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.

```c
#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:

\[
\begin{align*}
\text{chars[]} & \quad \text{average} = 3 \\
\text{integers[]} & \quad \text{average} = 1054228
\end{align*}
\]

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.

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.
Instructions and functions tracing.

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.
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).
Searching the trace buffer.

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:

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.
The buggy loop.

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

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.

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...