Using IDA Pro's Debugger. © DataRescue 2005

This small tutorial introduces the main functionalities of the IDA Debugger plugin. IDA supports debugging of x86 Windows PE files, AMD64 Windows PE files, and x86 Linux ELF files, either locally or remotely. Let's see how the debugger can be used to locally debug a simple buggy C console program compiled under Windows.

The buggy program.

This program simply computes averages of a set of values (1, 2, 3, 4 and 5). Those values are stored in two arrays: one containing 8 bit values, the other containing 32-bit values.

```
#include <stdio.h>

char char_average(char array[], int count)
{
  int i;
  char average;

  average = 0;
  for (i = 0; i < count; i++)
    average += array[i];
  average /= count;
  return average;
}


int int_average(int array[], int count)
{
  int i, average;

  average = 0;
  for (i = 0; i < count; i++)
    average += array[i];
  average /= count;
  return average;
}


void main(void) {
  char chars[]    = { 1, 2, 3, 4, 5 };
  int  integers[] = { 1, 2, 3, 4, 5 };

  printf("chars[]    - average = %d\n",
    char_average(chars, sizeof(chars)));
  printf("integers[] - average = %d\n",
    int_average(integers, sizeof(integers)));
}
```

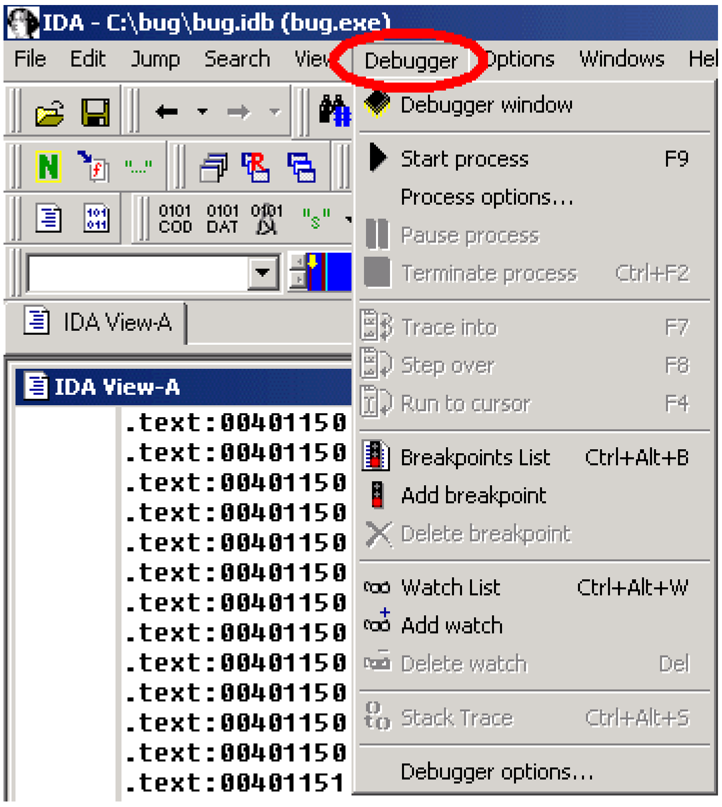
Running this program gives us the following results:

```
chars[]    - average = 3
integers[] - average = 1054228
```

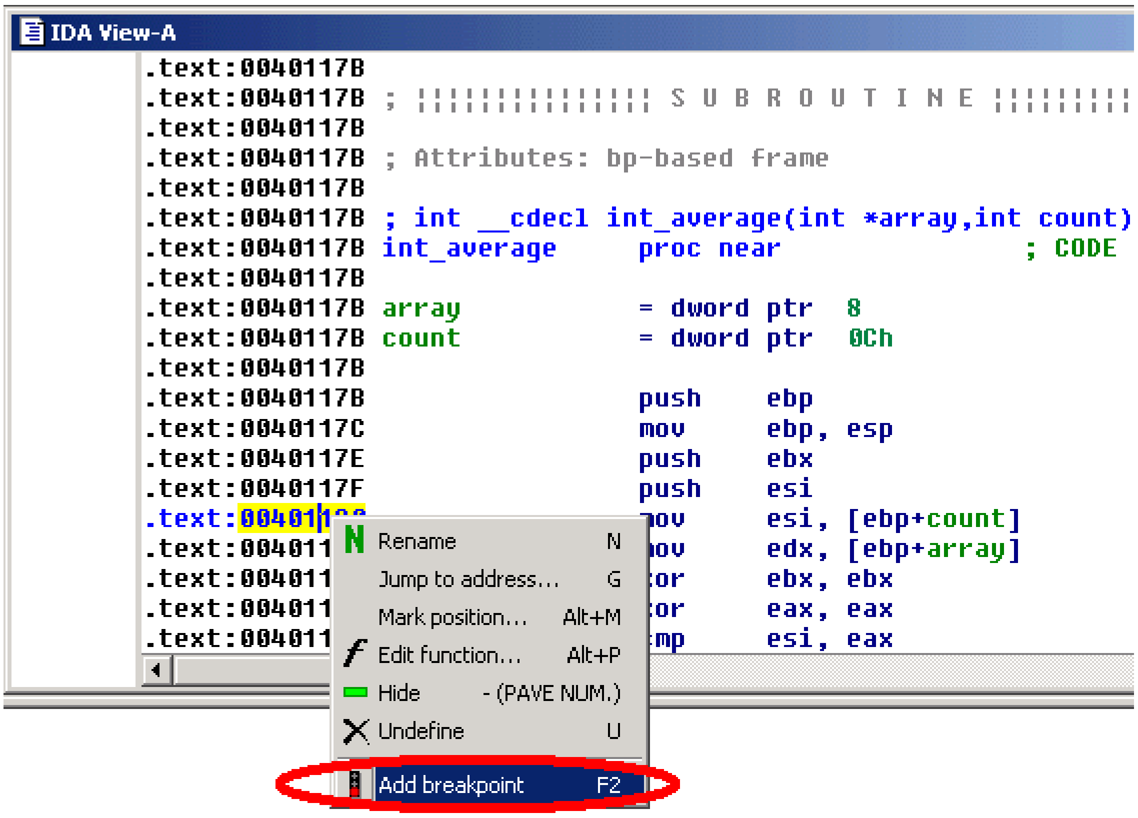
Obviously, the computed average on the integer array is wrong. Let's use IDA's debugger to understand the origin of this error !

