

Debugging Windows Applications with IDA Bochs Plugin

Copyright 2010 Hex-Rays SA

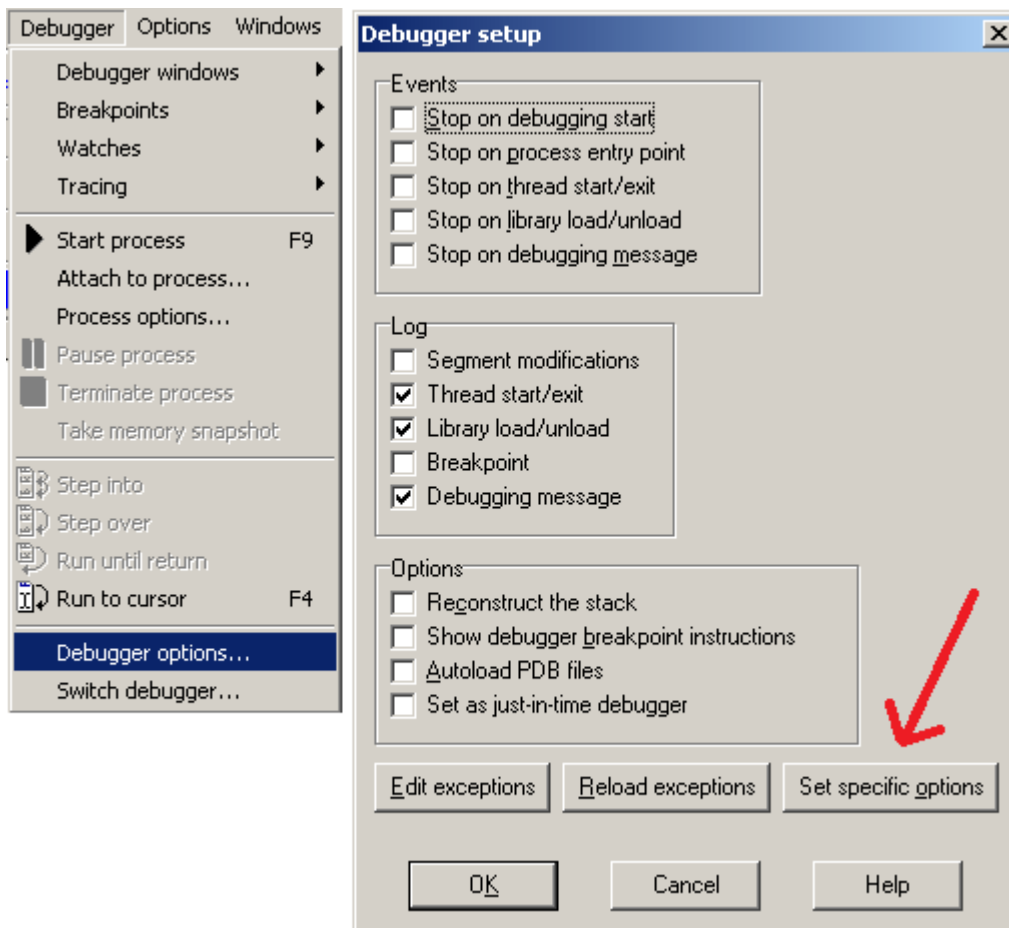
Quick overview:

The IDA Bochs debugger plugin allows malware researchers to debug malicious code in a safe/emulated environment. This is implemented using the open source x86 emulator Bochs.

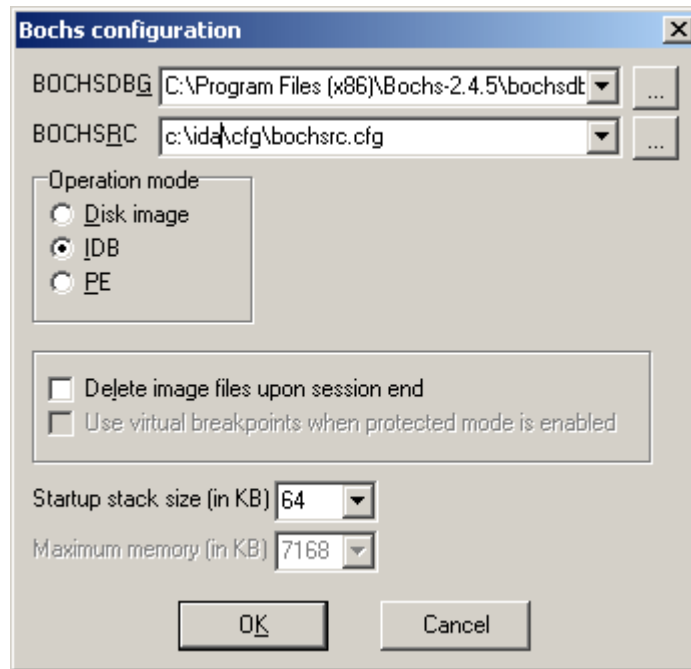
To get started, you need to install supported Bochs version (v2.3.7 or above) from <http://bochs.sourceforge.net/>

After installing it, make sure you « Switch Debugger » and select the Bochs debugger.

