCurrentThread, all calls into the program redirected to IAT[N] have to be exatly those to that API. This can be a problem cos the IAT hasn't why to be ordered (neither in alphabetical order, nor...).
5) The OriginalFirstThunk should be a back-up of the first think with the RVAs and so they should agree. NOTE: setting OriginalFirstThunk = FirstThunk (i.e. equalling the RVAs) works.

As you see, a lot of work to be done. Fortunately for us there're several tools that can make our life easier (i won't review them in this paper, i just focus on understanding this stuff) like f.e. SnippetCreator.

1) Example 2. MAIN.EXE:

The code for main.exe, that can be directly compiled with TASM, is the following:

```
;------------------------------------------------------------------
;                                     main.asm
;------------------------------------------------------------------
; To compile this file with TASM make a .BAT with the following:
;     @echo off
;     tasm32 /ml /m2 /w2 /zi main,,;
;     tlink32 /r  /Tpe /aa /c /v main,main,, import32.lib
;------------------------------------------------------------------
.386p
.model flat, stdcall

extrn       ExitProcess : PROC
extrn       MessageBoxA : PROC

.DATA
       szHello    db 'Hello',0


 .CODE

Main:


      push 0
      push 0 ; pushing 0 here displays "error" as a title
      push offset szHello
      push 0
      call MessageBoxA

      push 0
      Call ExitProcess

End   Main                    ; End of code
```

As you see, a totally plain application that displays a message box and exits. Compile it and let's have a look at the imports directory:

```
          Virtual Address Virtual Size
Import        3000              90
IAT             0                0
```

I told you the IAT fields were not neccesary... didn't i? Indeed, you can set the IAT to any value and it'll always work correctly. If you recall, all the information needed to construct the IAT is already included into the Imports Table.

Main.exe is importing two APIs from 2 different dlls: MessageBoxA, from user32, and ExitProcess from kernel32. Therefore we will have 3 `IMPORT_IMAGE_DESCRIPTORs`, the last one being all nulled. If we have a look at the sections we see that the imports table starts at raw offset A00 and has raw size 200. Let's study the `IMPORT_IMAGE_DESCRIPTORs`, the starting offset is (in memory) 00403000 (you can also go to the file at offset A00):

```
00403000  3C 30 00 00 B3 C2 1F 37 00 00 F7 BF 5C 30 00 00
00403010  4C 30 00 00 44 30 00 00 CA 59 3B 37 00 00 F5 BF
00403020  69 30 00 00 54 30 00 00 00 00 00 00 00 00 00 00
00403030  00 00 00 00 00 00 00 00 00 00 00 00
```

You can see the nulled `IMPORT_IMAGE_DESCRIPTOR`, at the end. The first `IMPORT_IMAGE_DESCRIPTOR` has Original-FirstThunk = 0000303C, wich points (is an RVA) to the following:

```
0040303C  74 30 00 00 00 00 00 00
```

This is an RVA to the name of the first API followed by a zero dword to mark the end of the list of names. The name pointed, you can find it at 00403074, is ExitProcess. Pay attention on the hint, it's set to zeroe. We already have commented that this wasn't neccessary for the loader (the hint is only that, a "hint" to find a given API faster).

`IMPORT_IMAGE_DESCRIPTOR.name` points to "kernel32.dll" (ExitProcess belongs to kernel32).

Finally, we're gonna examine the FirstThunk for this `IMPORT_IMAGE_DESCRIPTOR`, RVA = 0000304C:

```
0040304C >10 A1 01 83 00 00 00 00
```

Note we're examining the exe once loaded, so we haven't an RVA to an API. For kernel32 we have one address stored - the address which will be called when doing "call ExitProcess" - and this is 8301A110:

```
8301A110    68 F8D4F8BF            PUSH KERNEL32.ExitProcess
8301A115   -E9 B0B6F73C            JMP KERNEL32.BFF957CA
```

