```
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
 □□    ____                    __      __          □□□□
□□   /  _/_ _  __ _  ___  ____/ /____ _/ /         □□ □□□
□□  _/ //  ' \/  ' \/ _ \/ __/ __/ _ `/ /          □□ □ □
□□ /___/_/_/_/_/_/_/\___/_/  \__/\_,_/_/           □□ □ □
□□   ____                         __        __     □□ □ □
□□  / __ \___ ___ _______ ___  ___/ /__ ____  / /____□□ □ □
□□ / /_/ / -_|_-</ __/ -_) _ \/ _  / _ `/ _ \/ __(_-<□□ □ □
□□/_____/\__/___/\__/\__/_//_/\_,_/\_,_/_//_/\__/___/□□ □ □
□□                                                   □□ □ □
□□      Web: http://www.ImmortalDescendants.org      □□ □ □
□□                Author: [yAtEs]                    □□ □ □
□□                  Date: 08/JUN/00                  □□ □ □
□□        Topic: Understanding Import Tables         □□ □ □
□□                                                   □□ □ □
□□             Level: Beginner                       □□ □ □
□□                                                   □□ □ □
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □ □
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □
    □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

===
PUBLISHER'S NOTE: An additional .exe file associated with this essay can
be found here:
www.reverse-engineering.info/files/blah.exe
===

There are various documents around explaining PE, but i decided
to write about the import table in detail as it is usally an interest
to crackers.

The best way for me to explain this is to work with an example, this way
you can following it through and it shall all make sense ;) , i've chosen
a small exe that i just pulled of my HD, it was compiled with TASM and has
a few imports so its just right.

Ok so lets start, first we must find the import table, there is a pointer to it
in the PE-Header at offset 80, so load our exe into a hex editor, Hex Workshop
ownz because you can select a chuck of bytes and navigate easy through a file,
well anyway we need to find the start of the PE header, this is quiet simple
because it always starts with PE,0,0. at offset 100 you will see this, the real
way to work the PE-Header offset out is to goto offset 0x3C in any win32 exe and
theres a pointer to it, 0001 0000 , because address are stored backwards this
looks like , 00000100 which is offset 100 like i said before, anyway now we can
get our import pointer from the PE, 100+80=180 goto offset 180 we see 0030 0000
so this is 00003000, this means our import table is at 3000 in memory we must convert
this into a disk offset.

now because the import table usally starts at the beginning of a section we can use a PE editer to view our section virtual offsets then look for 3000 and find the raw offset, simple pimple.

```
-CODE  00001000  00001000  00000200  00000600
-DATA  00001000  00002000  00000200  00000800
.idata 00001000  00003000  00000200  00000A00
.reloc 00001000  00004000  00000200  00000C00
```

after a quick scan over that list, theres our .idata VA 3000, the raw offset is A00 so 3000-A00=2600 we should remember 2600h for when converting other offsets later, if you don't see a virtual offset for your import pointer then look for the closest section.

ok goto offset 0xA00 now we have reached what we call the IDDs IMAGE_IMPORT_DESCRIPTORs, these are a set of 5 dwords which contain information about 1 DLL, the iids are terminated
by a full nulled iid

```
***************************************************************************
(iid) IMAGE_IMPORT_DESCRIPTOR structure..
      OriginalFirstThunk, TimeDateStamp, ForwarderChain, Name, FirstThunk

  OriginalFirstThunk
        An RVA (32 bit) pointing to a 0-terminated array of RVAs to
        IMAGE_THUNK_DATAs, each describing one imported function. The
        array will never change.

    TimeDateStamp
        A 32-bit-timestamp that has several purposes. Let's pretend that
        the timestamp is 0, and handle the advanced cases later.

    ForwarderChain
        The 32-bit-index of the first forwarder in the list of imported
        functions. Forwarders are also advanced stuff; set to all-bits-1
        for beginners.

    Name
        A 32-bit-RVA to the name (a 0-terminated ASCII string) of the
        DLL.