IDA can automatically detect where Bochs is installed, but if it fails it is possible to reconfigure it in the Debugger Setup / Specific Debugger options:



In this screen, « Debugger specific options », we configure the Bochs plugin:



BOCHSDBG

This parameter specifies the path to the bochsdbg.exe executable. IDA tries to guess it by looking at the BXSHARE environment variable or by checking the system registry for Bochs registry keys

BOCHSRC

This is the path to the Bochs configuration file template. It contains special variables prefixed with "\$". These variables should not be modified or changed by the user, they are automatically filled by the plugin. Other entries in this template can be modified as needed.

Operation mode

The user can choose between the following three operation modes:

- Disk image: Debug a complete operating system (use IDA Pro as an interface to the Bochs debugger)
- IDB: Debug the contents of the database (or just a selection)
- PE: Debug an MS Windows PE files

Delete image files upon session end

If enabled, IDA will automatically delete the Bochs disk images used during the debugging session (this option only applies to IDB and PE operation modes).

In the IDB operation mode, the Bochs plugin tries to find a previously created image, verifies that it corresponds to the database and uses it as is. Unchecking this option in this case (IDB operation mode) can speed up launching the debugger.

Debugging a Bochs disk image:

With the disk image loader it is possible to debug any Bochs disk image.

First prepare a virtual machine:

- myvm.bochsrc: This file contains the Bochs configuration, such as the disk image file name, cdrom config, network card, bios file, etc...
- myvm_diskimage.bin: This is the actual disk image file containing the operating system

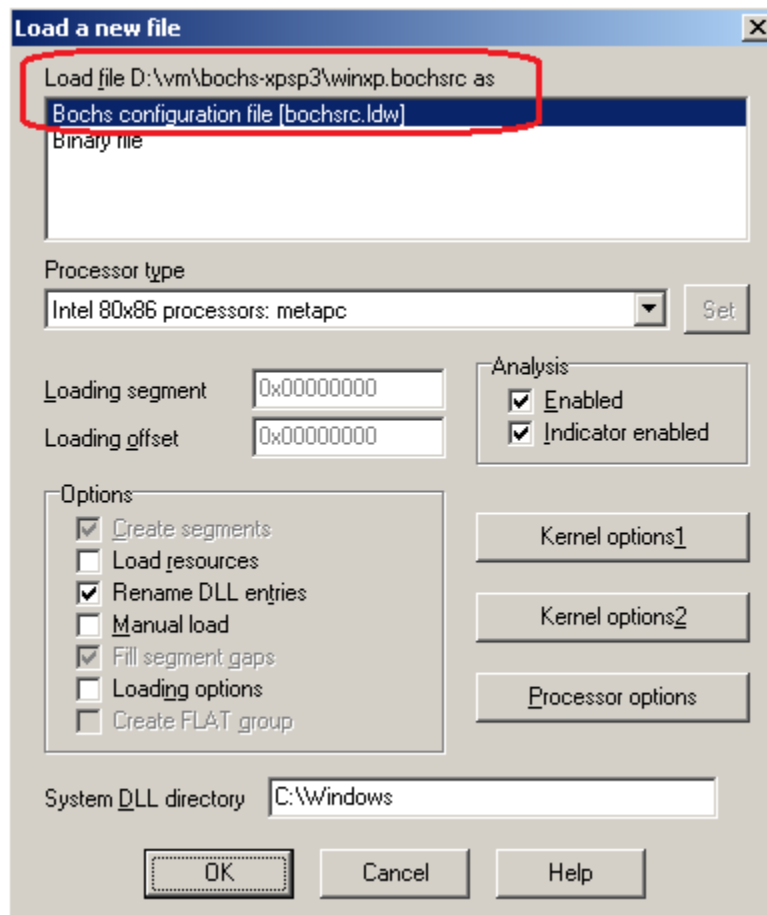
Then test if the VM works properly with Bochs:

« bochs.exe -f myvm.bochsrc »

If everything is okay then Bochs should start emulating the operating system in question.

