Using IDA Pro's tracing features. © DataRescue 2005

The first debugger tutorial explained how to use traditional debugger commands in order to debug a simple buggy C console program. This small tutorial will present another approach to debug this application, by introducing the tracing functionalities of IDA's debugger.

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:

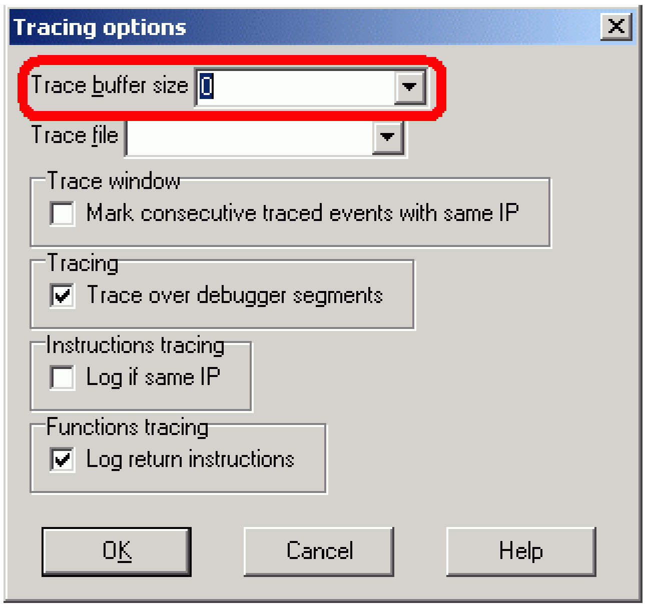
```
chars[]    - average = 3
integers[] - average = 1054228
```

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

What is "tracing" ?

Tracing allows you to record various information during the execution of an application. We will call this tracing information "trace events".

IDA memorizes such trace events in a "**trace buffer**". The size of this **trace buffer** can be either limited (in this case, newer trace events will overwrite older ones) or unlimited (you may need a lot of memory). In our case, as our application is rather small, we will specify an unlimited trace buffer: select *Tracing options* in the *Tracing* submenu of the *Debugger* main menu, and set *Trace buffer size* to 0.



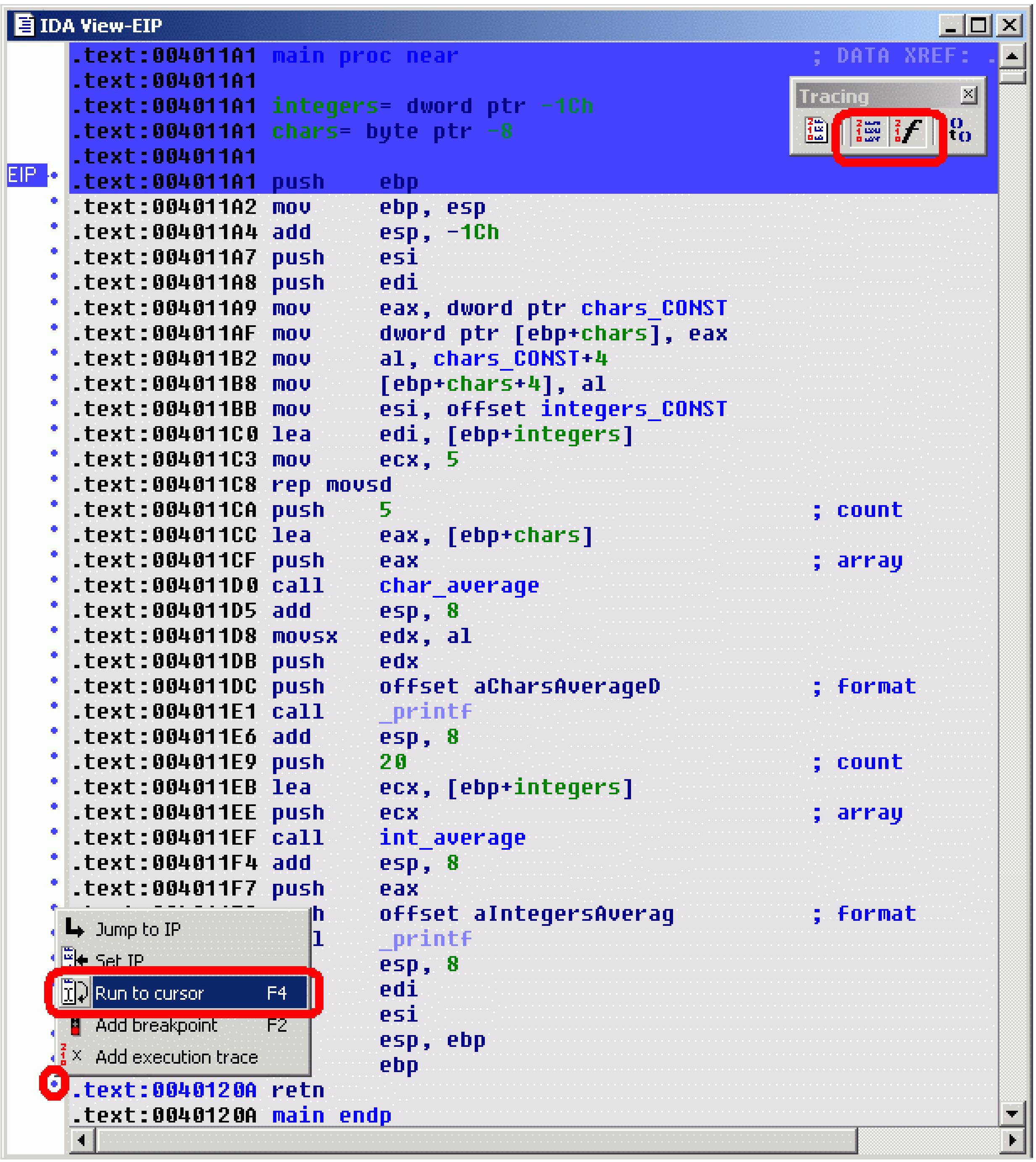
Trace buffer size: 0
Trace file:
Trace window
[ ] Mark consecutive traced events with same IP
Tracing
[✓] Trace over debugger segments
Instructions tracing
[ ] Log if same IP
Functions tracing
[✓] Log return instructions

