Using Trace Replayer Debugger and Managing Traces in IDA

Copyright 2014 Hex-Rays SA

Table of contents

Introduction....2
  Quick Overview....2
  Following this tutorial....2
  Supplied files....2
Replaying and managing traces....2
  Recording traces....2
  Working with traces....6
  Loading an overlay and viewing differences in flow....9
  Diffing traces....18
  Reverting the diff....19
  Replaying traces....20
Summary....24

Introduction

Quick Overview

The trace replayer is an IDA pseudo debugger plugin that appeared first in IDA 6.3. This plugin can replay execution traces recorded with any debugger backend in IDA, such as local Win32 or Linux debuggers, WinDbg, remote GDB debugger, etc...

Following this tutorial

This tutorial was created using the Linux version of IDA and a Linux binary as target. However, it can be followed on any supported platform (MS Windows, Mac OS X and Linux) by setting up remote debugging. Please refer to the IDA online help for more information regarding remote debugging.

Supplied files

Among with the tutorial the following files are also provided at http://www.hex-rays.com/products/ida/support/tutorials/replayer/ida-replayer-tutorial.tar.gz

File name	SHA1	Description
intoverflow.c	6424d3100e3ab1dd3fceae53c7d925364cea75c5	Program's source code.
intoverflow.elf	69a0889b7c09ec5c293702b3b50f55995a1a2daa	Linux ELF32 program.
no_args.trc	773837c2b212b4416c8ac0249859208fd30e2209	IDA binary trace file version 1
second_run.trc	4e0a5effa34f805cc50fe40bc0e19b78ad1bb7c4	IDA binary trace file version 1
crash.trc	f0ee851b298d7709e327d8eee81657cf0beae69b	IDA binary trace file version 1

Replaying and managing traces

Recording traces

Before using the trace replayer plugin we will need to record an execution trace of a program. We will use the following toy vulnerable program as an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int foo(char *arg, int size)
{
    char *buf;
```

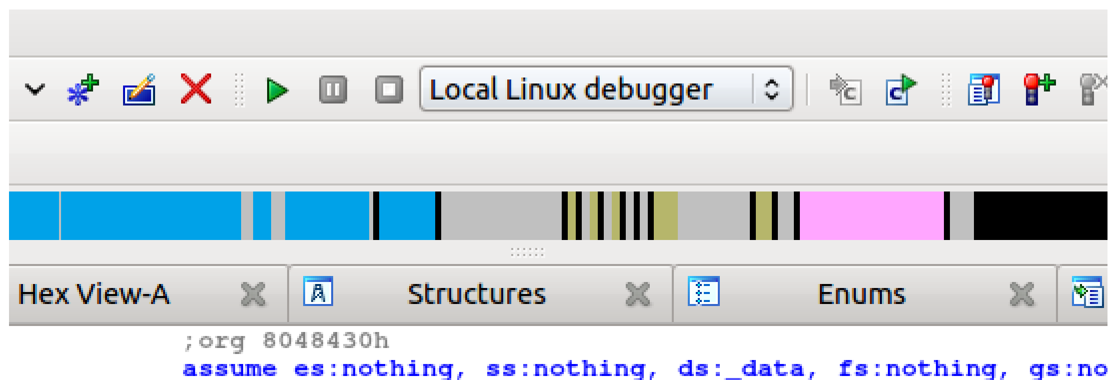
```
    if ( strlen(arg) > size )
    {
        printf("Too big!\n");
        return 1;
    }

    buf = malloc(size);
    strcpy(buf, arg);
    printf("Buffer is %s\n", buf);
    free(buf);
    return 0;
}

int main(int argc, char **argv)
{
    if ( argc != 3 )
    {
        printf("Invalid number of arguments!\n");
        return 2;
    }