Therefore this corresponds to the "debug" case. Now it's important to look at how our call to ExitProcess has got compiled to understand the whole process:

```
CALL 00401017 ; call to kernel32.ExitProcess (redirected thru the IAT)
CALL 0040101D ; call to user32.MessageBoxA

00401017: JMP DWORD PTR DS:[40304C] ; [40304C] = 8301A110
0040101D: JMP DWORD PTR DS:[403054] ; [403054] = 8301A120
...

8301A110    68 F8D4F8BF            PUSH KERNEL32.ExitProcess    ; we really call
                                                                  this address
8301A115   -E9 B0B6F73C            JMP KERNEL32.BFF957CA

8301A120    68 2E41F5BF            PUSH USER32.MessageBoxA
8301A125   -E9 A0B6F73C            JMP KERNEL32.BFF957CA
```

So it has resulted into:

```
      Name of the Api -----------> 'ExitProcess',0
      Door = 8301A115 -----------> PUSH KERNEL32.ExitProcess; JMP KERNEL32.BFF957CA

      Name of the Api -----------> 'MessageBoxA',0
      Door = 8301A120 -----------> PUSH USER32.MessageBoxA; JMP KERNEL32.BFF957CA
```

The IAT, holding the "doors" is:

```
40304C: 8301A110 ; door to ExitProcess
403054: 8301A120  ; door to MessageBoxA
```

As you see, in this case the calls have been compiled into something slightly different: "call xxxxxxxxh", this is just another possibility. You can also notice that the offsets where the doors have been kept are consecutive, this hasn't to be always this way.
Is everything clear until now? Next example...

2) Example 3. WORDPAD.EXE: Let's have a look at the data directories of Wordpad.

```
        RVA   virtual size

import 1CB40     DC
IAT    1000      E90
```

Now, we go to the sections and see what section is the import table in, the answer is into the .txt section. The .txt, standard name for the code section, starts at raw offset 1000 and has raw size 1E000. From this we can conclude that the import table can be found, into the wordpad.exe file, at raw offset 1CB40 (cos it starts CB40 bytes after the RVA of the .txt section).

So, we go to that RVA into the debugger and find this:

```
0101CB40    . 74CD0100             DD 74CD0100      ;  Struct 'IMAGE_IMPORT_DESCRIPTOR'
0101CB44    . 1A701738             DD 3817701A
0101CB48    . 0000405F             DD OFFSET MFC42.#1340
0101CB4C    . ACDA0100             DD 0001DAAC
0101CB50    . 58110000             DD 00001158
```

Let's examine it:

First is the OriginalFirstThunk, points to `IMAGE_BASE` + 74CD0100 (cos it's an RVA). There we can find the following:

```
0101CD74  2D 03 00 80 2F 01 00 80 55 01 00 80 8E 02 00 80
0101CD84  82 14 00 80 FE 09 00 80 70 11 00 80 0C 19 00 80
```

What's that "2D 03 00 80"? NOT an RVA! as i mentioned above, the dword starts by an 80 to mark that the import is done by ordinal. So this means we're importing the API number 032D from that DLL. The same applies to the rest of them.

The second and third offsets contain not-interesting (for us) information about this dll, next is the pointer to the name of the DLL, MCFC42.DLL.

Finally, our favourite one!, the FirstThunk. This has the RVA 00001158, once aligned to the image base it points to the following information:

```
01001158 >4F 85 41 5F 76 D5 40 5F EC F5 40 5F 4F 8E 41 5F
01001168 >C7 D5 40 5F 71 D9 40 5F B6 C7 40 5F A6 3D 4D 5F
```

If we analyze the contents of that addresses:

```
01001158: 4F 85 41 5F          ; address of MCF42.#813
0100115C: 76 D5 40 5F          ; address of MCF42.#303
...
```

So, in this case the FirstThunk holds the addresses of the APIs directly inside MCF42.DLL. The calls to this APIs are done as follows (jumping to the value given by the IAT):

```
CALL 01014CD0                  ; call MCF42.#303
...
01014CD0    $-FF25 5C110001    JMP DWORD PTR DS:[&MFC42.#303]  ;  MFC42.#303
```

As you see, in the imports table we had the address inside MCF42.DLL of the APIs we need to import. But later, the calls are compiled like call xxxxxxxxh instead of call [xxxxxxxxh]. This means that at xxxxxxxxh we need to have a jump to the door. Obviously, if we'd have call [xxxxxxxxh] then at xxxxxxxxh it'd a pointer to the door.