Loading the file.

The debugger is perfectly integrated with IDA: to debug, we must first load the executable in IDA, to create a database. The user can disassemble the file interactively, and all the information which he will have added to the disassembly will be available during debugging. If the disassembled file is recognized as valid (x86/ARM64 PE or x86 ELF) by the debugger, a Debugger menu automatically appears in IDA's main window.



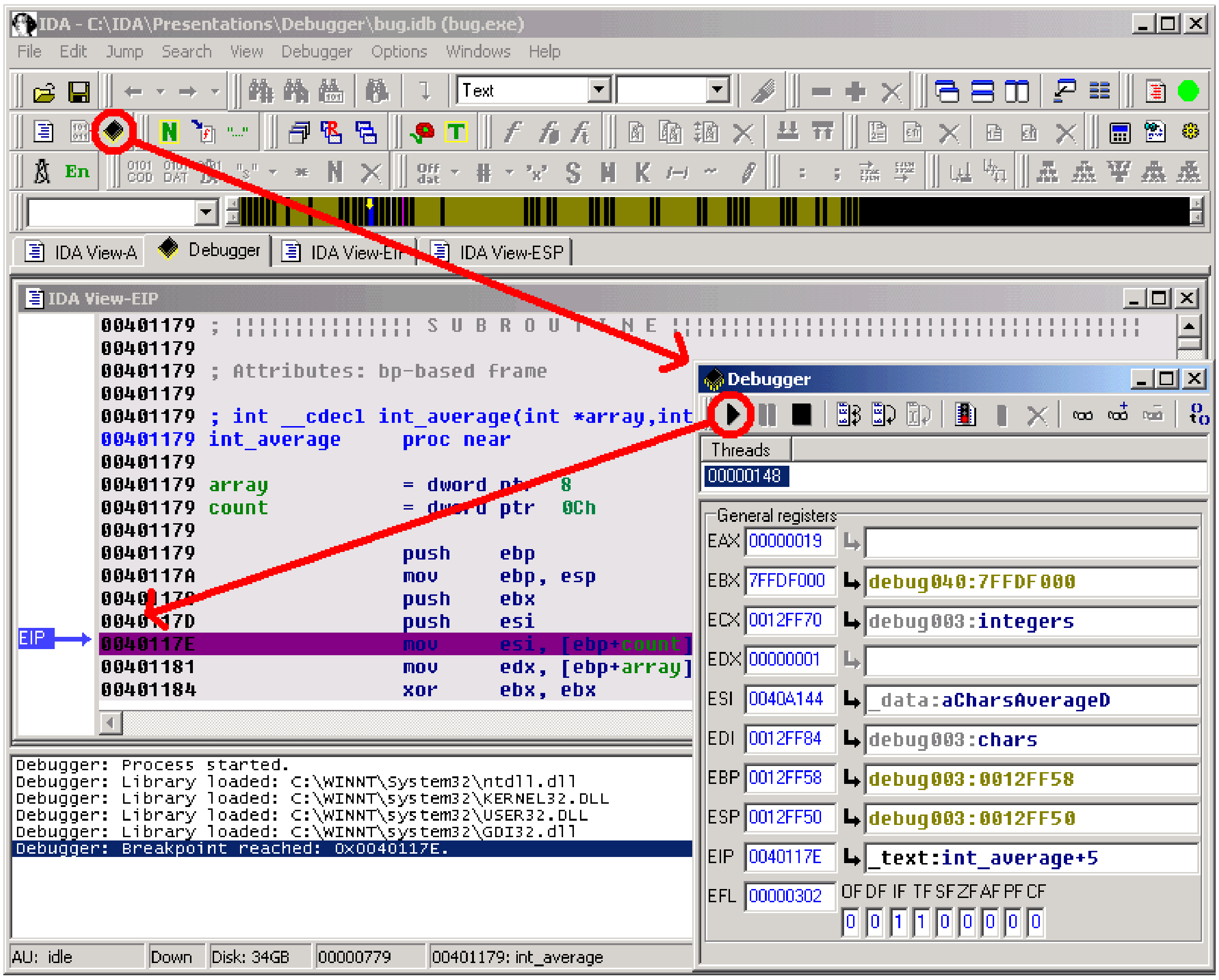
Once we located our *int_average()* function in the disassembly, let's add a breakpoint just after its prolog, by selecting the *Add breakpoint* command in the popup menu, or by pressing the F2 key.



```
.text:0040117B
.text:0040117B ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦
.text:0040117B
.text:0040117B ; Attributes: bp-based frame
.text:0040117B
.text:0040117B ; int __cdecl int_average(int *array,int count)
.text:0040117B int_average     proc near                ; CODE
.text:0040117B
.text:0040117B array           = dword ptr  8
.text:0040117B count           = dword ptr  0Ch
.text:0040117B
.text:0040117B                 push    ebp
.text:0040117C                 mov     ebp, esp
.text:0040117E                 push    ebx
.text:0040117F                 push    esi
.text:00401                    ov      esi, [ebp+count]
.text:004011                   ov      edx, [ebp+array]
.text:004011                   or      ebx, ebx
.text:004011                   or      eax, eax
.text:004011                   mp      esi, eax
```

Program execution.