IDA offers different tracing mechanisms:

- **Instructions tracing:** IDA will record the execution of each instruction, and save the resulting register values. By using this information, it can determine the execution flow of the application, and detect registers which were modified by a given instruction. The computer running the IDA Pro interface will be called the "debugger client".
- **Functions tracing:** IDA will record all function calls and function returns.
- **Read/Write-Write-Execute tracing:** IDA will record all access to a specified address. Internally, Read/Write, Write and Execute tracings are nothing more than non-stopping breakpoints.

For each tracing mechanism, related trace events will be added to the trace buffer. Trace events can also be stored in a text file, if specified in the *Tracing options* dialog box.

To locate the bug in our code, we will record all our program's instructions, function calls, and function returns. We don't want to record the execution of instructions preceding our *main()* function, so we put the cursor on the *main()* function's start (at address 0x004011A1) and we press the F4 key to start and run the application to this address. We then enable instructions and functions tracing by pushing the appropriate icons in the *Tracing* toolbar. Finally, we continue the execution until we reach the *main()* function's end (at address 0x0040120A). Note the *Run to* command is now also available in both disassembly views and arrows panel dots popup menus.

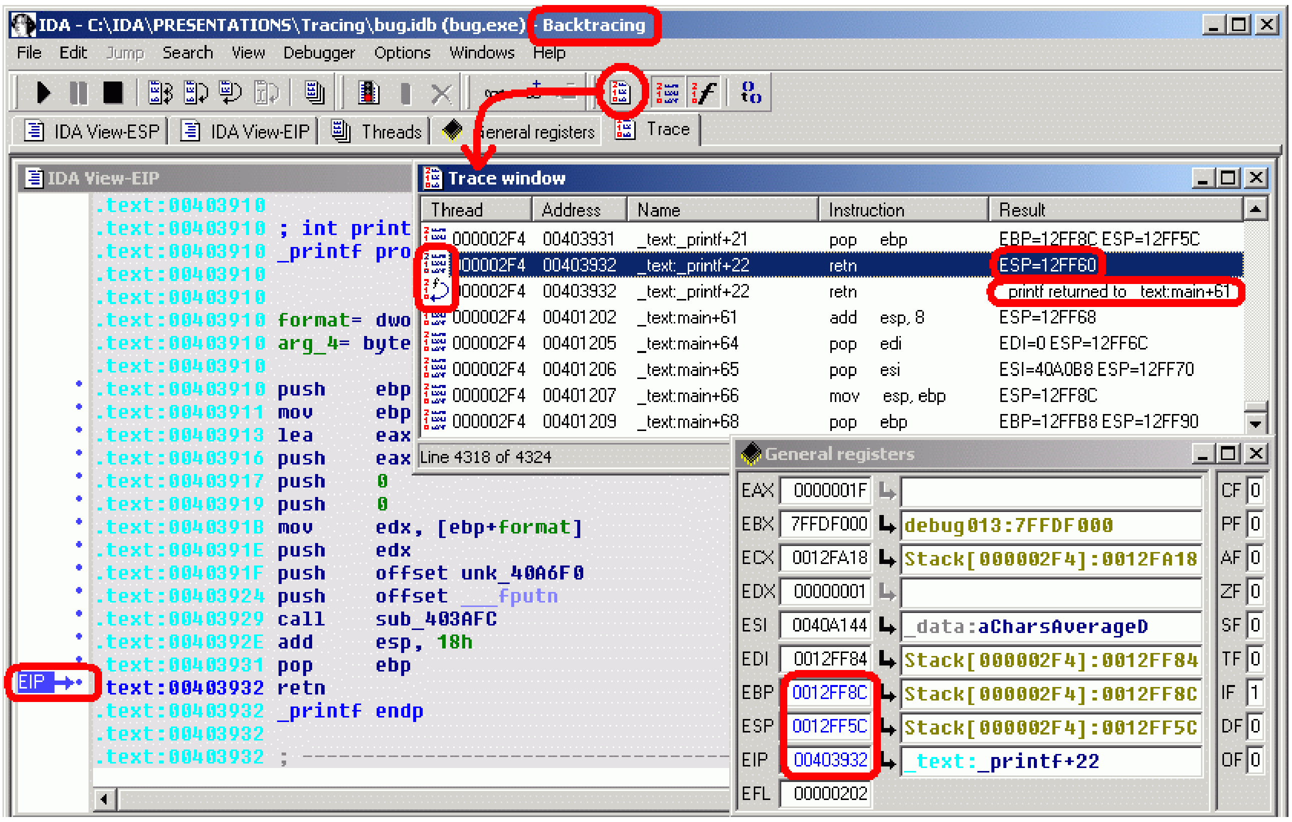


```
.text:004011A1 main proc near                              ; DATA XREF: .
.text:004011A1
.text:004011A1 integers= dword ptr -1Ch
.text:004011A1 chars= byte ptr -8
.text:004011A1
.text:004011A1 push    ebp
.text:004011A2 mov     ebp, esp
.text:004011A4 add     esp, -1Ch
.text:004011A7 push    esi
.text:004011A8 push    edi
.text:004011A9 mov     eax, dword ptr chars_CONST
.text:004011AF mov     dword ptr [ebp+chars], eax
.text:004011B2 mov     al, chars_CONST+4
.text:004011B8 mov     [ebp+chars+4], al
.text:004011BB mov     esi, offset integers_CONST
.text:004011C0 lea     edi, [ebp+integers]
.text:004011C3 mov     ecx, 5
.text:004011C8 rep movsd
.text:004011CA push    5                                   ; count
.text:004011CC lea     eax, [ebp+chars]
.text:004011CF push    eax                                 ; array
.text:004011D0 call    char_average
.text:004011D5 add     esp, 8
.text:004011D8 movsx   edx, al
.text:004011DB push    edx
.text:004011DC push    offset aCharsAverageD               ; format
.text:004011E1 call    _printf
.text:004011E6 add     esp, 8
.text:004011E9 push    20                                  ; count
.text:004011EB lea     ecx, [ebp+integers]
.text:004011EE push    ecx                                 ; array
.text:004011EF call    int_average
.text:004011F4 add     esp, 8
.text:004011F7 push    eax
                  h    offset aIntegersAverag              ; format
                  l    _printf
                       esp, 8
                       edi
                       esi
                       esp, ebp
                       ebp
.text:0040120A retn
.text:0040120A main endp
```

IDA has now memorized the execution of our program. To view the resulting trace events, we click on the *Trace window* button in the *Tracing* toolbar.