3) Example 4. CDPLAYER.EXE:

```
 RVA       virtual     size

Import      10000      C8
IAT         103CF      320
```

The .idata section starts at offset 10000 and has raw size 2000 (its virtual address is 10000). Now, we load the file and go to 10000:

```
00410000  60 01 01 00 B3 C2 1F 37 FF FF FF FF 70 09 01 00
00410010  80 04 01 00 04 02 01 00 CA 59 3B 37 FF FF FF FF
00410020  DA 0F 01 00 24 05 01 00 A0 03 01 00 CF 59 3B 37
00410030  FF FF FF FF 66 10 01 00 C0 06 01 00 04 01 01 00
00410040  72 C2 3E 35 FF FF FF FF 34 12 01 00 24 04 01 00
00410050  F4 00 01 00 CA 59 3B 37 FF FF FF FF 50 12 01 00
00410060  14 04 01 00 EC 01 01 00 CF 59 3B 37 FF FF FF FF
00410070  B0 12 01 00 0C 05 01 00 98 03 01 00 CA 59 3B 37
00410080  FF FF FF FF CE 12 01 00 B8 06 01 00 DC 00 01 00
00410090  CA 59 3B 37 FF FF FF FF 2A 13 01 00 FC 03 01 00
004100A0  EC 03 01 00 CF 59 3B 37 FF FF FF FF 6E 13 01 00
004100B0  0C 07 01 00 00 00 00 00 00 00 00 00 00 00 00 00
004100C0  00 00 00 00 00 00 00 00 00 00 00 00
```

This are all the headers, `IMPORT_IMAGE_DESCRIPTORs`, for the dlls. We're gonna examine the first one, as usual.

```
Name of DLL: KERNEL32.DLL
```

OriginalFirstThunk: 60 01 01 00, that leads to this:

```
00410160  F4 08 01 00 12 09 01 00 4C 09 01 00 3C 09 01 00
00410170  30 09 01 00 E8 08 01 00 DE 08 01 00 D4 08 01 00
```

This points to the following API names:

```
WritePrivateProfileSectionA
WritePrivateProfileStringA
...
```

And if we have a look at the FirstThunk, 80 04 01 00, we'll see that it holds this values:

```
00410480 >E8 C6 ED 82 F8 C6 ED 82 08 C7 ED 82 18 C7 ED 82
00410490 >28 C7 ED 82 38 C7 ED 82 48 C7 ED 82 58 C7 ED 82

82EDC6E8    68 7B17FABF             PUSH KERNEL32.WritePrivateProfileSectionA
82EDC6ED   -E9 D8900B3D             JMP KERNEL32.BFF957CA

82EDC6F8    68 C378F7BF             PUSH KERNEL32.WritePrivateProfileStringA
82EDC6FD   -E9 C8900B3D             JMP KERNEL32.BFF957CA
...
```

Finally we need to examine the IAT, a call to WritePrivateProfileSectionA gets loaded as follows:

```
CALL DWORD PTR DS:[410480] ; call kernel32.WritePrivateProfileA
...
00410480: >E8 C6 ED 82        ; door to kernel32.WritePrivateProfileA
```

Ok, we're gonna experiment a little bit with the IAT. Take each one of the examples and set the IAT values to RVA = 0, Virtual Size = 0:

```
         runs ok with zeroes

Notepad:      yes
Wordpad:      yes
Main:         yes (it already had them set to zeroe)
CDplayer:     yes
```

Therefore the IAT is redundant. Now, change the values for main.exe to anything else and you will see it still loads correctly. Have a look at main.exe to find out if there're some difference, you will see that there's nothing interesting excluding that the address of the doors have changed (this corresponds to the memory available for allocating a heap with the doors).

So, we conclude the following: The IAT information at the directories of the PE header is totally irrelevant, what really matters is the ImportsTable (check for win2k, i haven't it).