Now, we can start the execution. We simply open the debugger window by using the appropriate icon, and run the program until it reaches our breakpoint, by pressing the F9 key or clicking the *Start* button in the debugger toolbar.



```
00401179 ; ||||||||||||||| S U B R O U T I N E ||||||||||||||||||||||||||||||||||||||||
00401179
00401179 ; Attributes: bp-based frame
00401179
00401179 ; int __cdecl int_average(int *array,int
00401179 int_average     proc near
00401179
00401179 array           = dword ptr  8
00401179 count           = dword ptr  0Ch
00401179
00401179                 push    ebp
0040117A                 mov     ebp, esp
0040117C                 push    ebx
0040117D                 push    esi
0040117E                 mov     esi, [ebp+count]
00401181                 mov     edx, [ebp+array]
00401184                 xor     ebx, ebx
```

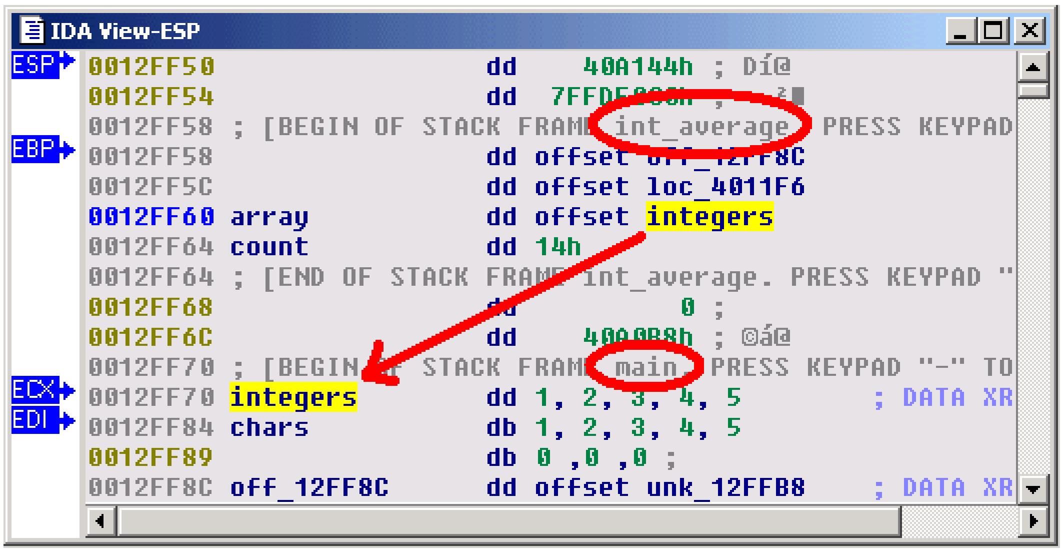
```
Debugger: Process started.
Debugger: Library loaded: C:\WINNT\System32\ntdll.dll
Debugger: Library loaded: C:\WINNT\system32\KERNEL32.DLL
Debugger: Library loaded: C:\WINNT\system32\USER32.DLL
Debugger: Library loaded: C:\WINNT\system32\GDI32.dll
Debugger: Breakpoint reached: 0x0040117E.
```

```
Threads
00000148

General registers
EAX 00000019
EBX 7FFDF000  debug040:7FFDF000
ECX 0012FF70  debug003:integers
EDX 00000001
ESI 0040A144  _data:aCharsAverageD
EDI 0012FF84  debug003:chars
EBP 0012FF58  debug003:0012FF58
ESP 0012FF50  debug003:0012FF50
EIP 0040117E  _text:int_average+5
EFL 00000302  OF DF IF TF SF ZF AF PF CF
              0  0  1  1  0  0  0  0  0
```

The stack.

The IDA View-ESP window now shows us the stack frame of the interesting functions. We easily locate the array argument of our *int_average()* function, pointing to the integer array in the calling function (*main()* function).

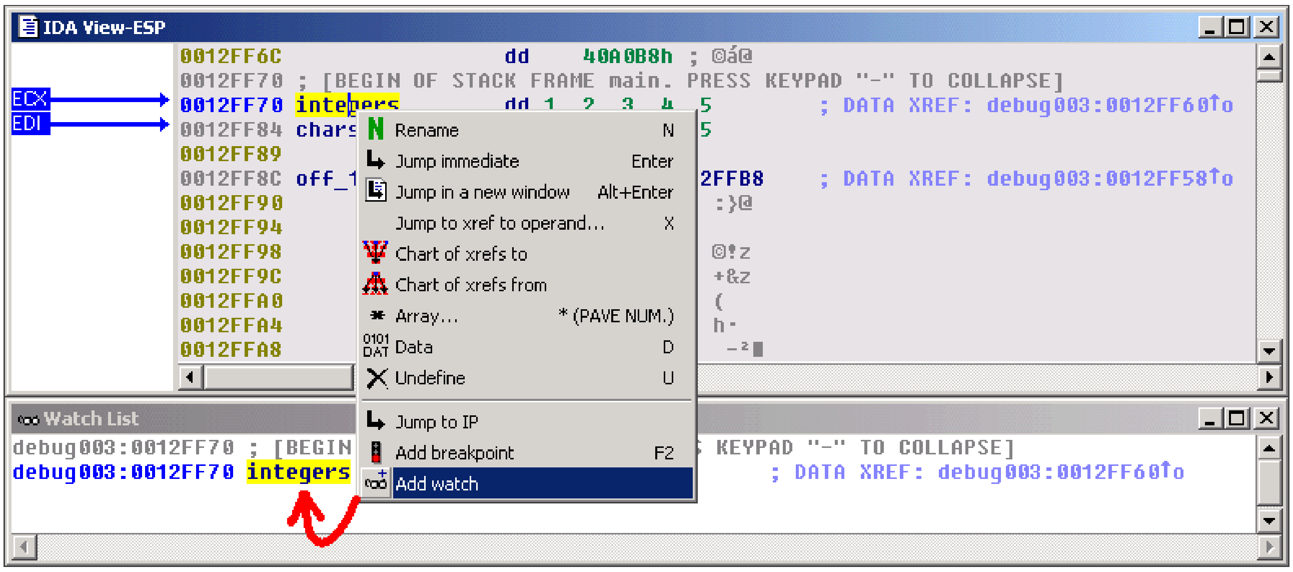


```
IDA View-ESP
ESP 0012FF50                  dd    40A144h ; Dí@
    0012FF54                  dd   7FFDF000h ;
    0012FF58 ; [BEGIN OF STACK FRAME int_average. PRESS KEYPAD
EBP 0012FF58                  dd offset off_12FF8C
    0012FF5C                  dd offset loc_4011F6
    0012FF60 array            dd offset integers
    0012FF64 count            dd 14h
    0012FF64 ; [END OF STACK FRAME int_average. PRESS KEYPAD "
    0012FF68                  dd          0 ;
    0012FF6C                  dd    40A0B8h ; ©á@
    0012FF70 ; [BEGIN OF STACK FRAME main. PRESS KEYPAD "-" TO
ECX 0012FF70 integers         dd 1, 2, 3, 4, 5          ; DATA XR
EDI 0012FF84 chars            db 1, 2, 3, 4, 5
    0012FF89                  db 0 ,0 ,0 ;
    0012FF8C off_12FF8C       dd offset unk_12FFB8      ; DATA XR
```

Now, we can start the execution. We simply open the debugger window by using the appropriate icon, and run the program until it reaches our breakpoint, by pressing the F9 key or clicking the Start button in the debugger toolbar.