If we click on a trace event in the *Trace window*, IDA updates various information on the screen to represent the state of the program when the trace event was recorded. the following items are particularly interesting:

- **IDA's titlebar:** the *Backtracing* word indicates that information on the screen reflect information from a previously recorded trace event.
- **Trace event icons** (in the first column of the *Trace window*): indicate the type of the recorded trace event: instruction execution, function call, function return, ... For example, we observe two different recorded trace events at the end of the *_printf()* function: one represents the instruction , the second represents the function return trace event.
- **Result column** (in the *Trace window*): will contain specific information related to the trace event: in the case of an instruction trace event, it will display registers which were modified by the instruction. Note the IP register is never displayed, as it is usually modified by all instructions.
- **Register arrows** (in the arrows panel of disassembly views): will reflect the value of the register before the instruction was run.
- **Registers windows:** for instruction trace events, each registers window will display the value of the registers before the instruction was run, and the latest modified register values (which are logically the same as the ones in the *Result* column of the previous instruction trace event).


IDA - C:\IDA\PRESENTATIONS\Tracing\bug.idb (bug.exe) - Backtracing
File Edit Jump Search View Debugger Options Windows Help
IDA View-ESP
IDA View-EIP
Threads
General registers
Trace
IDA View-EIP
Trace window
EIP

```
.text:00403910
.text:00403910 ; int print
.text:00403910 _printf pro
.text:00403910
.text:00403910
.text:00403910 format= dwo
.text:00403910 arg_4= byte
.text:00403910
.text:00403910 push    ebp
.text:00403911 mov     ebp
.text:00403913 lea     eax
.text:00403916 push    eax
.text:00403917 push    0
.text:00403919 push    0
.text:0040391B mov     edx, [ebp+format]
.text:0040391E push    edx
.text:0040391F push    offset unk_40A6F0
.text:00403924 push    offset ___fputn
.text:00403929 call    sub_403AFC
.text:0040392E add     esp, 18h
.text:00403931 pop     ebp
.text:00403932 retn
.text:00403932 _printf endp
.text:00403932
.text:00403932 ; ------------------------------------------
```

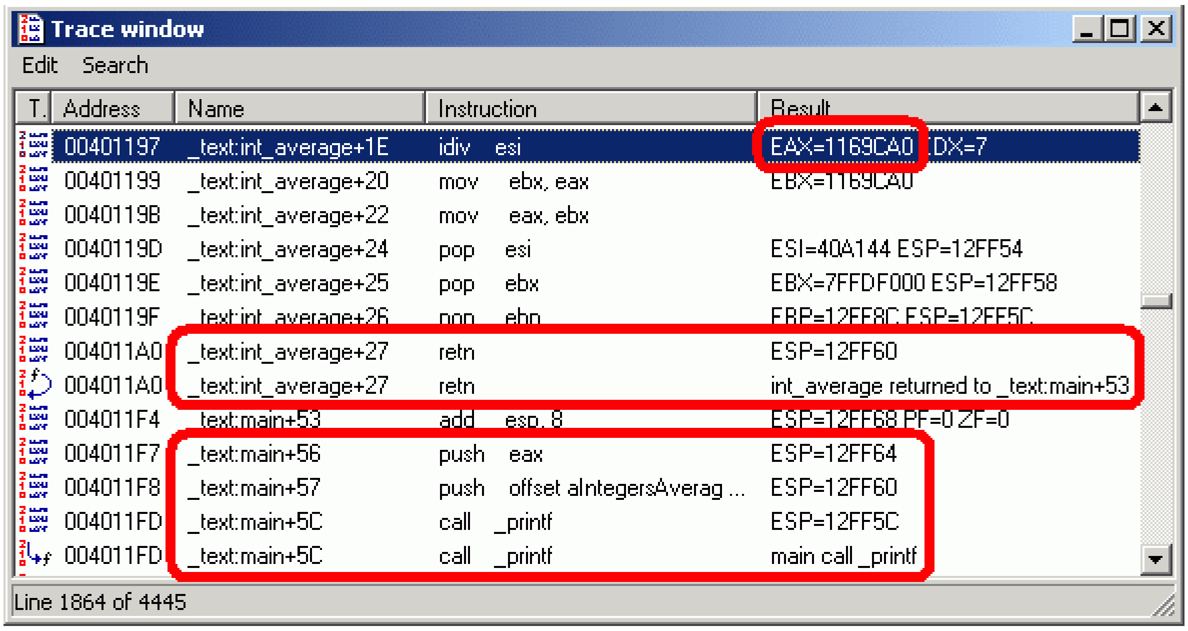
Thread	Address	Name	Instruction	Result
000002F4	00403931	_text:_printf+21	pop ebp	EBP=12FF8C ESP=12FF5C
000002F4	00403932	_text:_printf+22	retn	ESP=12FF60
000002F4	00403932	_text:_printf+22	retn	printf returned to text:main+61
000002F4	00401202	_text:main+61	add esp, 8	ESP=12FF68
000002F4	00401205	_text:main+64	pop edi	EDI=0 ESP=12FF6C
000002F4	00401206	_text:main+65	pop esi	ESI=40A0B8 ESP=12FF70
000002F4	00401207	_text:main+66	mov esp, ebp	ESP=12FF8C
000002F4	00401209	_text:main+68	pop ebp	EBP=12FFB8 ESP=12FF90