Now let us debug the image with IDA Pro:

- Run IDA Pro and load the bochsrc file (for example: myvm.bochsrc)
- IDA Pro should recognize the file:

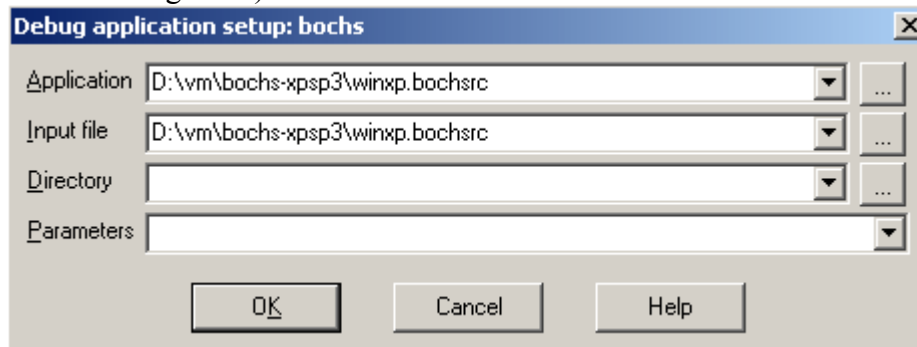


- The first disk sector will be loaded into IDA Pro

- Now simply add breakpoints and/or just press F9 to start debugging

In case IDA Pro did not recognize the bochsrc file, then manually setup Bochs like this:

- Run IDA Pro with a dummy database: “idag -t”
- Save the database in the same directory as the Bochs diskimage and bochsrc file
- Select the “Bochs local debugger” from the debuggers menu
- Go to “Debugger / Specific options” and make sure the “Disk image” operation mode is selected
- Go to “Debugger / Process options” and enter the bochsrc file name in the “Application” field. (the other fields are ignored)




That's it. Now we can simply press F9 and start the debugger.

In this screenshot, IDA Bochs in Disk image mode is used to debug MS Windows XP.

```
push    eax
mov     ebx, [ebp+arg_8]
push    esi
xor     esi, esi
add     ebx, 0Ch
cmp     ebx, 14h
push    edi
mov     [ebp+var_4], esi
jnb     short loc_F857F4BF
push    14h
pop     ebx

; CODE XREF: sub_F857F4B0
add     ebx, 3
push    offset unk_F8589DE8
and     ebx, 0FFFFFFCh
call   near ptr unk_F857F628
mov     edi, [ebp+arg_0]
mov     eax, [edi+4]
sub     eax, edi
sub     eax, 18h
cmp     ebx, eax
pop     ecx
ja      loc_F857F5AD
mov     [ebp+arg_8], edi

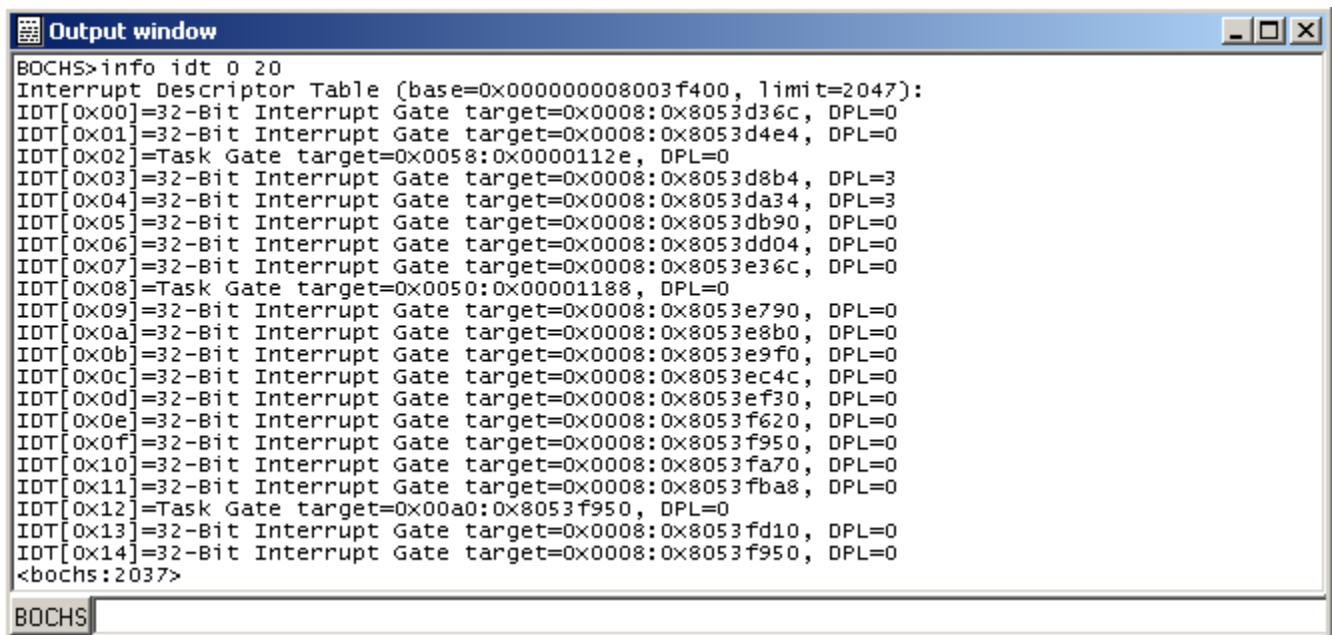
; CODE XREF: sub_F857F4B0
push    ebx
push    edi
call   sub_F857F370
mov     esi, eax
test   esi, esi
pop     ecx
pop     ecx
jnz    short loc_F857F50E
mov     eax, [edi+4]
sub     eax, [edi+10h]
```