Watches.

Why not add a watch on this array, in order to observe the evolution of its values during the process execution ?

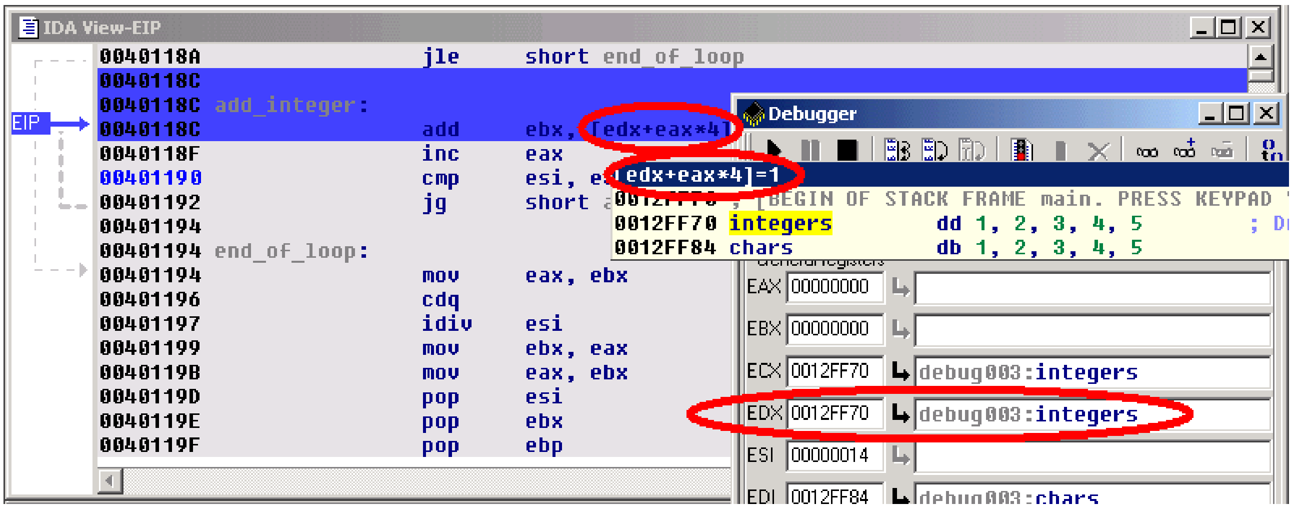


```
IDA View-ESP
    0012FF6C               dd    40A0B8h ; ©á@
    0012FF70 ; [BEGIN OF STACK FRAME main. PRESS KEYPAD "-" TO COLLAPSE]
ECX 0012FF70 integers      dd 1, 2, 3, 4, 5      ; DATA XREF: debug003:0012FF60↑o
EDI 0012FF84 chars                        5
    0012FF89
    0012FF8C off_1              2FFB8            ; DATA XREF: debug003:0012FF58↑o
    0012FF90                     :}@
    0012FF94
    0012FF98                     ©!z
    0012FF9C                     +&z
    0012FFA0                     (
    0012FFA4                     h·
    0012FFA8                     -²

Watch List
debug003:0012FF70 ; [BEGIN       KEYPAD "-" TO COLLAPSE]
debug003:0012FF70 integers       ; DATA XREF: debug003:0012FF60↑o
```

Address evaluation.

By analyzing the disassembled code, we can now locate the loop which computes the sum of the values, and stores the result in the EBX register. The [edx+eax*4] operand clearly shows us that the EDX register points to the start of the array, and that the EAX register is used as an index in this array. Thus, this operand will successively point to each integer from the *integers* array.



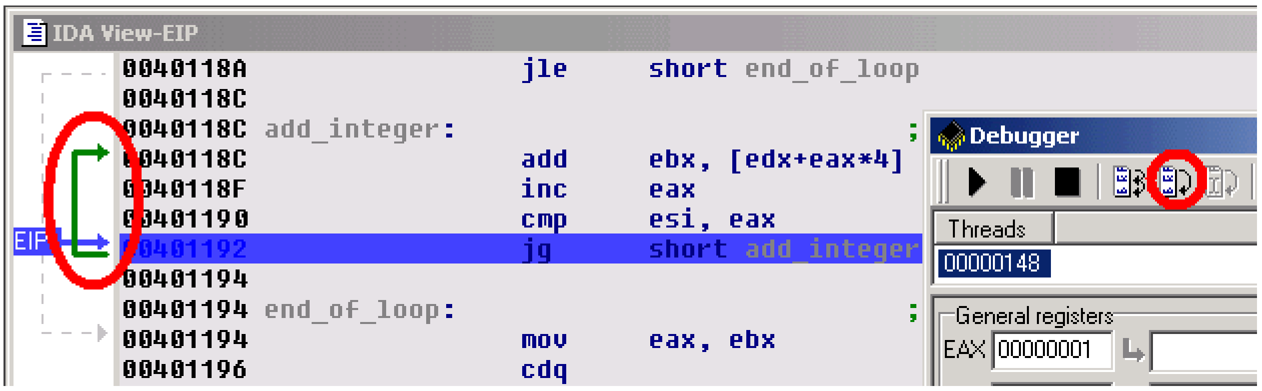
```
0040118A                 jle     short end_of_loop
0040118C
0040118C add_integer:
0040118C                 add     ebx, [edx+eax*4]
0040118F                 inc     eax
00401190                 cmp     esi, e
00401192                 jg      short a
00401194
00401194 end_of_loop:
00401194                 mov     eax, ebx
00401196                 cdq
00401197                 idiv    esi
00401199                 mov     ebx, eax
0040119B                 mov     eax, ebx
0040119D                 pop     esi
0040119E                 pop     ebx
0040119F                 pop     ebp
```

```
[edx+eax*4]=1
0012FF70 ; [BEGIN OF STACK FRAME main. PRESS KEYPAD '
0012FF70 integers        dd 1, 2, 3, 4, 5        ; D
0012FF84 chars           db 1, 2, 3, 4, 5
```

```
EAX 00000000
EBX 00000000
ECX 0012FF70  debug003:integers
EDX 0012FF70  debug003:integers
ESI 00000014
EDI 0012FF84  debug003:chars
```

Step by step and jump targets.