    return foo(argv[1], atoi(argv[2]));
}
```

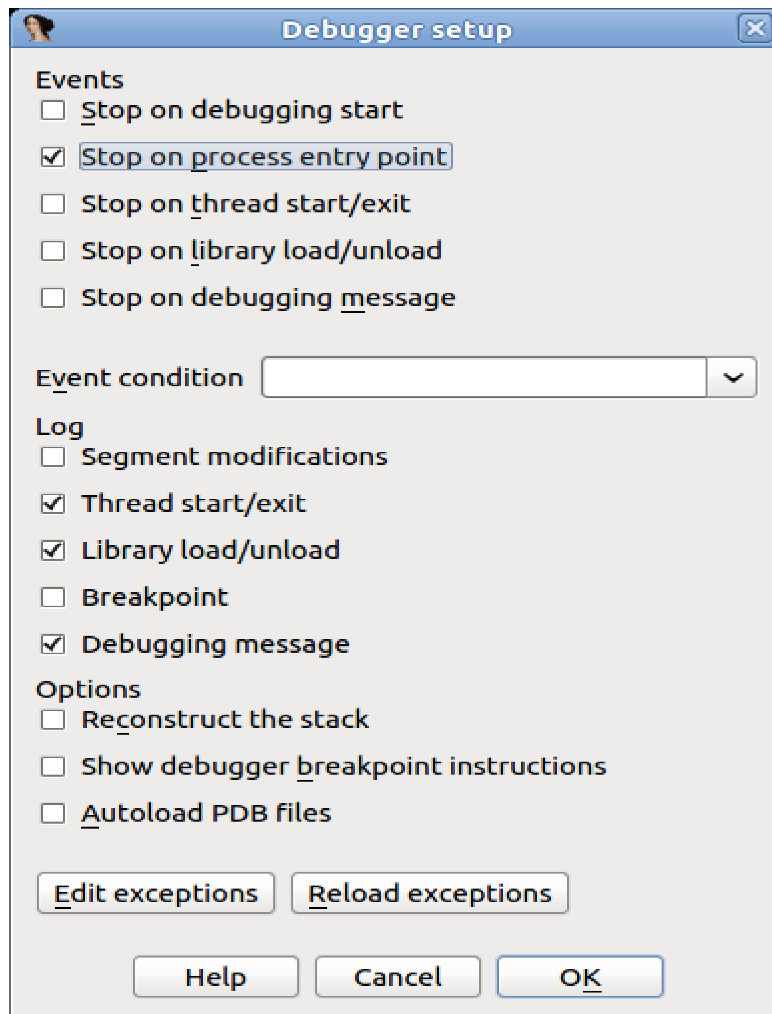
Please compile this sample program (in this example, we used GCC compiler for Linux) or use the supplied ELF binary, open the binary in IDA and wait until the initial analysis completes. When done, select a suitable debugger from the drop down list ("Local Linux debugger", or "Remote Linux debugger" if you're following this tutorial from another platform):



We have two ways of telling IDA to record a trace:

1. Break on process entry point and manually enable tracing at this point.
2. Or put a trace breakpoint at the very first instruction of the program.

In the case we prefer the first approach we will need to click on the menu "Debugger → Debugger Options" and then mark the check box "Stop on process entry point" as shown bellow:



After checking this option press OK and run the program pressing F9. When the entry point is reached, we can select from the menu "Debugger → Tracing" one of the following three options:

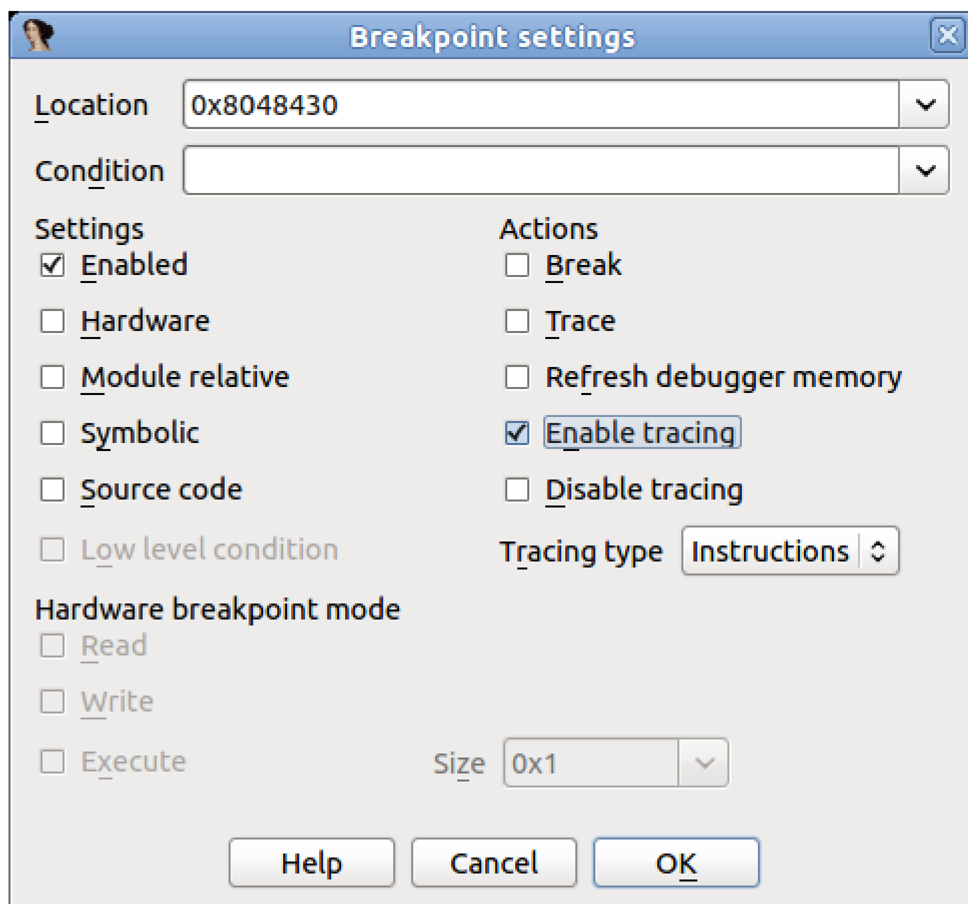
1. Instruction tracing: All instructions executed will be recorded.
2. Function tracing: Only function calls and returns will be recorded.
3. Basic block tracing: Similar to instruction tracing but, instead of single stepping instruction by instruction, IDA will set temporary breakpoints in the end of every known basic block, as well as on function calls.

For this example we will select "Instruction tracing". Check this option and let the program continue by pressing F9. The program will resume execution and finish quickly. Now, we have a recorded trace! To see it, select "Debugger → Tracing → Trace Window". A new tab will open with a content similar to the following:

IDA View-A	Trace window	Hex View-A	Structures

Thread	Address	Instruction
000047F8	.text:_start	Memory layout changed: 33 segments
000047F8		