The screenshot shows a window titled "Bochs for Windows - Display". The window content is a black background with the Microsoft Windows XP logo in the center. The logo consists of four colored panes (red, green, blue, yellow) and the text "Microsoft® Windows[™] xp". Below the logo is a small, empty rectangular box. At the bottom left of the window, it says "Copyright © Microsoft Corporation". At the bottom right, it says "Microsoft".

It is possible to send commands using the command line interpreter in IDA Pro. First press « . » (dot) and then type the desired command.

For example, we can send Bochs debugger the « **info idt** » command:



```
BOCHS>info idt 0 20
Interrupt Descriptor Table (base=0x000000008003f400, limit=2047):
IDT[0x00]=32-Bit Interrupt Gate target=0x0008:0x8053d36c, DPL=0
IDT[0x01]=32-Bit Interrupt Gate target=0x0008:0x8053d4e4, DPL=0
IDT[0x02]=Task Gate target=0x0058:0x0000112e, DPL=0
IDT[0x03]=32-Bit Interrupt Gate target=0x0008:0x8053d8b4, DPL=3
IDT[0x04]=32-Bit Interrupt Gate target=0x0008:0x8053da34, DPL=3
IDT[0x05]=32-Bit Interrupt Gate target=0x0008:0x8053db90, DPL=0
IDT[0x06]=32-Bit Interrupt Gate target=0x0008:0x8053dd04, DPL=0
IDT[0x07]=32-Bit Interrupt Gate target=0x0008:0x8053e36c, DPL=0
IDT[0x08]=Task Gate target=0x0050:0x00001188, DPL=0
IDT[0x09]=32-Bit Interrupt Gate target=0x0008:0x8053e790, DPL=0
IDT[0x0a]=32-Bit Interrupt Gate target=0x0008:0x8053e8b0, DPL=0
IDT[0x0b]=32-Bit Interrupt Gate target=0x0008:0x8053e9f0, DPL=0
IDT[0x0c]=32-Bit Interrupt Gate target=0x0008:0x8053ec4c, DPL=0
IDT[0x0d]=32-Bit Interrupt Gate target=0x0008:0x8053ef30, DPL=0
IDT[0x0e]=32-Bit Interrupt Gate target=0x0008:0x8053f620, DPL=0
IDT[0x0f]=32-Bit Interrupt Gate target=0x0008:0x8053f950, DPL=0
IDT[0x10]=32-Bit Interrupt Gate target=0x0008:0x8053fa70, DPL=0
IDT[0x11]=32-Bit Interrupt Gate target=0x0008:0x8053fba8, DPL=0
IDT[0x12]=Task Gate target=0x00a0:0x8053f950, DPL=0
IDT[0x13]=32-Bit Interrupt Gate target=0x0008:0x8053fd10, DPL=0
IDT[0x14]=32-Bit Interrupt Gate target=0x0008:0x8053f950, DPL=0
<bochs:2037>
```

Debugging code snippets:

The IDB operation mode is used to debug code snippets by simply selecting the code from the database.

In this mode, the input file format is not important. It is possible to debug code snippet from an object file, DLL file, memory dump or any other database that contains x86/32bits code.