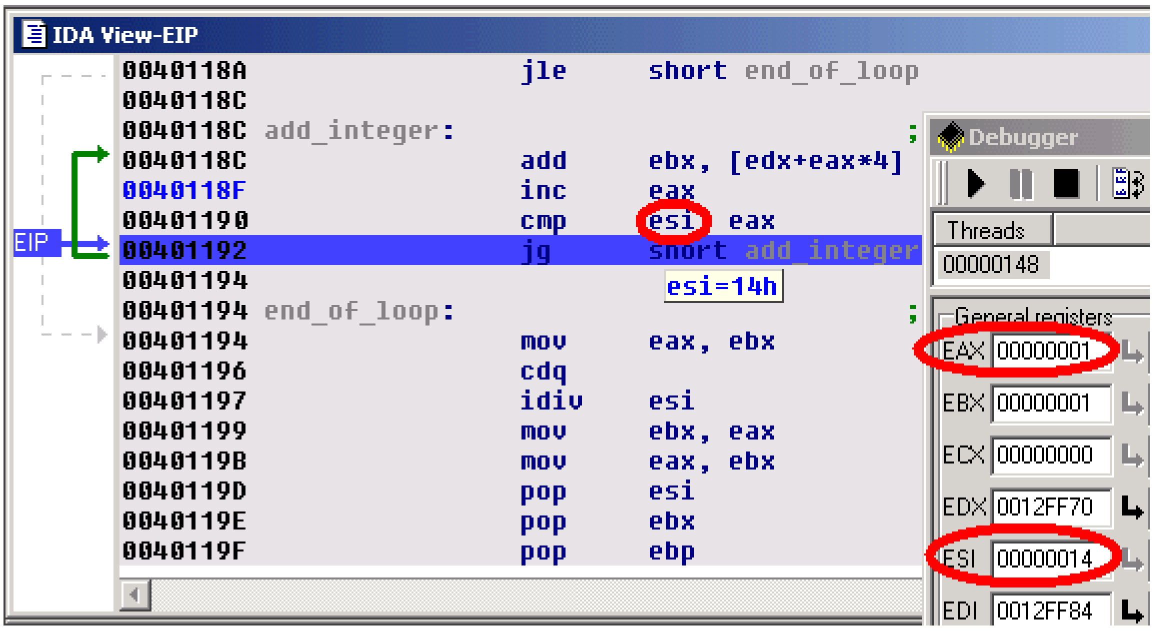
Let's advance step by step in the loop, by clicking on the adequate button in the debugger toolbar or by pressing the F8 key. If necessary, IDA draws a green arrow to show us the target of a jump instruction.



```
0040118A                 jle     short end_of_loop
0040118C
0040118C add_integer:                            ;
0040118C                 add     ebx, [edx+eax*4]
0040118F                 inc     eax
00401190                 cmp     esi, eax
00401192                 jg      short add_integer
00401194
00401194 end_of_loop:                            ;
00401194                 mov     eax, ebx
00401196                 cdq
```

```
00000148
EAX 00000001
```

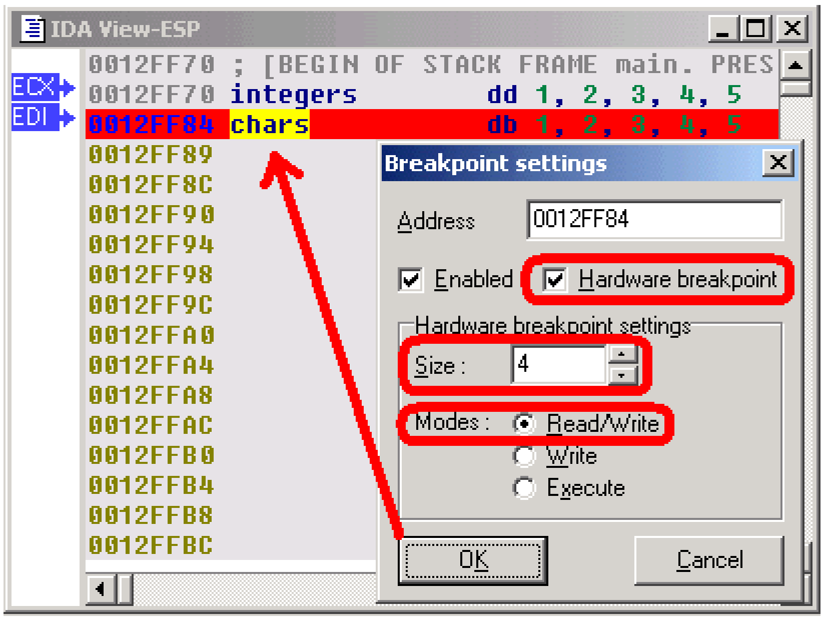
Now, let's have a look at ESI's value. The EAX register (our index in the array) is compared to this register at each iteration: so, we can conclude that the ESI register is used as a counter in the loop. But, we also observe that ESI contains a rather strange number of elements: 14h (= 20). Remember that our original array contains only 5 elements ! It seems we just found the source of our problem...



```
0040118A                 jle     short end_of_loop
0040118C
0040118C add_integer:                                 ;
0040118C                 add     ebx, [edx+eax*4]
0040118F                 inc     eax
00401190                 cmp     esi, eax
00401192                 jg      short add_integer
00401194                          esi=14h
00401194 end_of_loop:                                 ;
00401194                 mov     eax, ebx
00401196                 cdq
00401197                 idiv    esi
00401199                 mov     ebx, eax
0040119B                 mov     eax, ebx
0040119D                 pop     esi
0040119E                 pop     ebx
0040119F                 pop     ebp
```