III. Rebuilding the whole imports table by hand

In this section, we'll study how to add imports to a program by hand. In my case, i was interested in the next problem:

Input: Names of all APIs and address into the IAT to write the offset of the door.
Output: A full working app

Why this problem?

First i need to tell you about how some anti-cracking programs protect the apps. Imaging you protect some app with the anti-cracking tool AC: AC will encrypt the imports table of your app and will construct its IAT at startup. At the beginning, you can see that the AC-protected app has changed the RVA and VirtualSize of your app and has set it to point to the anti-cracking program. The most frequent APIs imported by the anti-cracking tools are GetModuleHandleA and GetProcAddress, so you will see the ImportsTable with only this two APIs very frequently.

Let's review how this ImportsTable reconstruction could be done. For example, if the protected app makes use of CreateFileA and ExitProcess then we could see:

AC keeps CRC32("CreateFileA"), CRC32("ExitProcess") when it protects your app

When you run the AC-protected program:

- Checks if it's running debugged, if so terminates.
- Looks for an API having CRC32(name) = CRC32(CreateFileA) and writes its address to the IAT
- Looks for an API having CRC32(name) = CRC32(ExitProcess) and writes its address to the IAT

As you see, if we dump the executable we don't have too much information. But, fortunately, in my case i'm able to hook the IAT filling process and i know exactly which API is stamped and where. Indeed, i could also dump an almost correct imports table but future versions of the protector would make this useless (you simply need to overwrite all the imports table) and so i've decided to use a harder to beat method.

Some tips:

Let me tell you some of the usual tricks applied by the anti-cracking tools for protecting the imports:

```
    Delete the OriginalFirstThunk
    Delete all the strings with the API names
```

I also take profit to comment you that, most protectors, forget to set the hints to zeroe making our work much simpler (hints are right 99% of the times). All this stuff could be enough for a collection of tutorials, doubtless it shall be... :D

A. Rebuilding: How To

There's a subtle difference between Win9x, WinXP and Win2k. The loader in Win2k crashes if we have an empty ImportsTable, it DOES need to have kernel32.dll imported by your app. Interestingly enough, this doesn't happen to Win9x or WinXP and therefore we can load into our debugger any app with an empty import table (Win9x and WinXP automatically load kernel32.dll before running your app). The problem of crashing importless apps will be treated in section 6, look there first if you have Win2k running in your machine, otherwise....

We're gonna reconstruct the imports of an importless program so we first have to get such a thing. Go to your PE editor and open main.exe, Do you see a nice .idata section?. REMOVE IT!.

Ok, now you have the target. We'll add the following APIs to our imports:

```
Kernel32: ExitProcess, IsDebuggerPresent
User32: MessageBoxA, IsWindow
Advapi32: RegOpenKeyExA, RegQueryValueExA, RegCloseKey
```

Adding more imports is just a matter of patiente. Ok, try to load the app with your debugger... it crashes...Well, change the values of the ImportsTable into the PE header and set them to zeroe. Now try again... works!, the app has been loaded (this only happens in Win9x and WinXP, not Win2k. If you run the later refer to the section "importless app").

```
; target's Entry Point (after removing the imports):

00401000  ModuleEntryP  $ 6A 00                 PUSH 0
00401002                . 6A 00                 PUSH 0
00401004                . 68 00204000           PUSH MAIN.00402000
00401009                . 6A 00                 PUSH 0
0040100B                . E8 0D000000           CALL MAIN.0040101D
00401010                . 6A 00                 PUSH 0
00401012                . E8 00000000           CALL MAIN.00401017


00401017                $ FF25 4C304000         JMP DWORD PTR DS:[40304C]
0040101D          $ FF25  54304000 JMP DWORD PTR DS:[403354]
```

Let's start by observe that the offsets 40304C and 403054 no longer exist, cos they were into the removed section. This are the steps you must follow:

1) Look for room for your imports. In this case, we have plenty of it into the .txt section and so we'll place our fake import section below those jumps.
2) Add the strings with the names of all DLLs and APIs you need. There's no order you have to follow here, i put them grouped by DLL, see below how i did it (observe that all hints are set to zeroe, no need to do extra work):

```
00401030  4B 45 52 4E 45 4C 33 32 2E 44 4C 4C 00 00 00 49  KERNEL32.DLL...I
00401040  73 44 65 62 75 67 67 65 72 50 72 65 73 65 6E 74  sDebuggerPresent
00401050  00 00 00 45 78 69 74 50 72 6F 63 65 73 73 00 55  ...ExitProcess.U
00401060  53 45 52 33 32 2E 44 4C 4C 00 00 00 49 73 57 69  SER32.DLL...IsWi
00401070  6E 64 6F 77 00 00 00 4D 65 73 73 61 67 65 42 6F  ndow...MessageBo
00401080  78 41 00 41 44 56 41 50 49 33 32 2E 44 4C 4C 00  xA.ADVAPI32.DLL.
00401090  00 00 52 65 67 4F 70 65 6E 4B 65 79 45 78 41 00  ..RegOpenKeyExA.
004010A0  00 00 52 65 67 51 75 65 72 79 56 61 6C 75 65 45  ..RegQueryValueE
004010B0  78 41 00 00 00 52 65 67 43 6C 6F 73 65 4B 65 79  xA...RegCloseKey
004010C0  41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

3) Add the first thunk: Theoretically, the first thunk can be separated into different pieces one for each DLL. We don't need to do such a thing, this is our First Thunk holding the RVAs of all API names (load the app with a debugger):

```
; APIs from kernel32 (the FirstThunk consists of all the RVAs to the names of the APIs)

                  ; RVA to:
DD 0000103D       ; 0,0,'IsDebuggerPresent',0
DD 00001051       ; 0,0,'ExitProcess',0
DD 0              ; end of list of kernel32.dll

; APIs from user32

DD 0000106A       ; 0,0,'MessageBoxA',0
DD 00001075       ; 0,0,'IsWindow',0
DD 0              ;

; APIs from Advapi32

DD 00001090       ; 0,0,'RegOpenKeyExA',0
DD 000010A0       ; 0,0,'RegQueryValueExA',0
DD 000010B3       ; 0,0,'RegCloseKey',0
DD 0              ;
```

We can place it where we swant, for example we're gonna place it below the strings, this results into:

```
004010B0  78 41 00 00 00 52 65 67 43 6C 6F 73 65 4B 65 79  xA...RegCloseKey
004010C0  41 00 3D 10 00 00 51 10 00 00 00 00 00 00 6A 10  A
004010D0  00 00 75 10 00 00 00 00 00 00 90 10 00 00 A0 10
004010E0  00 00 B3 10 00 00 00 00 00 00 00 00 00 00 00 00
```

Observe that all values are kept in indian order.

4) Adding the `IMPORT_IMAGE_DESCRIPTORS`:
We need 4 `IMPORT_IMAGE_DESCRIPTORS`, the last one - used as terminating header - being nulled. Since we're a bit lazy, Aren't?, we're gonna set OriginalFirstThunk = FirstThunk and also Timestamp = 0 and ForwarderChain = 0. Our `IMPORT_IMAGE_DESCRIPTORS` are the following:

```
; KERNEL32.DLL

DD 000010C2       ; OriginalFirstThunk (is an RVA)
DD 0              ; unused
DD 0              ; unused
DD 00001030       ; RVA to the name of the DLL
DD 000010C2       ; FirstThunk (agreeing with the OriginalFirstThunk)

; USER32.DLL

DD 000010CE       ; OriginalFirstThunk
DD 0              ;
DD 0              ;
DD 0000105F       ; RVA to the name of the DLL
DD 000010CE       ; FirstThunk

; ADVAPI32.DLL

DD 000010DA       ; OriginalFirstThunk
DD 0              ;
DD 0              ;
DD 00001083       ; RVA to the name of the DLL
DD 000010DA       ; FirstThunk

; TERMINATING IMAGE_IMPORT_DESCRIPTOR

DD 0, DD 0, DD 0, DD 0, DD 0
```

For me, pasting it into the hex resulted into this:

```
004010F0                            C2 10 00 00 00 00 00 00
00401100  00 00 00 00 30 10 00 00 C2 10 00 00 CE 10 00 00
00401110  00 00 00 00 00 00 00 00 5F 10 00 00 CE 10 00 00
00401120  DA 10 00 00 00 00 00 00 00 00 00 00 83 10 00 00
00401130  DA 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

5) Redirecting the imports table to the new one
   Our new imports table starts at RVA 000010F8 and has size 3*5*4 = 60 = 3C bytes. Open the app with your PE editor and update those values, next load it. It works :D
   Tip: What happens if you incorrectly set up the imports table?, How can you find where's the mistake? I use the next trick, that i think might be useful for you: When is a ”loading” error you see a message like ”the app XXX.exe is linked to a not existing API inside ADVAPI32.DLL”. Now, you know that the error is at ADVAPI32 but if you have imported 50 APIs from there it can be a nightmare to examine one by one. To shorten this process, i recommend to do the following into the FirtThunk:

```
; ADVAPI32.DLL

DD 000031A0       ; RegOpenKeyExA
DD 00009721       ; RegOpenKeyA
...               ;
DD 00000000       ; Overwrite with zeroes !!!
DD 0000734B       ; RegQueryValueExA
...
DD 0000AAFF       ; RegCloseA
```

   This way the loader will only load the APIs given till the zeroe we have inserted, if it crashes we know the mistake is before the overwritten one otherwise after. Now, it's evident how to proceed to locate the exact API that has failed in a few steps, regardless of the length of the list.

6) The final touch: Diverting the calls.
   Of course, once we have created our ImportsTable we have to use it to do redirect the calls to our IAT, the patchs are as follows:

```
call IsDebuggerPresent = call [004010C2]
call ExitProcess       = call [004010C6]
```

   As you see, 004010C2 is the offset where we have the RVA to IsDebuggerPresent and 004010C6 the one holding ExitProcess. If you recall, we commented that the pointers to the names of the APIs where substituted by the addresses of the APIs by the Windows loader:

```
004010B7                                    10 11 03 83 20
004010C7   11 03 83 00 00 00 00 30 11 03 83 40 11 03 83 00
004010D7   00 00 00 50 11 03 83 60 11 03 83 70 11 03 83

83031110                68 F646F9BF             PUSH KERNEL32.IsDebuggerPresent
83031115               -E9 B046F63C             JMP KERNEL32.BFF957CA

83031120                68 F8D4F8BF             PUSH KERNEL32.ExitProcess
83031125               -E9 A046F63C             JMP KERNEL32.BFF957CA

83031130                68 A54EF5BF             PUSH USER32.IsWindow
83031135               -E9 9046F63C             JMP KERNEL32.BFF957CA

83031140                68 2E41F5BF             PUSH USER32.MessageBoxA
83031145               -E9 8046F63C             JMP KERNEL32.BFF957CA

...
```

IV. Reallocating the imports table

Other interesting problem is how to add, in a neat way, several imports to an app that runs perfectly. This problem arises when we wanna patch an app but we don't have all the APIs we need (this is the most common case).

For example, open Notepad with your debugger (the imports table is at RVA 6000). It imports APIs from SHELL32, KERNEL32, USER32, ADVAPI32,... but, as you see, it doesn't have FlushViewOfFile (KERNEL32) and i wanna use it. The approach to do it is not unique, but the simplest one - in my opinion - is the next:

case 1. We do NOT have enough space at the imports table to add another `IMPORT_IMAGE_DESCRIPTOR`

1) Move all the `IMPORT_IMAGE_DESCRIPTORs` somewhere else (where you have more room, i liked 000061E0)
2) Update the Directory of the PE header to the new ImportsTable's RVA.
3) Round up the size of the section where you've put the new ImportsTable so everything is mapped in memory (in my case i had to increase the VirtualSize of the .idata section to 1000h)
4) Check it works, does it? Congratulations!, you've been upgraded to case 2. Doesn't? Check the injected descriptors are mapped in memory and check too that the RVA of the ImportsDescriptor is right...