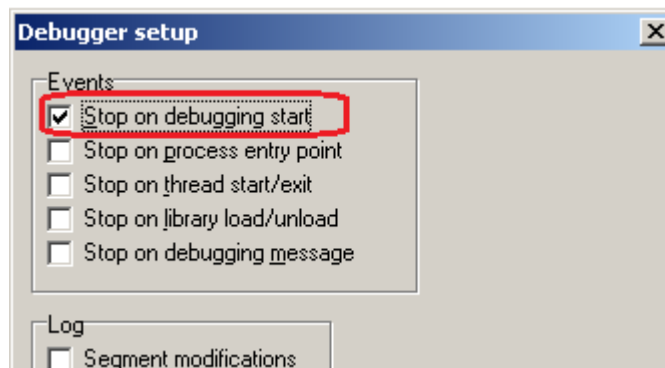
Using this mode is in fact too simple, we just have to tell it what code to debug:

- Current cursor position: Position the cursor in the database and press F9 to start debugging
- Selection: Select start and End ranges and press F9
- ENTRY and EXIT labels: Create these two labels and press F9. IDA will start executing at « ENTRY » and stop when it reaches « EXIT »

In the following screenshot, after selecting the code, we press F9 to start debugging.

```
.text:00401001      mov     ebp, esp
.text:00401003      add     esp, 0FFFFFFBCh
.text:00401006      pusha
.text:00401007      ror    edi, 15h
.text:0040100A      mov    dx, 9616h
.text:0040100E      lea   edi, ds:4F79E667h
.text:00401014      lea   ebx, ds:2FD465DEh
.text:0040101A      mov    si, 17DBh
.text:0040101E      not   ecx
.text:00401020      mov    ebx, 0D75C2031h
.text:00401025      push  offset loc_40105A
.text:0040102A      mov    edx, ebx
.text:0040102C      rol   edx, 0Fh
.text:0040102F      mov    ebx, esi
```

It is advisable to turn on the “Stop on debugging start” option when using the IDB mode. This enables the debugger to suspend automatically when the debugger starts.



Debugging win32 programs:

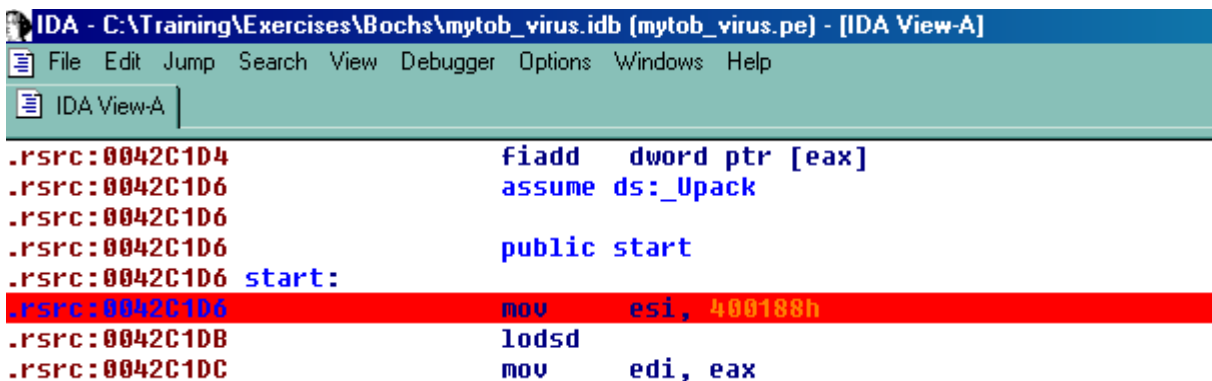
It is possible to use the Bochs debugger plugin / PE mode to debug MS win32 programs, which can be PE programs, DLLs and even system driver files.

The PE mode has many features, which are detailed in the help file and on the blog page http://hexblog.com/2008/11/bochs_plugin_goes_alpha.html

This mode is best used to debug packed malware.