EAX	00000001
EBX	00000001
ECX	00000000
EDX	0012FF70
ESI	00000014
EDI	0012FF84

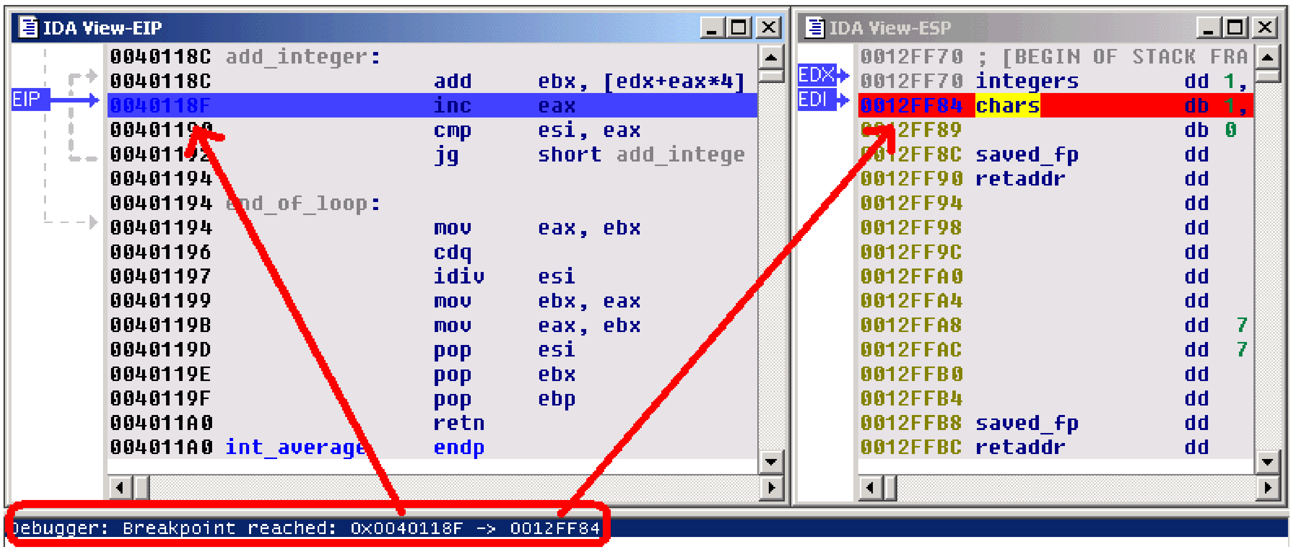
To be sure, let's add a hardware breakpoint, just behind the last value of our *integers* array (in fact, on the first value of the *chars* array). If we reach this breakpoint during the loop, it will indeed prove that we read integers outside our array. So, we setup a hardware breakpoint with a size of 4 bytes (classical size for an integer) in *Read* mode.



```
0012FF70 ; [BEGIN OF STACK FRAME main. PRES
0012FF70 integers        dd 1, 2, 3, 4, 5
0012FF84 chars           db 1, 2, 3, 4, 5
0012FF89
0012FF8C
0012FF90
0012FF94
0012FF98
0012FF9C
0012FFA0
0012FFA4
0012FFA8
0012FFAC
0012FFB0
0012FFB4
0012FFB8
0012FFBC
```

Address: 0012FF84
[✓] Enabled [✓] Hardware breakpoint
Hardware breakpoint settings
Size : 4
Modes : [✓ Read/Write] Write Execute

As foreseen, if we continue the execution, the hardware breakpoint indeed detects a read access to the first byte of the *chars* array. Remark that EIP points to the instruction following the one which caused the hardware breakpoint ! It is in fact rather logical: to cause the hardware breakpoint, the preceding instruction has been fully executed, so EIP now points to the next one.

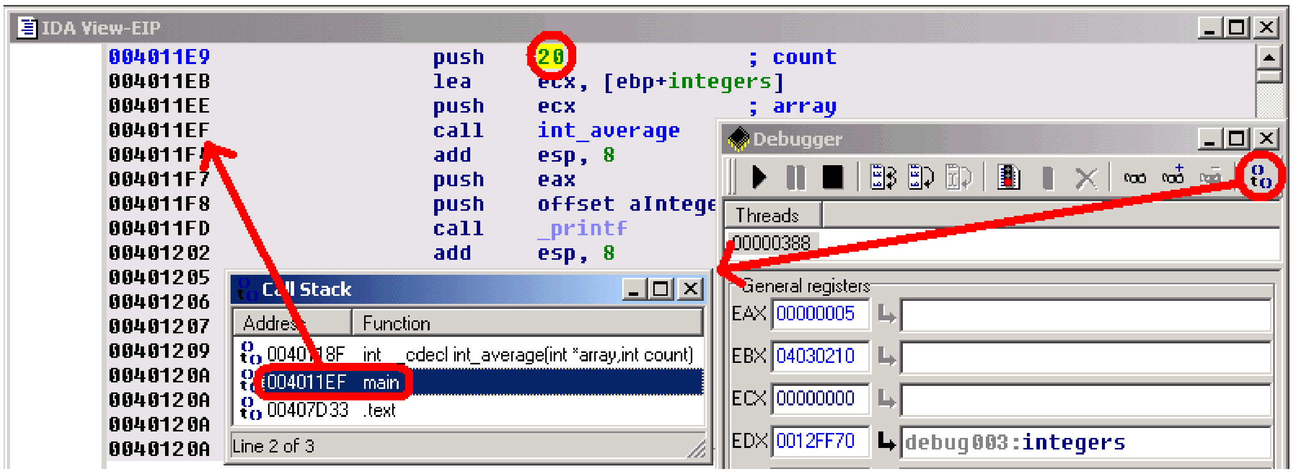


```
0040118C add_integer:
0040118C                 add     ebx, [edx+eax*4]
0040118F                 inc     eax
00401190                 cmp     esi, eax
00401192                 jg      short add_intege
00401194
00401194 end_of_loop:
00401194                 mov     eax, ebx
00401196                 cdq
00401197                 idiv    esi
00401199                 mov     ebx, eax
0040119B                 mov     eax, ebx
0040119D                 pop     esi
0040119E                 pop     ebx
0040119F                 pop     ebp
004011A0                 retn
004011A0 int_average     endp
```

```
0012FF70 ; [BEGIN OF STACK FRA
0012FF70 integers        dd 1,
0012FF84 chars           db 1,
0012FF89                 db 0
0012FF8C saved_fp        dd
0012FF90 retaddr         dd
0012FF94                 dd
0012FF98                 dd
0012FF9C                 dd
0012FFA0                 dd
0012FFA4                 dd
0012FFA8                 dd  7
0012FFAC                 dd  7
0012FFB0                 dd
0012FFB4                 dd
0012FFB8 saved_fp        dd
0012FFBC retaddr         dd
```

```
Debugger: Breakpoint reached: 0x0040118F -> 0012FF84
```