000047F8	.text:_start	xor ebp, ebp
000047F8	.text:_start+2	pop esi
000047F8	.text:_start+3	mov ecx, esp
000047F8	.text:_start+5	and esp, 0FFFFFFF0h
000047F8	.text:_start+8	push eax
000047F8	.text:_start+9	push esp
000047F8	.text:_start+A	push edx
000047F8	.text:_start+B	push offset __libc_csu_fini
000047F8	.text:_start+10	push offset __libc_csu_init
000047F8	.text:_start+15	push ecx
000047F8	.text:_start+16	push esi
000047F8	.text:_start+17	push offset main
000047F8	.text:_start+1C	call ___libc_start_main
000047F8	.plt:___libc_start_main	jmp ds:off_804A004

As previously stated, there are two ways to record traces: enabling it manually, or using an “Enable tracing” breakpoint. To set such a breakpoint we will go to the program's entry point (Ctrl+E) and put a breakpoint (F2) in the very first instruction. Then right click on the new breakpoint and select “Edit breakpoint”. In the dialog check the option “Enable tracing” and then select the desired “Tracing type” (for this example, we'll use “Instructions”):

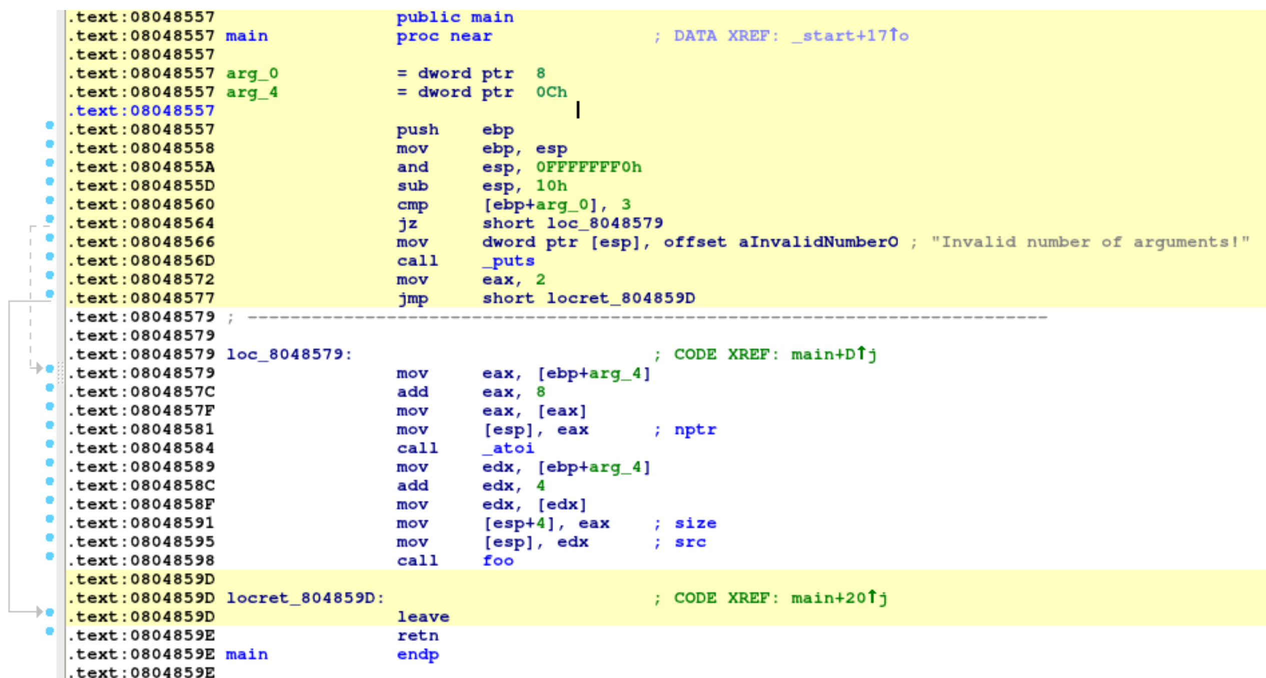


Remove the “Stop on process entry point” option we set in the prior example and press F9 to run the program.

This way is more convenient than the first because the tracing is turned on automatically and does not need manual intervention.

Working with traces

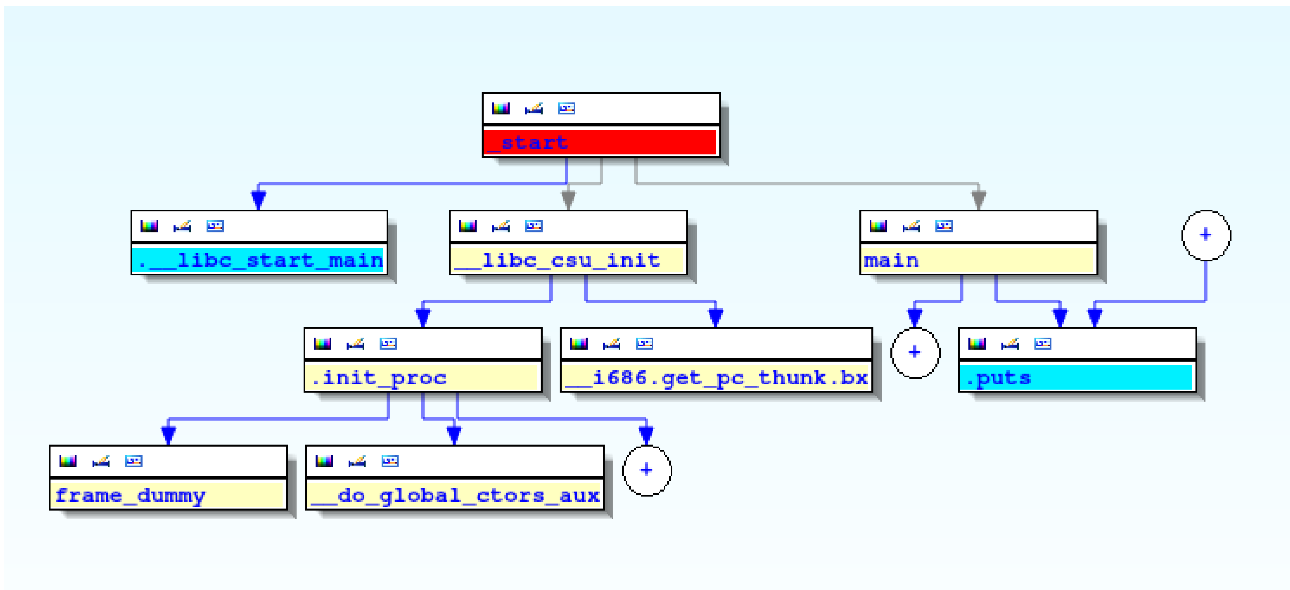
Now we have a new recorded trace, no matter which method we used. What can we do with it? First, we can check which instructions were executed, as they are highlighted in the disassembly, like in the screenshot bellow:



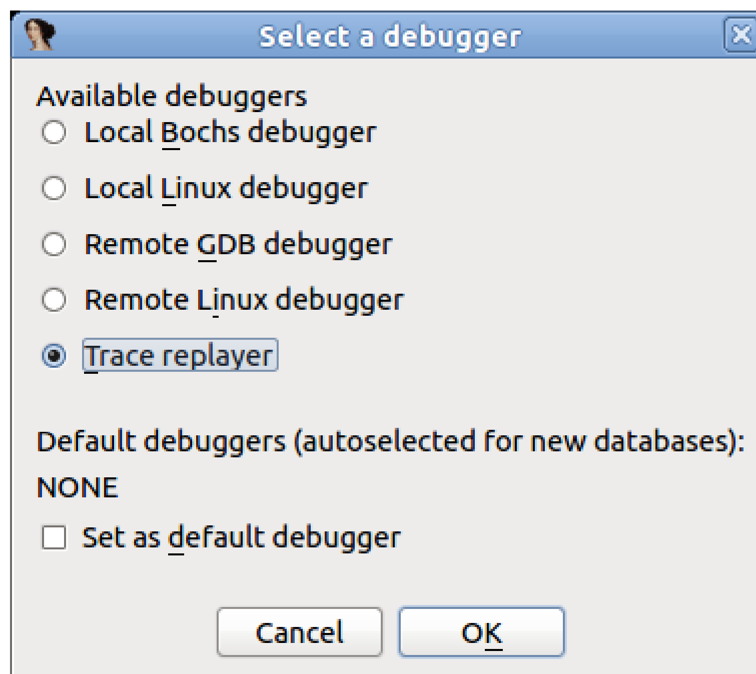
(the highlight color can be changed in “Debugger → Tracing → Tracing Options”)

Highlighting makes it clear which instructions have been executed.