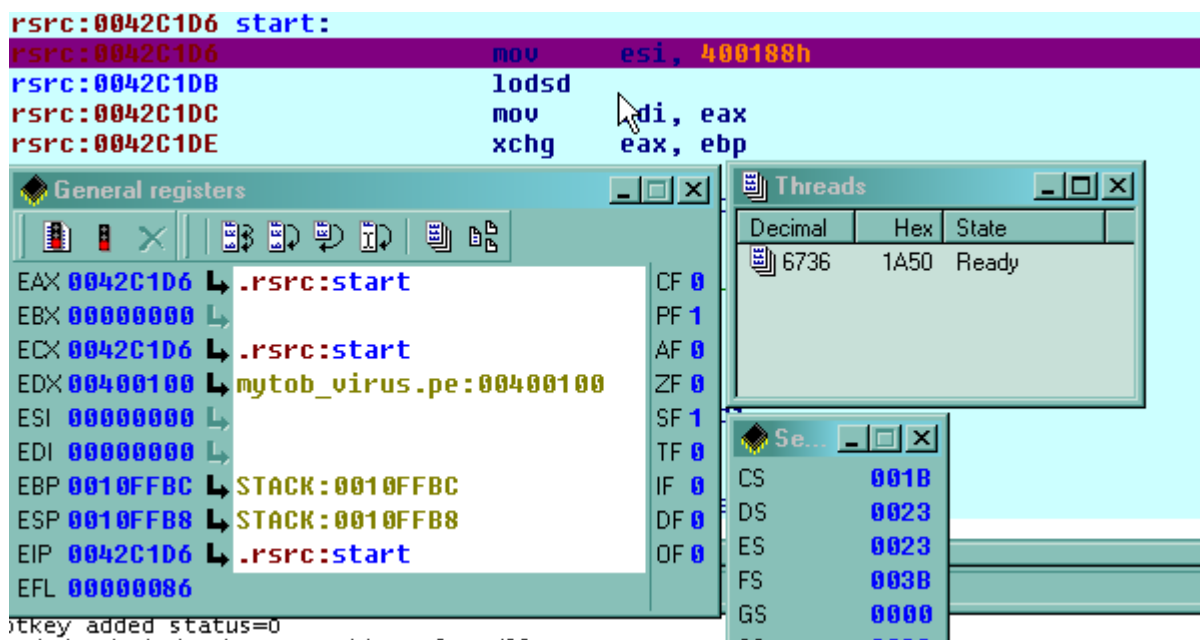
For example, we will load the “Mytob” virus into IDA Pro and then the Bochs debugger / PE mode to unpack this malware.

First we put a breakpoint at the start of the program:



```
IDA - C:\Training\Exercises\Bochs\mytob_virus.idb (mytob_virus.pe) - [IDA View-A]
File Edit Jump Search View Debugger Options Windows Help
IDA View-A
.rsrc:0042C1D4      fiadd  dword ptr [eax]
.rsrc:0042C1D6      assume ds:_Upack
.rsrc:0042C1D6
.rsrc:0042C1D6      public start
.rsrc:0042C1D6  start:
.rsrc:0042C1D6      mov    esi, 400188h
.rsrc:0042C1D8      lodsd
.rsrc:0042C1DC      mov    edi, eax
```

Now we press F9 to start the process and break at the beginning:



```
rsrc:0042C1D6  start:
rsrc:0042C1D6      mov    esi, 400188h
rsrc:0042C1D8      lodsd
rsrc:0042C1DC      mov    edi, eax
rsrc:0042C1DE      xchg   eax, ebp
```

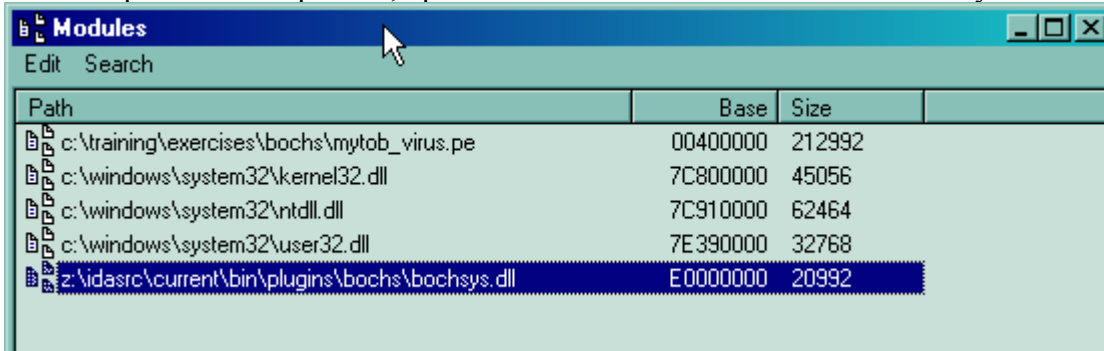
General registers	Value	Comment
EAX	0042C1D6	↳ .rsrc:start
EBX	00000000	↳
ECX	0042C1D6	↳ .rsrc:start
EDX	00400100	↳ mytob_virus.pe:00400100
ESI	00000000	↳
EDI	00000000	↳
EBP	0010FFBC	↳ STACK:0010FFBC
ESP	0010FFB8	↳ STACK:0010FFB8
EIP	0042C1D6	↳ .rsrc:start
EFL	00000086	