By looking at the disassembly, we see that the value stored in ESI comes from the count argument of our *int_average()* function. Let's try to understand why the caller gives us such a strange argument: If we open the Stack Trace window, we see a stack of all caller functions. Simply double click on the *main()* function, to jump to the caller code. With the help of IDA's PIT (Parameter Identification and Tracking) technology, we easily locate the *push 20* instruction, passing an erroneous count value to our *int_average()* function.



```
004011E9    push    20                      ; count
004011EB    lea     ecx, [ebp+integers]
004011EE    push    ecx                     ; array
004011EF    call    int_average
004011F     add     esp, 8
004011F7    push    eax
004011F8    push    offset aIntege
004011FD    call    _printf
00401202    add     esp, 8
00401205
00401206
00401207
00401209
0040120A
0040120A
0040120A
```

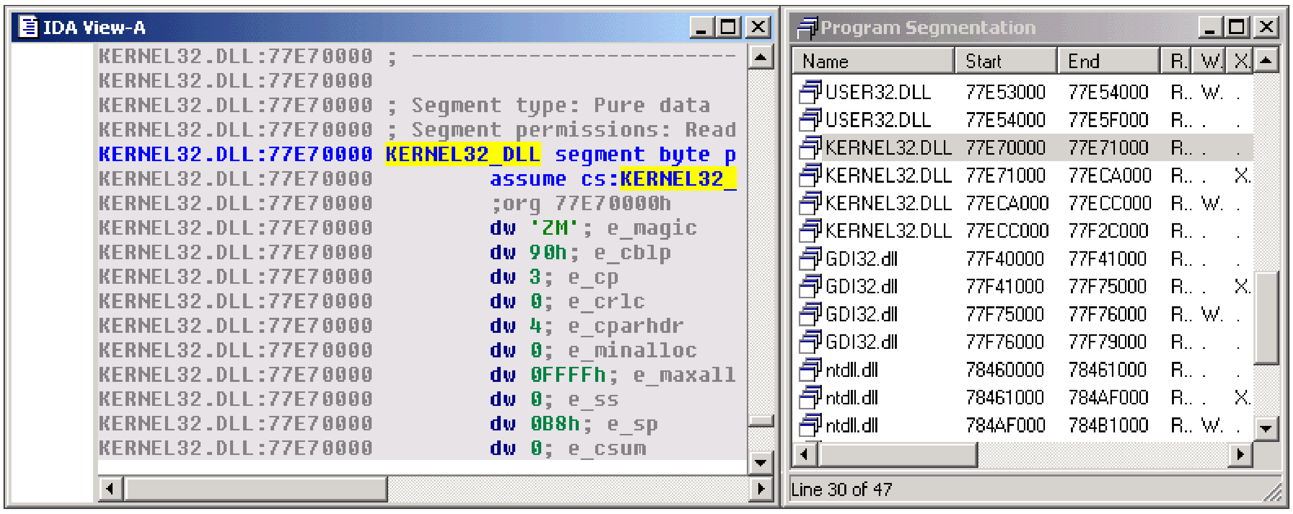
```
Address    Function
0040118F   int __cdecl int_average(int *array,int count)
004011EF   main
00407D33   .text
Line 2 of 3
```

```
00000388

EAX 00000005
EBX 04030210
ECX 00000000
EDX 0012FF70   debug003:integers
```

Now, by looking closer at the C source code, we understand our error: we used the *sizeof()* operator, which returns the **number of bytes** in the array, rather than returning the **number of items** in this array ! As, for the *chars* array, the number of bytes was equal to the number of items, we didn't notice the error...

IDA's debugger gives you access to all the segments of a debugged process's memory space, allowing you to use all of IDA's powerful features: you can apply structures to bytes in memory, draw graphs, create breakpoints in DLLs, ...