Note: the `IMPORT_IMAGE_DESCRIPTORs`, FirstThunk and OriginalFirstThunk only have RELATIVE ADDRESSES, this means you can cut and paste them wherever you want (taking in account that has to mapped in memory) and simply changing the RVA of the ImportsTable will make the app to work perfect.

case 2. We DO have enough space after the imports table to add another `IMPORT_IMAGE_DESCRIPTOR`

Mind that you always have to put a nulled `IMPORT_IMAGE_DESCRIPTOR`. In this case, simply add a new `IMPORT_IMAGE_DESCRIPTOR` for kernel32.dll with the information neccesary to link FlushViewOfFile. Let's review how to do the rest of the process in our example:

1) Add the string for the API FlushViewOfFile after the rest of the strings. Note you don't need to add the name of the DLL cos this was already included by Notepad somewhere else.

```
00006DD8: 41 00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00  A.Advapi32.dll..
00006DE8: 00 46 6C 75 73 68 56 69 65 77 4F 66 46 69 6C 65  .FlushViewOfFile
00006DF8: 00
```

2) Add the FirstThunk

```
; KERNEL32.DLL (for adding FlushViewOfFile)
00006EE0: DD 00006DE7 ; RVA to 0,0,'FlushViewOfFile',0
00006EE4: DD 0 ;
```

3) Add the new `IMPORT_IMAGE_DESCRIPTOR` after the last, not nulled, one

```
; newly added IMPORT_IMAGE_DESCRIPTOR
00006E88: DD 00006EE0 ; OriginalFirstThunk
00006E8B: DD 0 ;
00006E90: DD 0 ;
00006E94: DD 0000679C ; RVA to 'KERNEL32.DLL',0
00006E98: DD 00006EE0 ; FirstThunk
```

4) Nothing left to be done, Notepad should work perfectly. Moreover, if you open it with a debugger then you should see that the address of FlushViewOfFile has been correctly set at `IMAGE_BASE` + 00006EE0. Now, you can insert your snippets calling this API.

V. Importless Applications

A. Creating an app without any imported API that works Win9x and WinXP

In this section we'll review how to compell to the app to load kernel32. This pretty interesting question was posted to me by Pegasus - thanks, Pegasus!.

As we mentioned above, importless app don't load in Win2k cos you always need to have mapped kernel32 into your app's memory space, this is a bit of an inconvinient. We'll see now how to make an importless app working well both in Win9x and WinXP.

The next program does the following:

1) Locate the image base of kernel32
2) Locate LoadLibraryA (by examining the exports of kernel32)
3) Load user32.dll
4) Locate MessageBoxA (by examining the exports of user32)
5) Show a message box saying hello
6) Locate ExitProcess
7) Exit(0)