Threads	Decimal	Hex	State
6736	1A50	Ready	

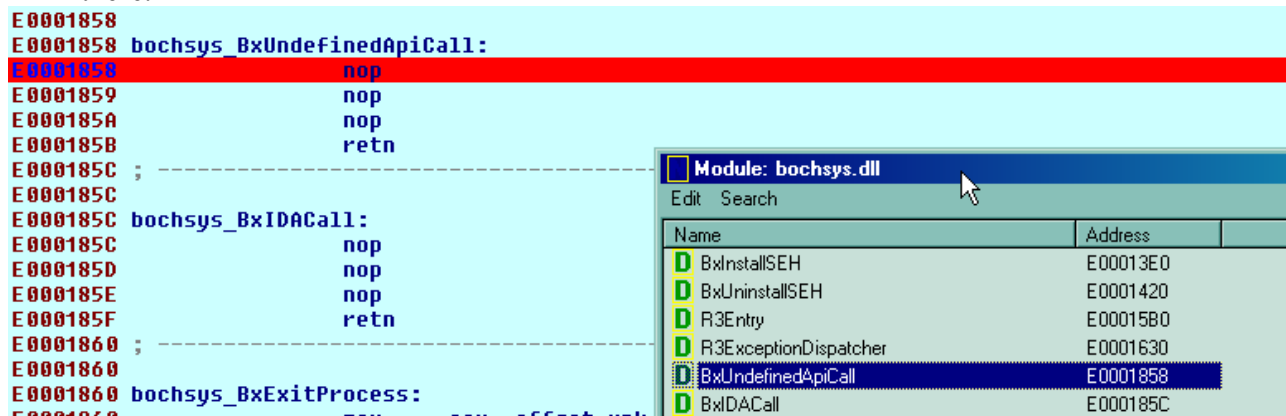
Search	Value
CS	001B
DS	0023
ES	0023
FS	003B
GS	0000

Let us use a feature in the Bochs PE mode that will allow us to break in the proximity of the original entrypoint. With this method, we do not have to trace all the way through the unpacking code.

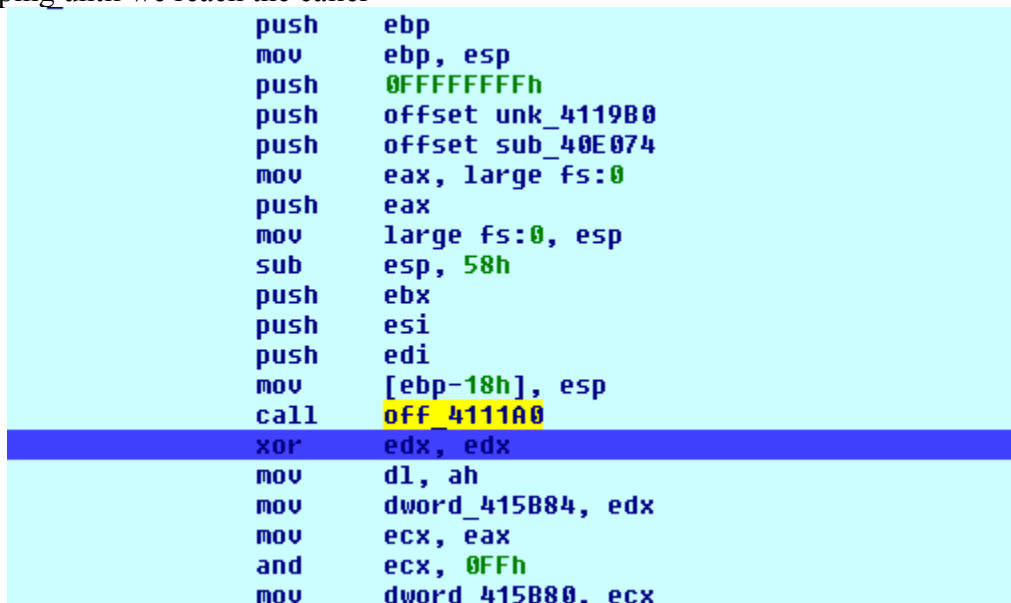
- While the process is suspended, open the modules window and select “bochsys.dll”



- Now double clicking on it will show all its exports. We are interested in “BxUndefinedApiCall”. So we simply select it and double click on it and put a breakpoint there.