We can also check what functions have been executed (instead of instructions) by opening the “Trace Window” via “Debugger → Tracing → Trace Window”, right clicking on the list and then selecting “Show trace call graph”:

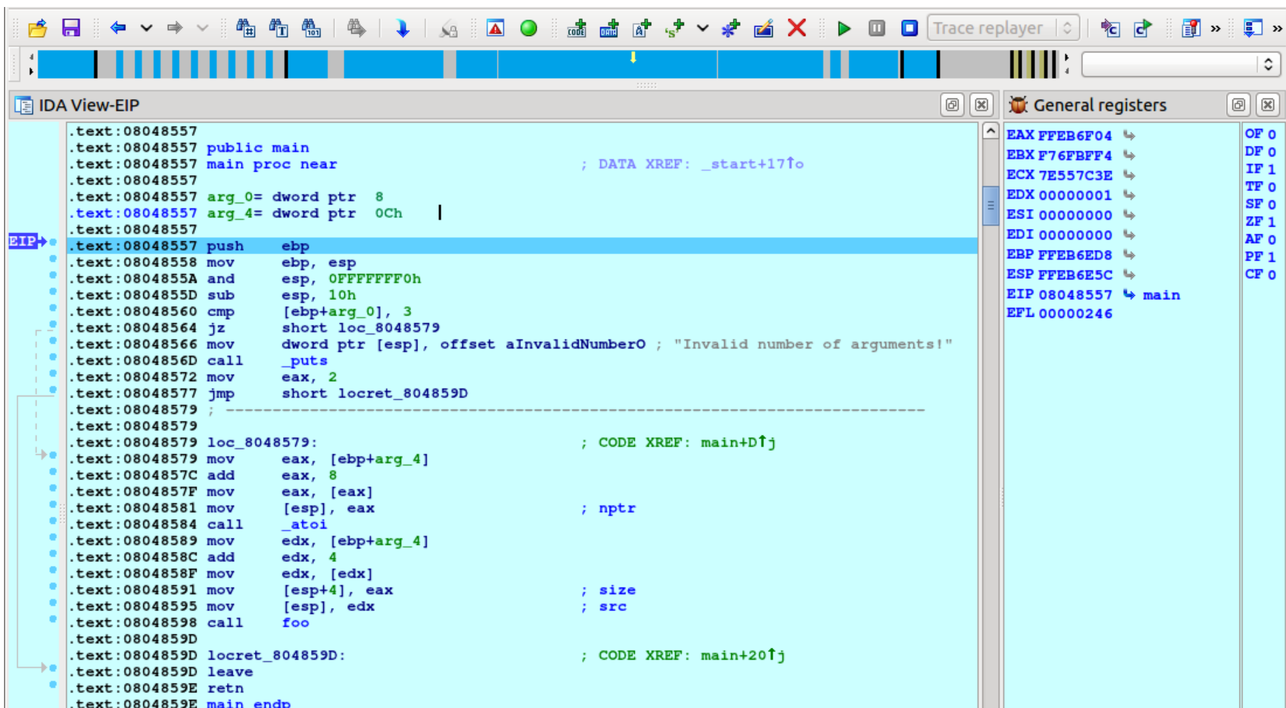


Now let's inspect the register values in order to understand why the check at 0x0848566 doesn't pass. Please select “Debugger → Switch debugger” and in the dialog box click on the “Trace replayer” radio button:

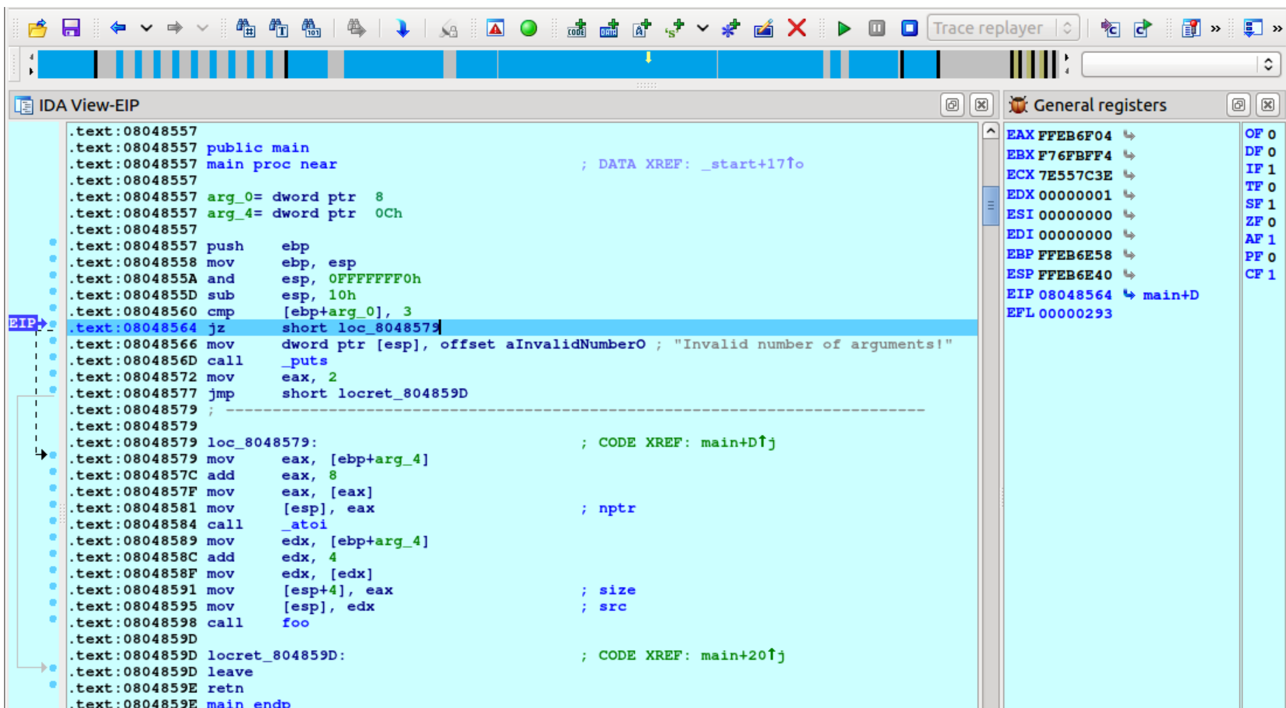


Click OK and press F4 in the first instruction of the “main” function.

The trace replayer will suspend execution at the “main” function and display the register values that were recorded when the program was executed:



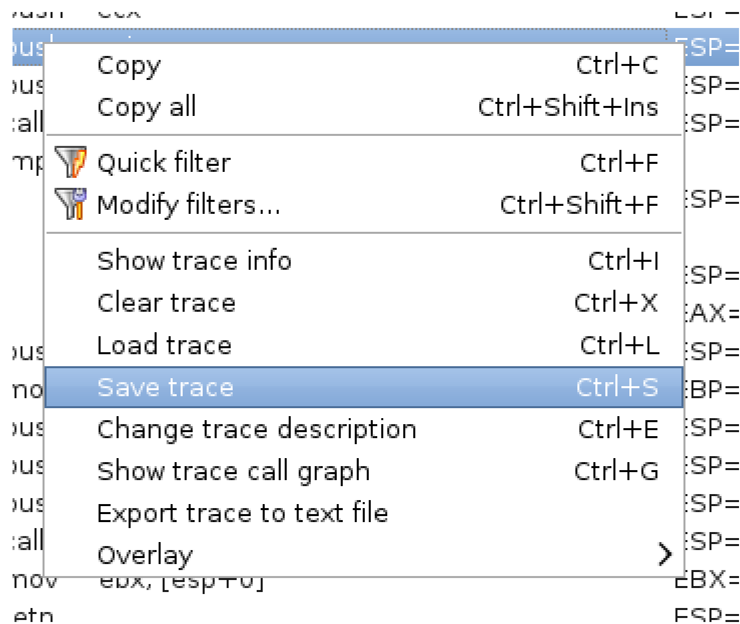
We can single step by pressing F7, as usual. Let us keep pressing F7 until the “jz” instruction is reached:



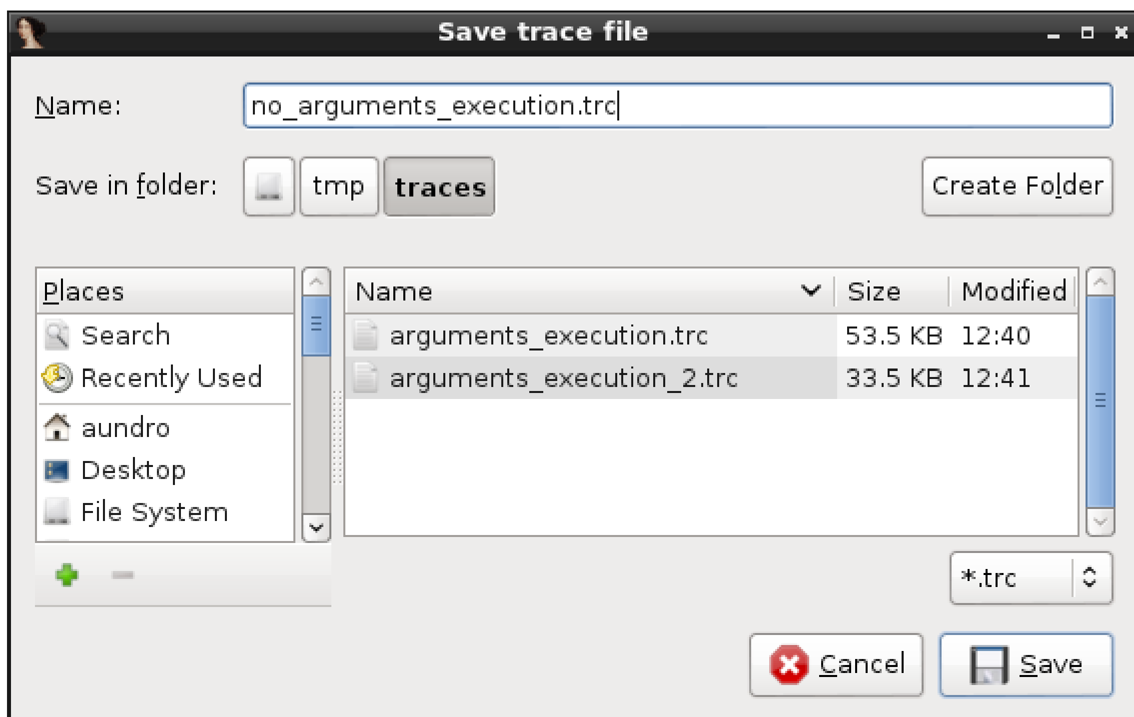
The comparison “cmp [ebp+arg_0], 3” was not successful (ZF=0) so the check does not pass. We need to give to the program two arguments to pass this check and record a new trace.

Loading an overlay and viewing differences in flow

Before doing another run, let's save the first trace to a file. Select “Debugger → Tracing → Trace Window”, right click in the middle of the newly opened tab, and select “Save trace” from the popup menu:



Then save the file:



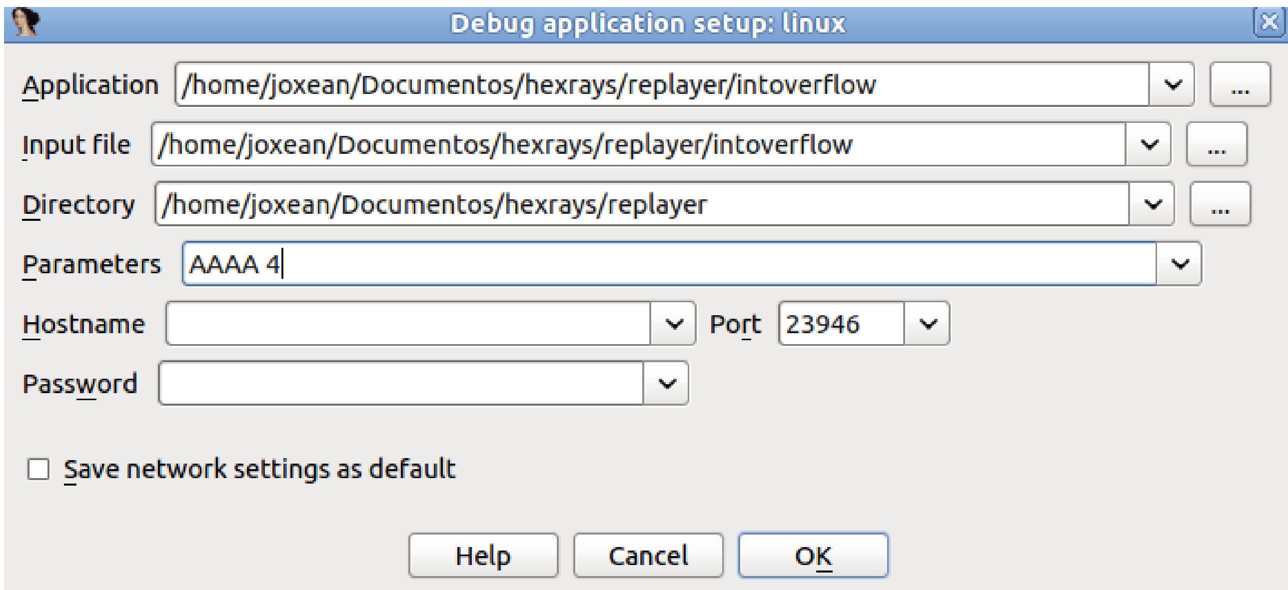
You will also be offered a chance to give the trace a description:

Trace description

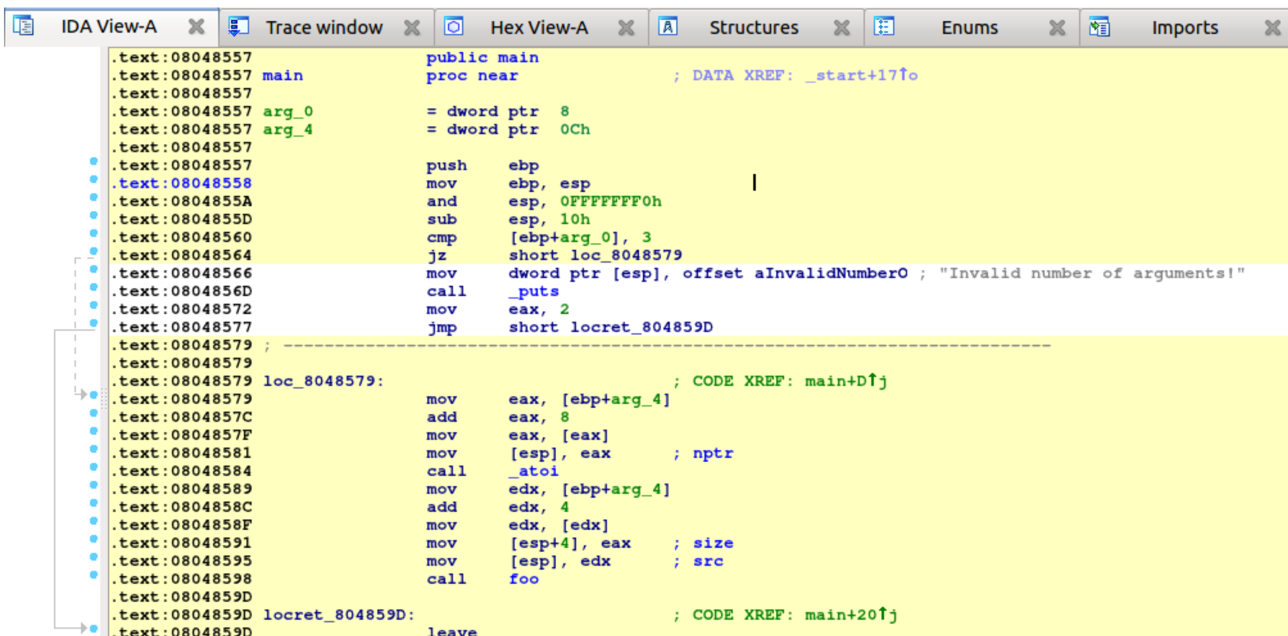
Enter description (or leave empty) Running without arguments.

Cancel OK

Now let's record a new trace but this time we will pass two command line arguments to the program. Select “Debugger → Process Options” and set “AAAA 4” as the arguments:



Close the dialog, revert to the “Local Linux debugger”, and press F9. A new trace will be recorded. If we check the “main” function we will see that different instructions have been executed this time:

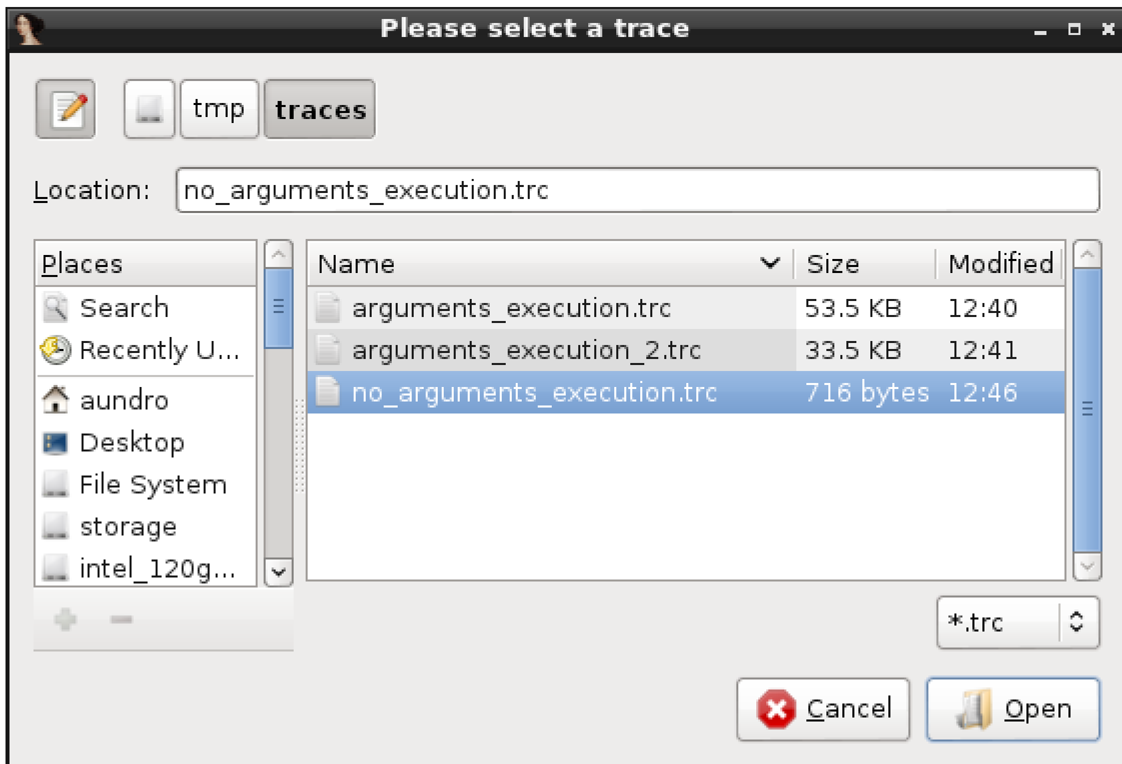


Let's check which instructions are different between the first and the second run.

First, we will need to load the previous trace as “overlay”:

push offset __libc_csu_init ESP=FFDEBC6C
Copy Ctrl+C ESP=FFDEBC68
Copy all Ctrl+Shift+Ins ESP=FFDEBC64
ESP=FFDEBC60
Quick filter Ctrl+F ESP=FFDEBC5C
Modify filters... Ctrl+Shift+F
Show trace info Ctrl+I ESP=FFDEBC58
Clear trace Ctrl+X
Load trace Ctrl+L ESP=FFDEBC54
Save trace Ctrl+S EAX=FFDEBC84 EBX=F779FI
Change trace description Ctrl+E ESP=FFDEBBD8
Show trace call graph Ctrl+G EBP=FFDEBBD8
Export trace to text file ESP=FFDEBBD4
ESP=FFDEBBD0
Overlay > Load overlay Ctrl+Shift+L
call __i686_get_pc_thunk_bx ESP=FFDEBBC8

Select the trace we saved:

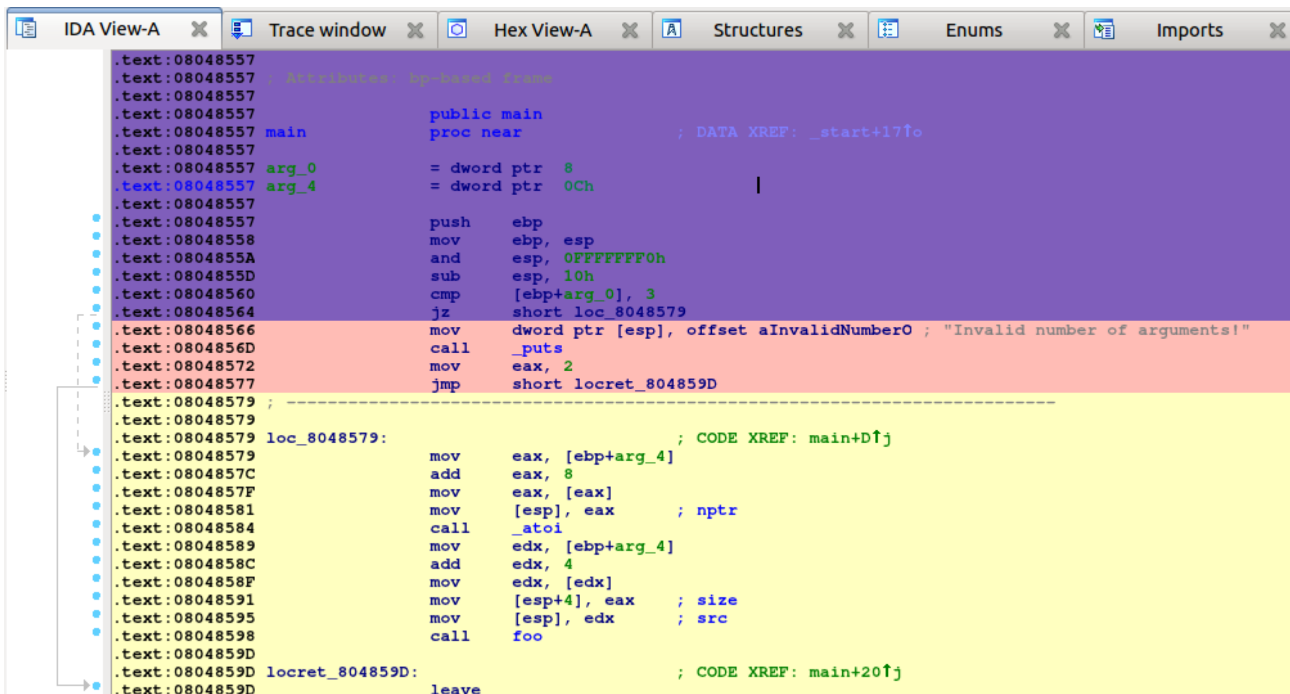


Note that we have now other options in the 'Overlay' submenu, now that there is an overlay present:

Menu item	Shortcut
Copy	Ctrl+C
Copy all	Ctrl+Shift+Ins
Quick filter	Ctrl+F
Modify filters...	Ctrl+Shift+F
Show trace info	Ctrl+I
Clear trace	Ctrl+X
Load trace	Ctrl+L
Save trace	Ctrl+S
Change trace description	Ctrl+E
Show trace call graph	Ctrl+G
Export trace to text file	
Overlay >	

Overlay submenu	Shortcut
Show overlay info	Ctrl+Shift+I
Clear overlay	Ctrl+Shift+X
Load overlay	Ctrl+Shift+L
Subtract overlay	Ctrl+D

Now go back to the disassembly view and check how the disassembly code is highlighted in three different colors:



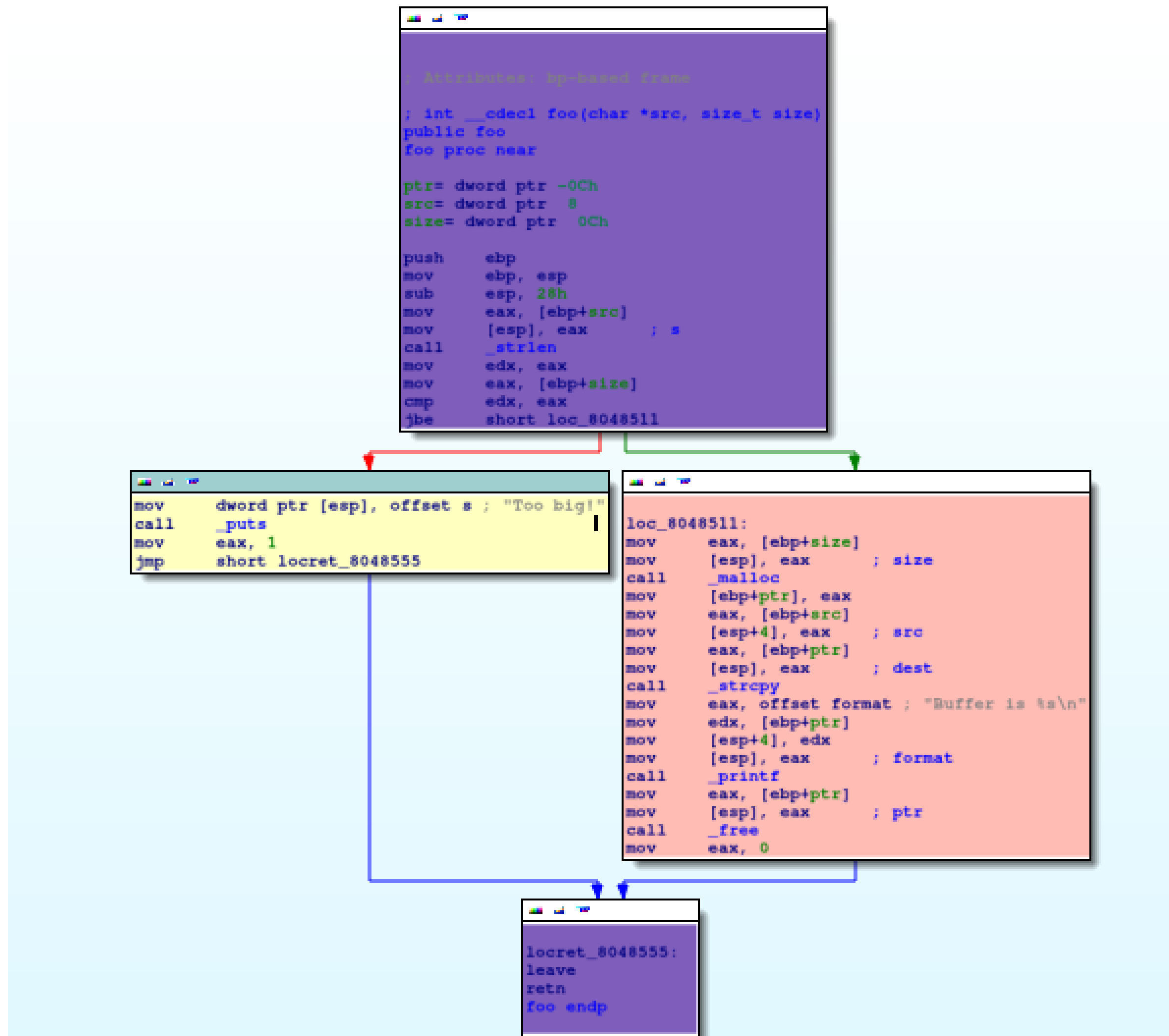
The code highlighted in yellow is the code executed in the current trace (the one listed in the "Trace Window"). The pink code was executed only in the overlay trace. And the code in purple is the code common to both traces. We can immediately see that there is some new code that have been executed, like the calls to **atoi** and **foo**.

Let's go to the “foo” function and see what happened here:

```
IDA View-A    Trace window    Hex View-A    Structures    Enums
.text:080484E4
.text:080484E4 ptr             = dword ptr -0Ch
.text:080484E4 src             = dword ptr  8
.text:080484E4 size            = dword ptr  0Ch
.text:080484E4
.text:080484E4                 push    ebp
.text:080484E5                 mov     ebp, esp
.text:080484E7                 sub     esp, 28h
.text:080484EA                 mov     eax, [ebp+src]
.text:080484ED                 mov     [esp], eax      ; s
.text:080484F0                 call    _strlen
.text:080484F5                 mov     edx, eax
.text:080484F7                 mov     eax, [ebp+size]
.text:080484FA                 cmp     edx, eax
.text:080484FC                 jbe     short loc_8048511
.text:080484FE                 mov     dword ptr [esp], offset s ; "Too big!"
.text:08048505                 call    _puts
.text:0804850A                 mov     eax, 1
.text:0804850F                 jmp     short locret_8048555
.text:08048511 ; ---------------------------------------------------------------------------
.text:08048511
.text:08048511 loc_8048511:                            ; CODE XREF: foo+18↑j
.text:08048511                 mov     eax, [ebp+size]
.text:08048514                 mov     [esp], eax      ; size
.text:08048517                 call    _malloc
.text:0804851C                 mov     [ebp+ptr], eax
.text:0804851F                 mov     eax, [ebp+src]
.text:08048522                 mov     [esp+4], eax    ; src
.text:08048526                 mov     eax, [ebp+ptr]
.text:08048529                 mov     [esp], eax      ; dest
.text:0804852C                 call    _strcpy
.text:08048531                 mov     eax, offset format ; "Buffer is %s\n"
.text:08048536                 mov     edx, [ebp+ptr]
.text:08048539                 mov     [esp+4], edx
.text:0804853D                 mov     [esp], eax      ; format
.text:08048540                 call    _printf
.text:08048545                 mov     eax, [ebp+ptr]
```

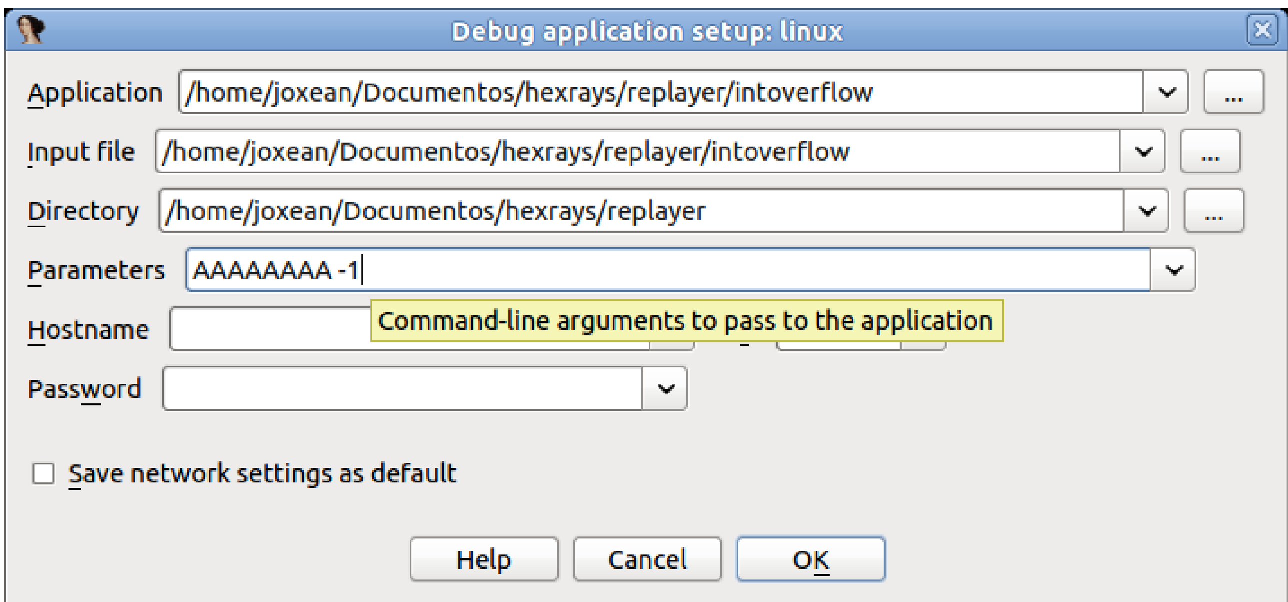
The code in yellow tells us that the check for the size at 0x800484FC passed and the calls to **malloc**, **strcpy** and **printf** were executed. Let's save this trace for later analysis and comparison with the future runs. As before, go to the trace window, right click on the list and select “Save trace”. Set the trace's description to 'Correct execution'.