    FirstThunk
        An RVA (32 bit) to a 0-terminated array of RVAs to
        IMAGE_THUNK_DATAs, each describing one imported function. The
        array is part of the import address table and will change.
***************************************************************************
```

Ok, that make sense? lets look at how may idds we have, here they are taken from 0xA00

```
    3C30 0000      /   0000 0000   /  0000 0000   / 8C30 0000 / 6430 0000
{OrignalFirstThunk} {TimeDateStamp} {ForwardChain} {Name}      {First Thunk}

    5C30 0000      / 0000 0000     /  0000 0000   / 9930 0000 / 8430 0000
{OrignalFirstThunk} {TimeDateStamp} {ForwardChain} {Name}       {First Thunk}

    0000 0000 / 0000 0000 / 0000 0000 / 0000 0000 / 0000 0000
```

The third one is the terminator, as we know each iid contains information about
1 DLL and since we have 2 iids we can assume our program imports 2 DLLs, even though
i bet you can guess what these are lets find out.

The 4th dword of an iid is the Name, this will tell us the DLL name, so the name
for the first iid would be

8C30 0000 - reversed to form an address is 0000308C, then -2600 to get our
raw offset, 308C-2600=A8C

goto offset A8C, and.. it says KERNEL32.dll, woo :)

ok so now we want to find the functions that have been imported from the
kernel library, go back to our first iid

The FirstThunk contains pointers to the ascii names of these functions, the Orignal
thunk is a copy of this firstthunk, the orignal thunk is just for backup and some
programs don't even have this, so we always look to the firstthunk which is initialised
during runtime.

the first thunk for our kernel iid is like so

6430 0000 - reversed to form an address 00003064 -2600 = A64
at 0xA64 is our IMAGE_THUNK_DATA, a thunk is an array of pointers
so goto 0xA64 you will set an array of addresses terminated by one
zero one.

```
DE30 0000 / EA30 0000 / F630 0000 / 0000 0000
```

normally in a full program you would have loads of these, we have only 3 functions
from the kernel library here, lets take a look at the first 2

DE30 - 30DE-2600=ADE @ 0xADE is ReadFile
EA30 - 30EA02600=AEA @ 0xAEA is WriteFile

you may notice there is 2 bytes before the function name, this is the hint
and is used as a reference

ok it is that simple, you should of picked it up by now, lets go back to 0xA00
our iids, look at the second dll

```
    5C30 0000      / 0000 0000     /  0000 0000   / 9930 0000 / 8430 0000
{OrignalFirstThunk} {TimeDateStamp} {ForwardChain} {Name}       {First Thunk}
```

lets find its name

9930 - 3099-2600 =A99 @ 0xA99 is USER32.dll

now lets move to the first thunk and extract the first two functions

8430 - 3084-2600=A84 0xA84 = Image_Thunk_Data{08310000}

oh well theres only 1 function so we won't extract the first 2 ;)

0831 - 3108-2600=B08 @ 0xB08 is MessageBoxA

There thats it, now you can go and play with your own exe files

summary
=======

Get the IMAGE_DIRECTORY_ENTRY_IMPORT from PE+80 which points to IMAGE_IMPORT_DESCRIPTORs which contain our Name and FirstThunk data for each DLL, the Forward Chain and TimeStamp are usally 0

during runtime a program uses GetProcAddress which takes the ascii name of an function as a parameter, this then retieves the API address in memory and writes it into the import table in memory. If you dump a file sometimes you may notice you have an initialised FirstThunk, for e.g. the initialised API GetProcAddress for my win98 machine would look like AE6DF7BF , all kernels looking like - xxxxF7BF on win98, if you see this in an import table, you can restore it using the orignal thunk or rebuilding the PE.

Hopefully i have taken you through how it all works, i'm no expert so if theres any mistakes
please let me know.

gREETz: to everyone i know

[yAtEs]
"Keep it locked, keep it hardcore. Roots 'n' phuture. Peace."

Republished - 12th November 2007 – Robert Yates

Republished - 12th November 2007 – Robert Yates