Line 4318 of 4324

General registers

EAX	0000001F		CF	0
EBX	7FFDF000	debug013:7FFDF000	PF	0
ECX	0012FA18	Stack[000002F4]:0012FA18	AF	0
EDX	00000001		ZF	0
ESI	0040A144	_data:aCharsAverageD	SF	0
EDI	0012FF84	Stack[000002F4]:0012FF84	TF	0
EBP	0012FF8C	Stack[000002F4]:0012FF8C	IF	1
ESP	0012FF5C	Stack[000002F4]:0012FF5C	DF	0
EIP	00403932	_text:_printf+22	OF	0
EFL	00000202			

Remember the strange average value computed for integers ? It was 18259104 (0x1169CA0 in hexadecimal). The *Trace window*'s *Search* command in the *Search* menu should find at least one trace event referencing this value. And indeed, if we start the search we find this value at the *int_average+1E* address:


Trace window

Edit Search

T.	Address	Name	Instruction	Result
	00401197	_text:int_average+1E	idiv esi	EAX=1169CA0 EDX=7
	00401199	_text:int_average+20	mov ebx, eax	EBX=1169CA0
	0040119B	_text:int_average+22	mov eax, ebx	
	0040119D	_text:int_average+24	pop esi	ESI=40A144 ESP=12FF54
	0040119E	_text:int_average+25	pop ebx	EBX=7FFDF000 ESP=12FF58
	0040119F	_text:int_average+26	pop ebp	EBP=12FF8C ESP=12FF5C
	004011A0	_text:int_average+27	retn	ESP=12FF60
	004011A0	_text:int_average+27	retn	int_average returned to _text:main+53
	004011F4	_text:main+53	add esp, 8	ESP=12FF68 PF=0 ZF=0
	004011F7	_text:main+56	push eax	ESP=12FF64
	004011F8	_text:main+57	push offset aIntegersAverag ...	ESP=12FF60
	004011FD	_text:main+5C	call _printf	ESP=12FF5C
	004011FD	_text:main+5C	call _printf	main call _printf

Line 1864 of 4445

By observing the trace, let's try to understand how the application uses this buggy value:

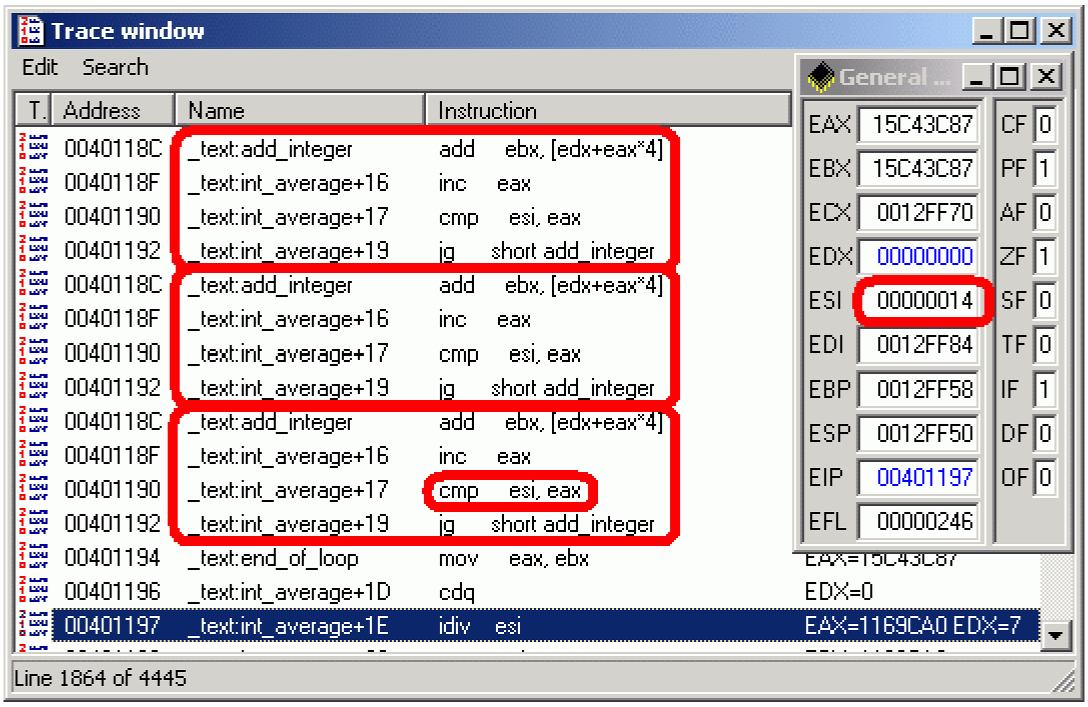
at *int_average+1E* : we found an "*idiv esi*" instruction which results in EAX register containing the buggy value.

at *int_average+27*: the *int_average()* function returns to its caller (the *main()* function), and the buggy value is used as function return value (the average of our integers).

at *main+5C* : the buggy average (stored in the EAX register) is then printed on the screen using the *_printf()* function.

This value represents the badly computed average.

Now, let's observe instructions run just before we reach the *idiv* instruction.



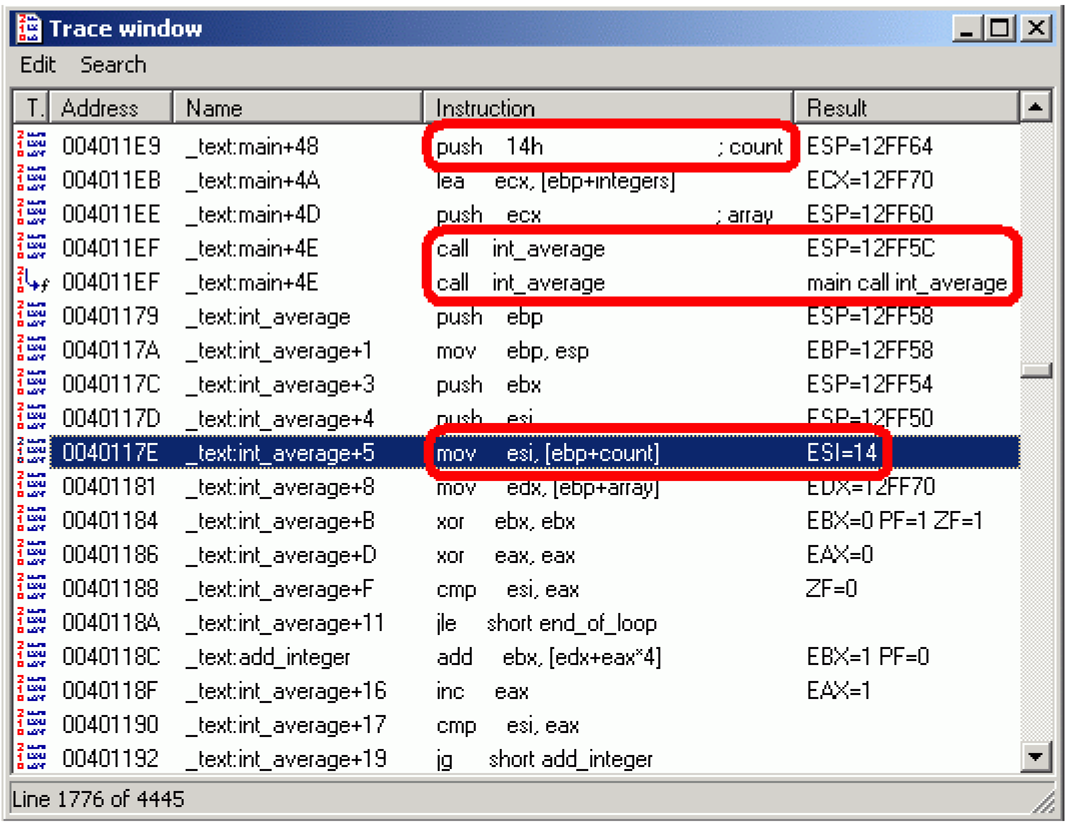
```
Trace window
Edit   Search
T. Address   Name                   Instruction
   0040118C  _text:add_integer      add   ebx, [edx+eax*4]
   0040118F  _text:int_average+16   inc   eax
   00401190  _text:int_average+17   cmp   esi, eax
   00401192  _text:int_average+19   jg    short add_integer
   0040118C  _text:add_integer      add   ebx, [edx+eax*4]
   0040118F  _text:int_average+16   inc   eax
   00401190  _text:int_average+17   cmp   esi, eax
   00401192  _text:int_average+19   jg    short add_integer
   0040118C  _text:add_integer      add   ebx, [edx+eax*4]
   0040118F  _text:int_average+16   inc   eax
   00401190  _text:int_average+17   cmp   esi, eax
   00401192  _text:int_average+19   jg    short add_integer
   00401194  _text:end_of_loop      mov   eax, ebx          EAX=15C43C87
   00401196  _text:int_average+1D   cdq                     EDX=0
   00401197  _text:int_average+1E   idiv  esi               EAX=1169CA0 EDX=7
Line 1864 of 4445

General ...
EAX 15C43C87  CF 0
EBX 15C43C87  PF 1
ECX 0012FF70  AF 0
EDX 00000000  ZF 1
ESI 00000014  SF 0
EDI 0012FF84  TF 0
EBP 0012FF58  IF 1
ESP 0012FF50  DF 0
EIP 00401197  OF 0
EFL 00000246
```