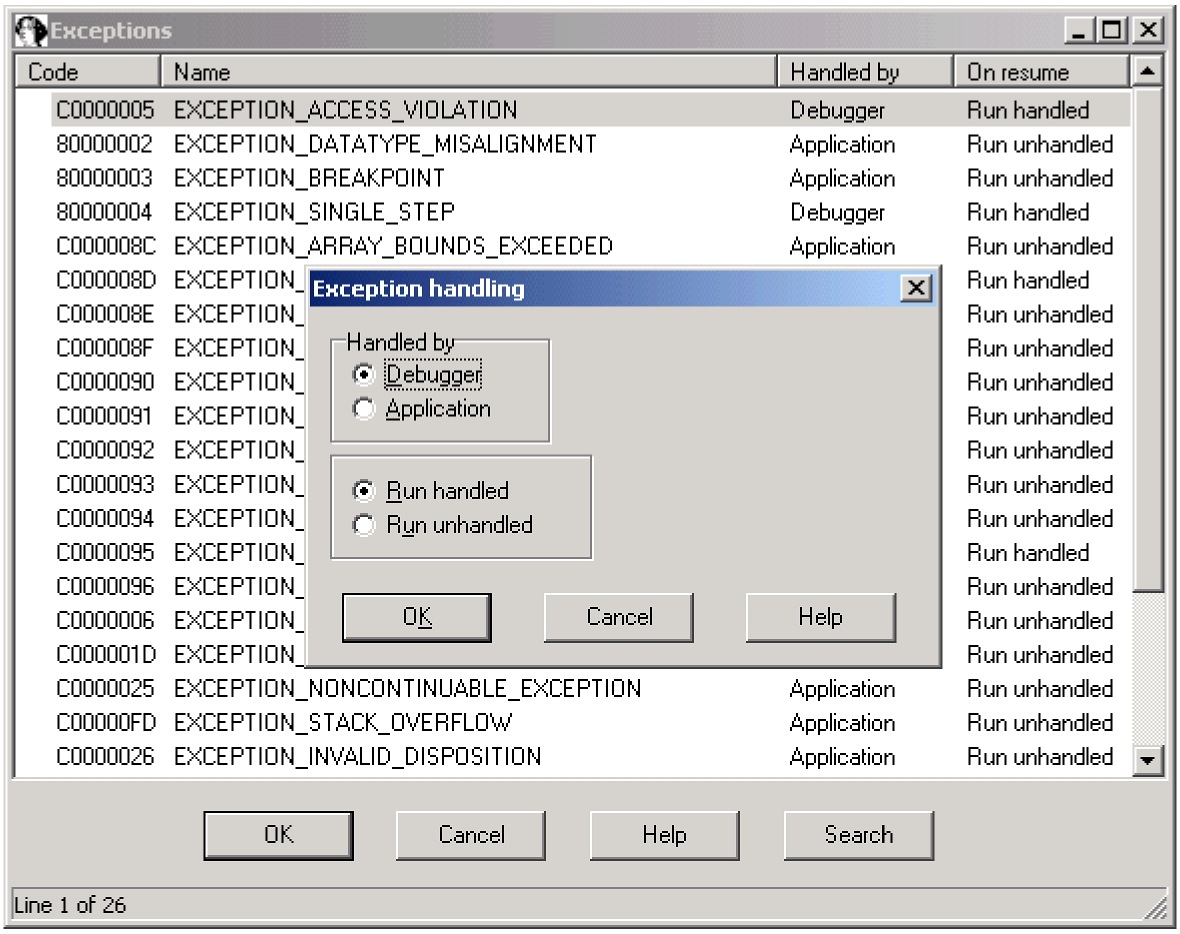


```
KERNEL32.DLL:77E70000 ; -------------------------
KERNEL32.DLL:77E70000
KERNEL32.DLL:77E70000 ; Segment type: Pure data
KERNEL32.DLL:77E70000 ; Segment permissions: Read
KERNEL32.DLL:77E70000 KERNEL32_DLL segment byte p
KERNEL32.DLL:77E70000             assume cs:KERNEL32_
KERNEL32.DLL:77E70000             ;org 77E70000h
KERNEL32.DLL:77E70000             dw 'ZM'; e_magic
KERNEL32.DLL:77E70000             dw 90h; e_cblp
KERNEL32.DLL:77E70000             dw 3; e_cp
KERNEL32.DLL:77E70000             dw 0; e_crlc
KERNEL32.DLL:77E70000             dw 4; e_cparhdr
KERNEL32.DLL:77E70000             dw 0; e_minalloc
KERNEL32.DLL:77E70000             dw 0FFFFh; e_maxall
KERNEL32.DLL:77E70000             dw 0; e_ss
KERNEL32.DLL:77E70000             dw 0B8h; e_sp
KERNEL32.DLL:77E70000             dw 0; e_csum
```

Name	Start	End	R.	W.	X.
USER32.DLL	77E53000	77E54000	R..	W.	.
USER32.DLL	77E54000	77E5F000	R..	.	.
KERNEL32.DLL	77E70000	77E71000	R..	.	.
KERNEL32.DLL	77E71000	77ECA000	R..	.	X.
KERNEL32.DLL	77ECA000	77ECC000	R..	W.	.
KERNEL32.DLL	77ECC000	77F2C000	R..	.	.
GDI32.dll	77F40000	77F41000	R..	.	.
GDI32.dll	77F41000	77F75000	R..	.	X.
GDI32.dll	77F75000	77F76000	R..	W.	.
GDI32.dll	77F76000	77F79000	R..	.	.
ntdll.dll	78460000	78461000	R..	.	.
ntdll.dll	78461000	784AF000	R..	.	X.
ntdll.dll	784AF000	784B1000	R..	W.	.

Line 30 of 47

The way the debugger reacts to exceptions is fully configurable by the user.



Code	Name	Handled by	On resume
C0000005	EXCEPTION_ACCESS_VIOLATION	Debugger	Run handled
80000002	EXCEPTION_DATATYPE_MISALIGNMENT	Application	Run unhandled
80000003	EXCEPTION_BREAKPOINT	Application	Run unhandled
80000004	EXCEPTION_SINGLE_STEP	Debugger	Run handled
C000008C	EXCEPTION_ARRAY_BOUNDS_EXCEEDED	Application	Run unhandled
C000008D	EXCEPTION_		Run handled
C000008E	EXCEPTION_		Run unhandled
C000008F	EXCEPTION_		Run unhandled
C0000090	EXCEPTION_		Run unhandled
C0000091	EXCEPTION_		Run unhandled
C0000092	EXCEPTION_		Run unhandled
C0000093	EXCEPTION_		Run unhandled
C0000094	EXCEPTION_		Run unhandled
C0000095	EXCEPTION_		Run handled
C0000096	EXCEPTION_		Run unhandled
C0000006	EXCEPTION_		Run unhandled
C000001D	EXCEPTION_		Run unhandled
C0000025	EXCEPTION_NONCONTINUABLE_EXCEPTION	Application	Run unhandled
C00000FD	EXCEPTION_STACK_OVERFLOW	Application	Run unhandled
C0000026	EXCEPTION_INVALID_DISPOSITION	Application	Run unhandled

Exception handling

Handled by: [✓ Debugger] Application

[✓ Run handled] Run unhandled

OK Cancel Help

Line 1 of 26

This debugger is thus the essential complement to IDA itself, allowing you to interactively disassemble and debug everything, everywhere.

This tutorial is © DataRescue SA/NV 2005

Revision 1.1

DataRescue SA/NV

40 Bld Piercot

4000 Liège, Belgium

T: +32-4-3446510   F: +32-4-3446514