The procedure to locate an API into a DLL, given its name and the DLL's image base, is based on examining the exports of the DLL (by name), you can follow very easily what it does with the only help of a debugger and a guide to the exports.

```
;-----------------------------------------------------------------
;                                    app.asm
;-----------------------------------------------------------------
; To compile this file with TASM make a .BAT with the following:
; @echo off
; tasm32 /ml /m2 /w2 app,,;
; tlink32 /r  /Tpe /aa /c /v app,app,, import32.lib
;-----------------------------------------------------------------

.386p
.model flat, stdcall

.DATA


      szTitle     db 'Not a single import in Win2k?',0
      szText      db 'YOU MUST BE JOKING!',0

      szMessageBoxA     db 'MessageBoxA',0
      szUser32          db 'User32',0
      szExitProcess     db 'ExitProcess',0
      szLoadLibraryA    db 'LoadLibraryA',0

      _MessageBoxA      dd 0 ; address of MessageBoxA
      _user32           dd 0 ; handle to user32
      _kernel32         dd 0 ; handle to kernel32




  .CODE

Main:


      mov eax, [esp]          ; at the very beginning the
                                first dword on the stack
                              ; contains a pointer inside
                                kernel32
      or eax, 00000FFFh       ; the image base has to be a
                                multiple of the memory alignment
      xor eax,00000FFFh       ;

compare:
      cmp word ptr [eax], 'ZM'
      je kernel32_found
      sub eax, 1000h
      jmp compare

kernel32_found:

      mov dword ptr [_kernel32], eax
      mov esi, offset szLoadLibraryA
      call GetFunctionAddress

      push offset szUser32            ;
      call eax                        ; Load user32.dll
                                      ; in return, eax = image base of user32

      mov esi, offset szMessageBoxA
      call GetFunctionAddress

      push 0
      push offset szTitle
      push offset szText
      push 0
      call eax                        ; call MessageBoxA

      mov eax, [_kernel32]
      mov esi, offset szExitProcess
      call GetFunctionAddress
      push 0
      call eax ; call ExitProcess


;----------------------------------------------------------------------------
;                              GetFunctionAddress
;----------------------------------------------------------------------------
; Input parameters:
;      esi = offset of a zeroe terminated string with the name of the Api.
;      eax = image base of the dll where the API "lives"
; Returns:
;      eax = address of desired API
;----------------------------------------------------------------------------


GetFunctionAddress PROC

        mov     ebx, [eax+3Ch]                          ; pointer to pe header
        add     ebx, eax
        add     ebx, 120
        mov     ebx, [ebx]
        add     ebx, eax                                ; EBX = Export Address

        xor     edx, edx
        mov     ecx, [ebx+32]
        add     ecx, eax
        push    esi
        push    edx


CompareNext:
        pop     edx
        pop     esi
        inc     edx
        mov     edi, [ecx]
        add     edi, eax
        add     ecx, 4
        push    esi
        push    edx

CompareName:
        mov     dl, [edi]
        mov     dh, [esi]
        cmp     dl, dh
        jne     CompareNext
        inc     edi
        inc     esi
        cmp     byte ptr [esi], 0
        je      GetAddress
        jmp     CompareName
GetAddress:
        pop     edx
        pop     esi
        dec     edx
        shl     edx, 1
        mov     ecx, [ebx+36]
        add     ecx, eax
        add     ecx, edx
        xor     edx, edx
        mov     dx, [ecx]
        shl     edx, 2
        mov     ecx, [ebx+28]
        add     ecx, eax
        add     ecx, edx
        add     eax, [ecx]

        ret

GetFunctionAddress ENDP

End   Main                  ; End of code
```

B. Compelling the loader to map a dll

If you're running Win9x or WinXP you can see that that app works and, as i promised, it hasn't a single imported API. Now, the problem becames to compell the loader to map kernel32 by importing ONLY the DLL. For this, we're gonna add a fake ImportsTable as follows:

```
000006B0: C8 03 CA 03 01 C3 00 00 4B 45 52 4E 45 4C 33 32  KERNEL32.DLL....
000006C0: 2E 44 4C 4C 00 00 00 00 00 11 00 00 00 00 00 00  ................
000006D0: 00 00 00 00 B0 10 00 00 00 11 00 00 00 00 00 00  ................
000006E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Observe that this is done with the only help of an hex-editor, cos we're working in Win2k. Now, let me explain what's going on: As you see, we've added the name of the DLL but not the name of any API (cos we don't want to import any API, right?). Inmediately after the name of the DLL i've placed the fake ImportsTable, you can see that OriginalFirstThunk = FirstThunk and that both are pointing to a place that has a zeroe dword (once mapped in memory). Therefore, when the loader has loaded KERNEL32 and goes to load each of the APIs we request it simply finds that no API is asked and so terminates (but doesn't crash!). Needless to say, the name of the DLL simply points to "KERNEL32.DLL".

To compute the RVA from the raw offset and viceversa you can use some of the many available tools or simply take in account where's gonna be mapped the .code section (check with a PE editor).

C. A new way of locating kernel32 for Win9x

It's amazing that if you have into the .exe file, for a given descriptor, ForwarderChain = Timestamp = 0 then the loader stamps into the Timestamp the one of the DLL and into the ForwarderChain the image base of the DLL =¿ therefore this is a way - to my few - of locating the image bases of all your DLLs. Pretty simple, ain't?.

That's all from me... (for now!)

Kind regards, Eduardo Labir