It's time to record another trace with different arguments to see what happens. For this new execution, we will longer command line arguments (eight “A” characters instead of four). Let's change the arguments in “Debugger → Process Options”, switch back to the “Local Linux debugger”, and run it. We have a new trace. Let's compare it against the previously recorded one. As we did before, go to the “Trace Window”, right click on the list, select “Overlay”, then “Load overlay”, and select the trace with description “Correct execution”.

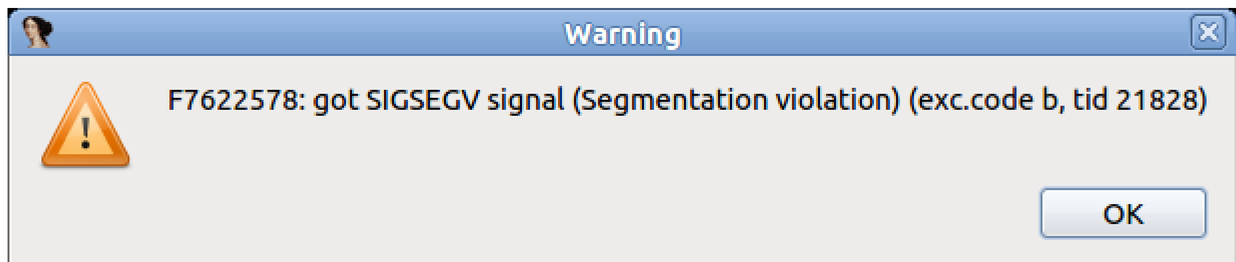


As we see, the code that alerts us the about a too big string was executed (it's highlighted in yellow). Let's save this recorded trace with the “String too big” description. Now we will record one more trace but this time we will use the number “-1” as the second command line argument.

Change the arguments in “Debugger → Process Options” as shown bellow:

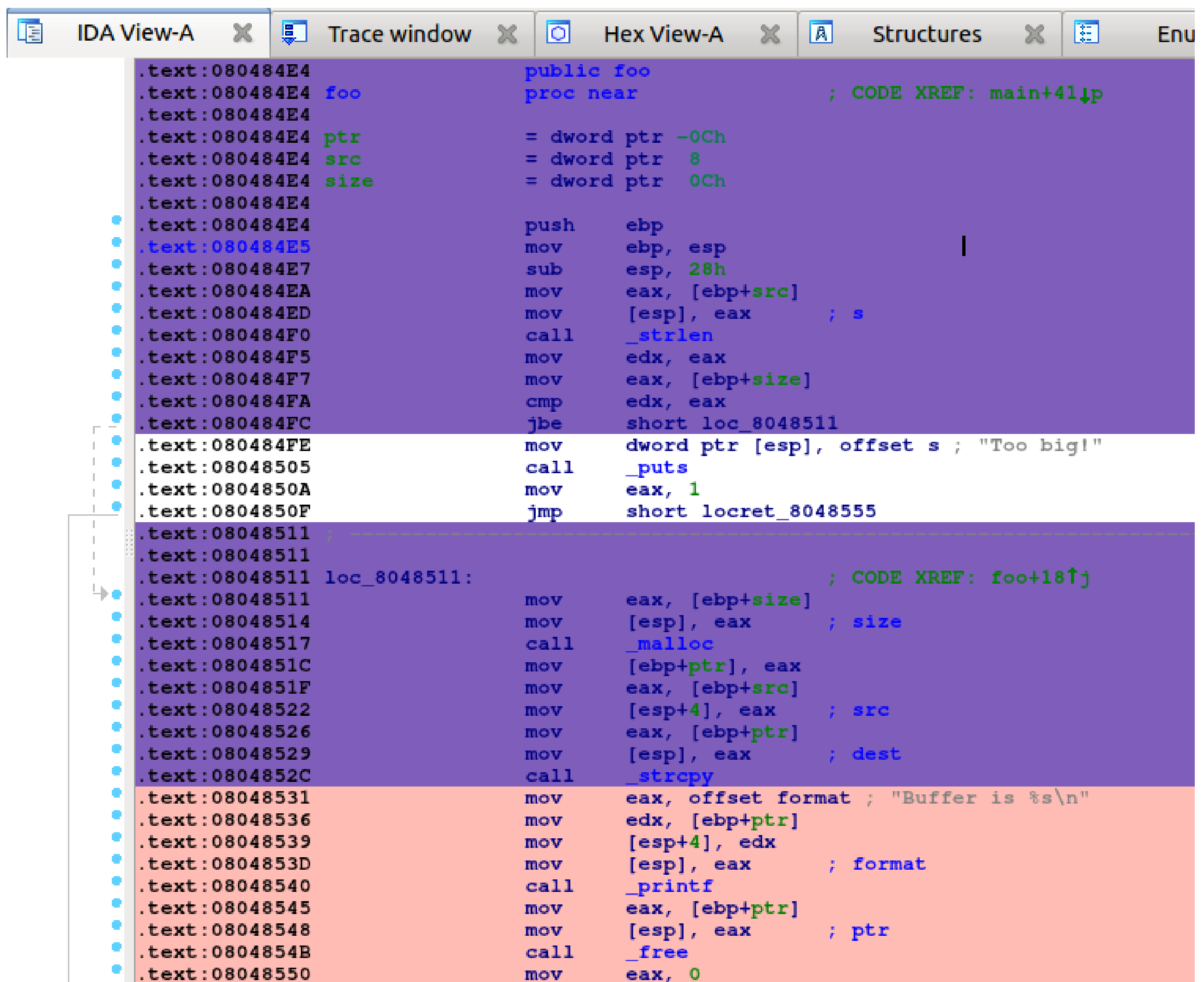


Then switch back again, to the “Local Linux debugger” (or to “Remote Linux debugger” if needed) and run it by pressing F9. The process will crash somewhere in the call to **strcpy**:



Stop the debugger and save this trace (let's call it “Crash”). Then diff this trace against the “Correct execution” trace.

We will see the following in the disassembly view:

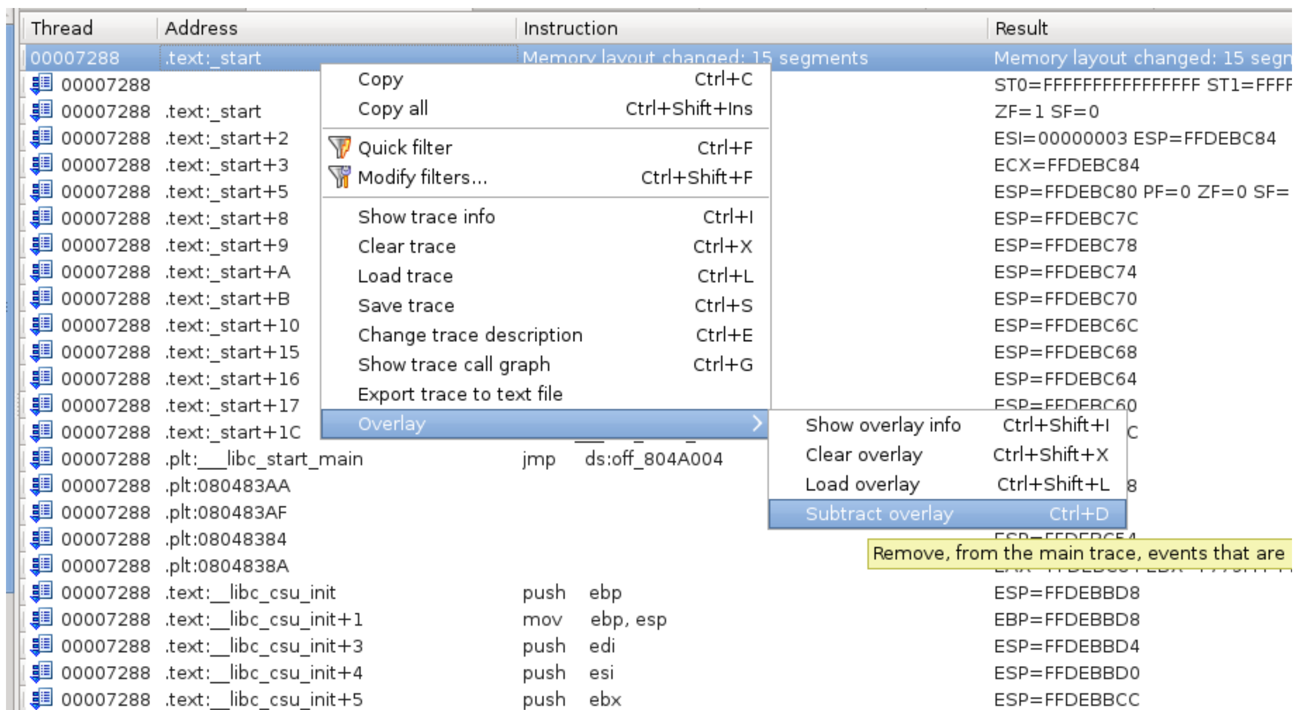


As we see, pretty much the same code as in the previous run was executed until the call to **strcpy**. It's time to replay this last execution and see what happened.

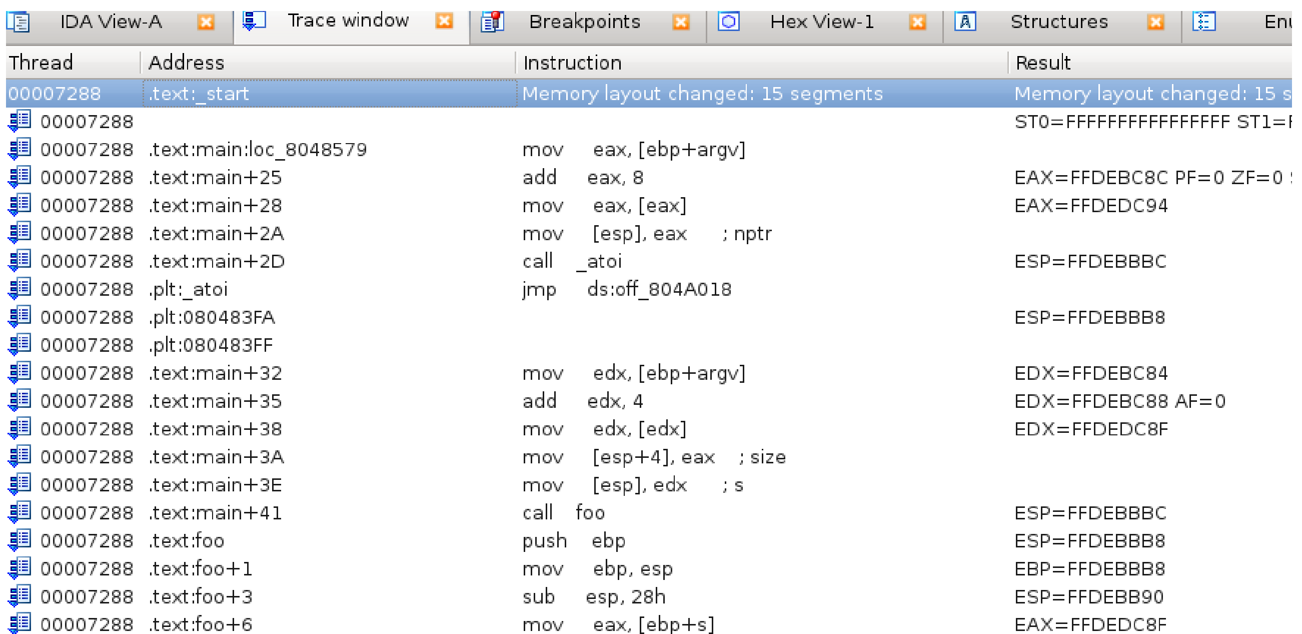
Diffing traces

When both a "main trace" and an "overlay trace" are present, the context menu item "Overlay → Subtract overlay" becomes available.

This allows one to subtract the list of events (e.g., instructions) that are present in the overlay, from the main trace:



Will give the following results:



As you can see, many events that were present in both the overlay & the main trace have been removed. Only those that were only present in the main trace remain.

Reverting the diff

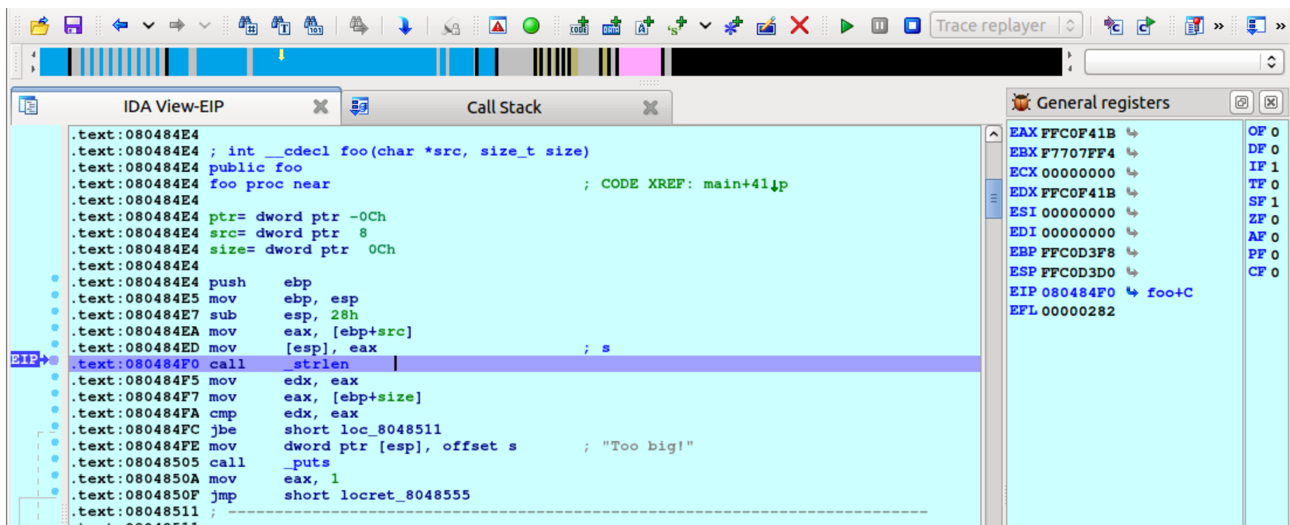
The diffing operation is reversible:

This will restore the main trace as it were, before the contents of the overlay were removed from it.

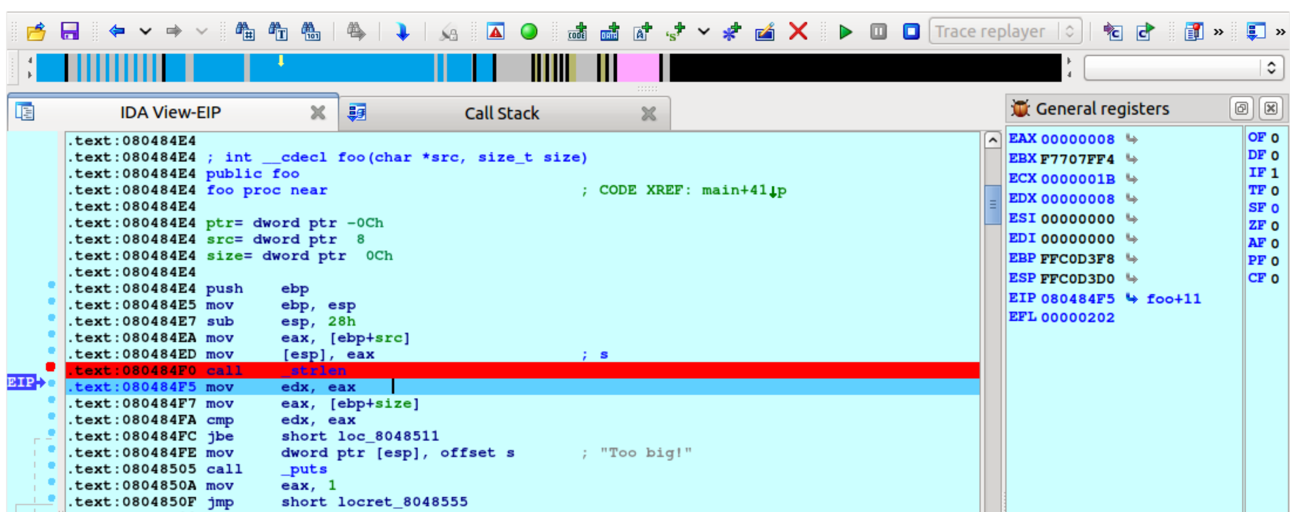
Replaying traces