- Now we press F9 again to let the malware run.
- The first time this breakpoint is reached, we could step a bit and inspect the caller. We have a high probability that the call came from the unpacked code. Anyway, we let us verify that by stepping until we reach the caller



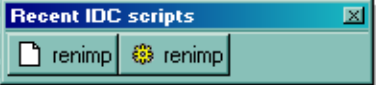
As we see, that tracing up to the caller led us to this code which looks familiar and is actually the startup code for many VC++ programs.

Let us go to that offset and see what we got:

```
00411184 off_411184 dd offset kernel32_CompareStringW
00411184 ; DATA XREF: sub_410323+3E1r
00411184 ; sub_410323+2611r
00411188 off_411188 dd offset kernel32_DeleteFileA ; DATA XREF: sub_402840+FD1r
00411188 ; sub_402840+1321r ...
0041118C off_41118C dd offset kernel32_SetThreadPriority
0041118C ; DATA XREF: .Upack:00403F4F1r
00411190 off_411190 dd offset unk_7C804247 ; DATA XREF: sub_4098E7+F01r
00411190 ; sub_40C1DD+281r
00411194 off_411194 dd offset kernel32_TerminateProcess
00411194 ; DATA XREF: sub_409FA2+171r
00411198 off_411198 dd offset kernel32_GetStartupInfoA
00411198 ; DATA XREF: .Upack:0040A18A1r
00411198 ; sub_40DDD0+591r
0041119C off_41119C dd offset kernel32_GetCommandLineA
0041119C ; DATA XREF: .Upack:0040A15F1r
004111A0 off_4111A0 dd offset kernel32_GetVersion ; DATA XREF: .Upack:0040A1111r
```

It looks like we located the import functions RVAs. Let us use the “renimp.idc” script to give relevant names to these offsets:

```
00411188 ; BOOL __stdcall DeleteFileA(LPCSTR lpFileName)
00411188 DeleteFileA dd offset kernel32_DeleteFileA ; DATA XREF: sub_402840+FD1r
00411188 ; sub_402840+1321r
0041118C ; BOOL __stdcall SetThreadPriority(HANDLE hThread,
0041118C SetThreadPriority dd offset kernel32_SetThreadPriority
0041118C ; DATA XREF: .Upack:00403F4F1r
00411190 off_411190 dd offset unk_7C804247 ; DATA XREF: sub_4098E7+F01r
00411190 ; sub_40C1DD+281r
00411194 ; BOOL __stdcall TerminateProcess(HANDLE hProcess, UINT uExitCode)
00411194 TerminateProcess dd offset kernel32_TerminateProcess
00411194 ; DATA XREF: sub_409FA2+171r
00411198 ; void __stdcall GetStartupInfoA(LPSTARTUPINFOA lpStartupInfo)
00411198 GetStartupInfoA dd offset kernel32_GetStartupInfoA
00411198 ; DATA XREF: .Upack:0040A18A1r
00411198 ; sub_40DDD0+591r
0041119C ; LPSTR __stdcall GetCommandLineA()
0041119C GetCommandLineA dd offset kernel32_GetCommandLineA
0041119C ; DATA XREF: .Upack:0040A15F1r
004111A0 ; DWORD __stdcall GetVersion()
004111A0 GetVersion dd offset kernel32_GetVersion ; DATA XREF: .Upack:0040A1111r
```



And finally let us go back to the OEP and see how it looks like now:

```
040A0EB start proc near ; CODE
040A0EB
040A0EB var_68 = dword ptr -68h
040A0EB var_64 = dword ptr -64h
040A0EB var_60 = dword ptr -60h
040A0EB var_5C = byte ptr -5Ch
040A0EB var_30 = dword ptr -30h
040A0EB var_2C = word ptr -2Ch
040A0EB var_18 = dword ptr -18h
040A0EB var_14 = dword ptr -14h
040A0EB var_4 = dword ptr -4
040A0EB
040A0EB push ebp
040A0EC mov ebp, esp
040A0EE push 0FFFFFFFh
040A0F0 push offset unk_4119B0
040A0F5 push offset sub_40E074
040A0FA mov eax, large fs:0
040A100 push eax
040A101 mov large fs:0, esp
040A108 sub esp, 58h
040A10B push ebx
040A10C push esi
040A10D push edi
040A10E mov [ebp+var_18], esp
040A111 call GetVersion
040A117 xor edx, edx
040A119 mov dl, ah
040A11B mov dword_415B84, edx
```

That is it!

We got it unpacked, now we can delete unused segments, take a memory snapshot and get ready for static analysis or even decompilation with the Hex-Rays decompiler.