We can observe a small loop which adds integers to compute our average. The trace window displays here the three latest iterations of this loop. If we look at this loop's exit condition, we see it compares the ESI register to the EAX register. So what does contain the ESI register ? The registers window answers our question and indicates ESI contains the 0x14 value (20 in decimal) at the end of the loop. Don't we expect the loop to iterate 5 times, rather than 20?

The bug uncovered.

Let's now browse instructions preceding the buggy loop to understand where this strange ESI value comes from.



```
T. Address   Name                   Instruction                       Result
   004011E9  _text:main+48          push   14h              ; count   ESP=12FF64
   004011EB  _text:main+4A          lea    ecx, [ebp+integers]        ECX=12FF70
   004011EE  _text:main+4D          push   ecx              ; array   ESP=12FF60
   004011EF  _text:main+4E          call   int_average                ESP=12FF5C
   004011EF  _text:main+4E          call   int_average                main call int_average
   00401179  _text:int_average      push   ebp                        ESP=12FF58
   0040117A  _text:int_average+1    mov    ebp, esp                   EBP=12FF58
   0040117C  _text:int_average+3    push   ebx                        ESP=12FF54
   0040117D  _text:int_average+4    push   esi                        ESP=12FF50
   0040117E  _text:int_average+5    mov    esi, [ebp+count]           ESI=14
   00401181  _text:int_average+8    mov    edx, [ebp+array]           EDX=12FF70
   00401184  _text:int_average+B    xor    ebx, ebx                   EBX=0 PF=1 ZF=1
   00401186  _text:int_average+D    xor    eax, eax                   EAX=0
   00401188  _text:int_average+F    cmp    esi, eax                   ZF=0
   0040118A  _text:int_average+11   jle    short end_of_loop
   0040118C  _text:add_integer      add    ebx, [edx+eax*4]           EBX=1 PF=0
   0040118F  _text:int_average+16   inc    eax                        EAX=1
   00401190  _text:int_average+17   cmp    esi, eax
   00401192  _text:int_average+19   jg     short add_integer
Line 1776 of 4445
```

Before the first loop iteration (starting at *int_average+F*), we directly see that ESI gets its value from the *count* argument of our *int_average()* function. If we now look at how our *int_average()* function gets called, with the help of IDA's PIT (Parameter Identification and Tracking) technology, we easily locate the *push 14h* instruction, passing the erroneous *count* argument. Now, by looking closer at our C source code, we better understand the 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...

DataRescue SA/NV

40 Bld Piercot

4000 Liège, Belgium

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