We know that the program is crashing somewhere in the call to **strcpy** but we don't know why the check at 0x080484FC passes since -1 is smaller than the size of the string (8 bytes). Let's put a breakpoint at the call to **strlen** at 0x080484F0, switch to the “Trace replayer” debugger, and "run" the program by pressing F9. Please note that we do not really run the program, we are merely replaying a previously recorded trace.

The debugger will stop at the **strlen** call:

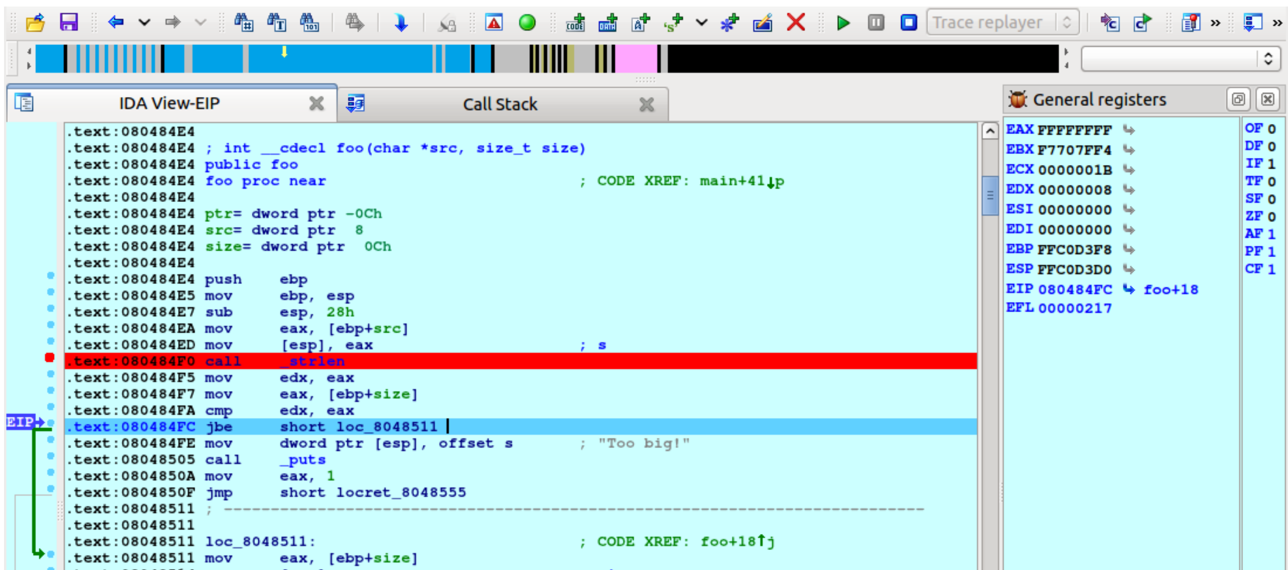


In the trace replayer we can use all usual debugging commands like “run to cursor” (F4), “single step” (F7), or “step over” (F8). Let's press F8 to step over the **strlen** call and check the result:

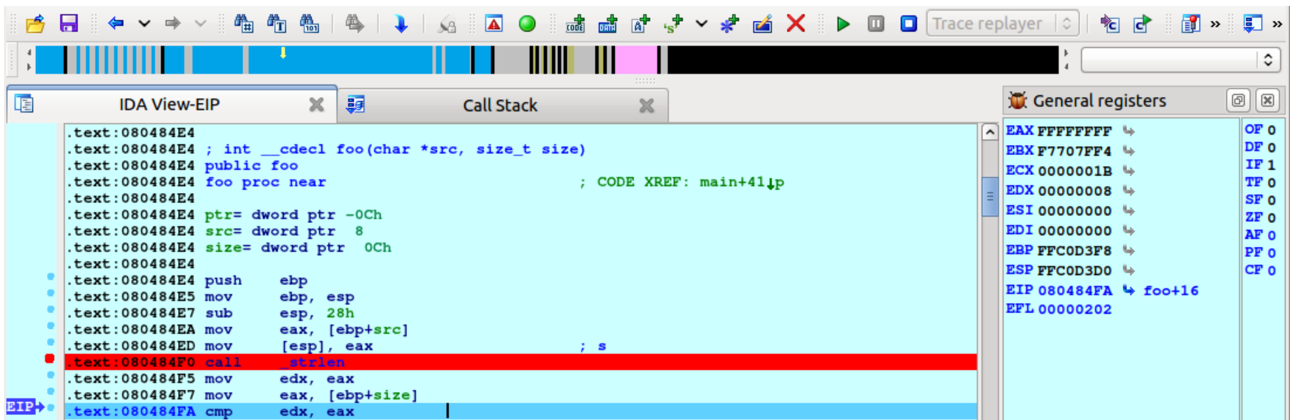


It returns 8 as expected. Now move to the address 0x080484FC and press F4 or right click on this address, select “Set IP”, and press F7 (we need to inform the replayer plugin that we changed the current execution instruction in order to refresh all the register values). The difference between “Run to” (F4) and “Set IP” is that “Run to” will replay all events happened until that point but “Set IP” will directly move to the nearest trace event happened at this address (if it's in the recorded trace, of course).

Regardless of how we moved to this point IDA will display the following:



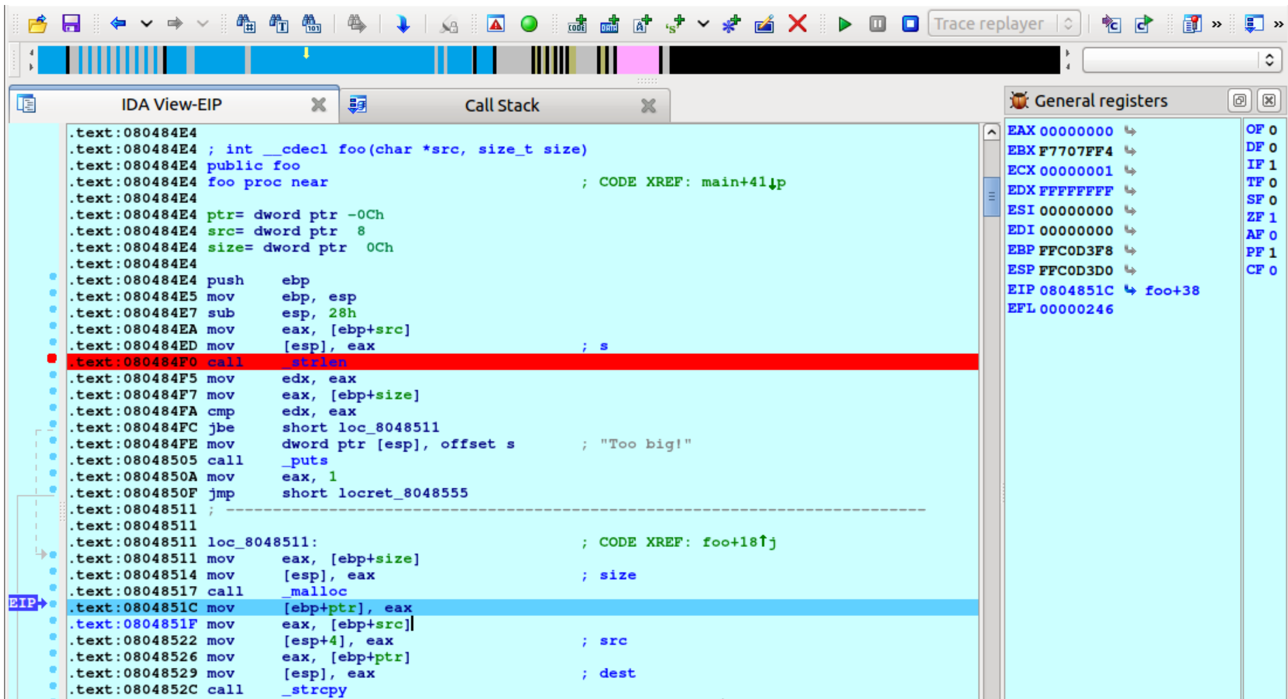
As we see, the jump was taken because CF was set to 1 in the previous instruction ("cmp edx, eax"). Let's step back to this instruction to see what values were compared. Select "Debugger → Step back" from the menu:



The flags are reset to 0 and we can see that EAX (0xFFFFFFFF) and EDX (8) are compared. Press F7 to step one instruction again and you will notice CF changes to 1. The instruction JBE performs an unsigned comparison between 8 and 0xFFFFFFFF and, as 8 <= 0xFFFFFFFF, the check passes. We just discovered the cause of the bug.

Let's continue analyzing it a bit more. Scroll down until the call to malloc at 0x08048517, right click, choose “Set IP”, and press F7 (or simply press F4). As we see, the argument given to malloc is 0xFFFFFFFF (4 GB).

Press F8 to step over the function call:



Obviously, **malloc** can not allocate so much memory and returns NULL. However, the program does not check for this possibility and tries to copy the contents of the given buffer to the address 0, resulting in a crash.

Summary

In this tutorial we showed you the basics of trace management and the trace replayer module in IDA. We hope you enjoy this new feature. Happy debugging!