The PIN Tracer module

Index

Introduction

The PIN tracer is a remote debugger plugin used to record execution traces. It allows to record traces on Linux and Windows (x86 and x86_64) from any of the supported IDA platforms (Windows, Linux and MacOSX). Support for MacOSX targets is not yet available.

PIN support for MacOSX

Recording traces on MacOSX target is not supported yet. Versions equal or prior to 2.13-62141 does not have support for the API PIN_SpawnInternalThread, which is needed by IDA to communicate with the PIN tool.

However, it's possible to record traces from a Linux or Windows target using the MacOSX version of IDA.

Building the PIN tool

Before using the PIN tracer the PIN tool module (distributed only in source code form) must be built as the Intel PIN license disallows redistributing PIN tools in binary form.

First, you will need to download the IDA SDK for the version of IDA you are using. The SDK can be found there: http://www.hex-rays.com/products/ida/support/download.shtml

The building process of the PIN tool is different for Windows and Linux. To build it in Windows:

1. Download PIN from http://www.pintool.org , and unpack it on your hard drive.
   *WARNING: the PIN tools are a little sensitive to spaces in paths. Therefore, we recommend unpacking in a no-space path. E.g., "C:\pin", but **not** "C:\Program Files (x86)\".*
2. Install Visual Studio. It is possible to build the PIN tool with the Express version of Visual Studio for C++.
3. Download the IDA pintool sources from:
   https://www.hex-rays.com/products/ida/support/freefiles/idapin$(IDAMAJMIN).zip (*)
   pintool 6.9 and higher should be build with PIN version 3.0 and higher, for earlier versions of pintool you should use PIN build 65163.
4. Unpack the .zip file into /path/to/pin/source/tools/
5. Open /path/to/pin/source/tools/idapin/IDADBG.sln in Visual Studio, select the correct build configuration (either Win32 or x64) and build the solution.

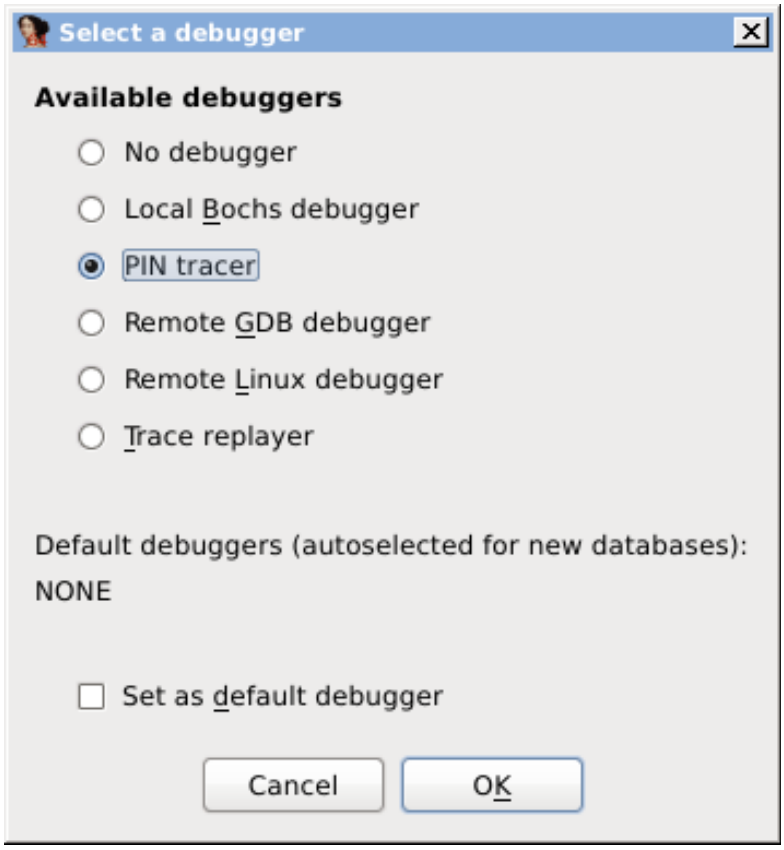
To build the Linux version of the PIN tool:

1. Download PIN from http://www.pintool.org , and unpack it on your hard drive.
   *WARNING: the PIN tools are a little sensitive to spaces in paths. Therefore, we recommend unpacking in a no-space path. E.g., "/home/bobby/pin", but **not** "/home/bobby/latest pin/".*
2. Install GCC 3.4 or later
3. Download the IDA pintool sources from:
   https://www.hex-rays.com/products/ida/support/freefiles/idapin$(IDAMAJMIN).zip (*)
4. Unpack the .zip file into /path/to/pin/source/tools/
5. Open a console, and do the following (only for versions of PIN prior to 3.0):
   1. cd /path/to/pin/ia32/runtime
   2. ln -s libelf.so.0.8.13 libelf.so
   3. cd /path/to/pin/intel64/runtime
   4. ln -s libelf.so.0.8.13 libelf.so
   5. cd /path/to/pin/source/tools/Utils
   6. ls testGccVersion 2>/dev/null || ln -s ../testGccVersion testGccVersion
   7. cd /path/to/pin/source/tools/idapin
      make TARGET=ia32 For building the x86 version.
      make TARGET=intel64 For the x64 version.

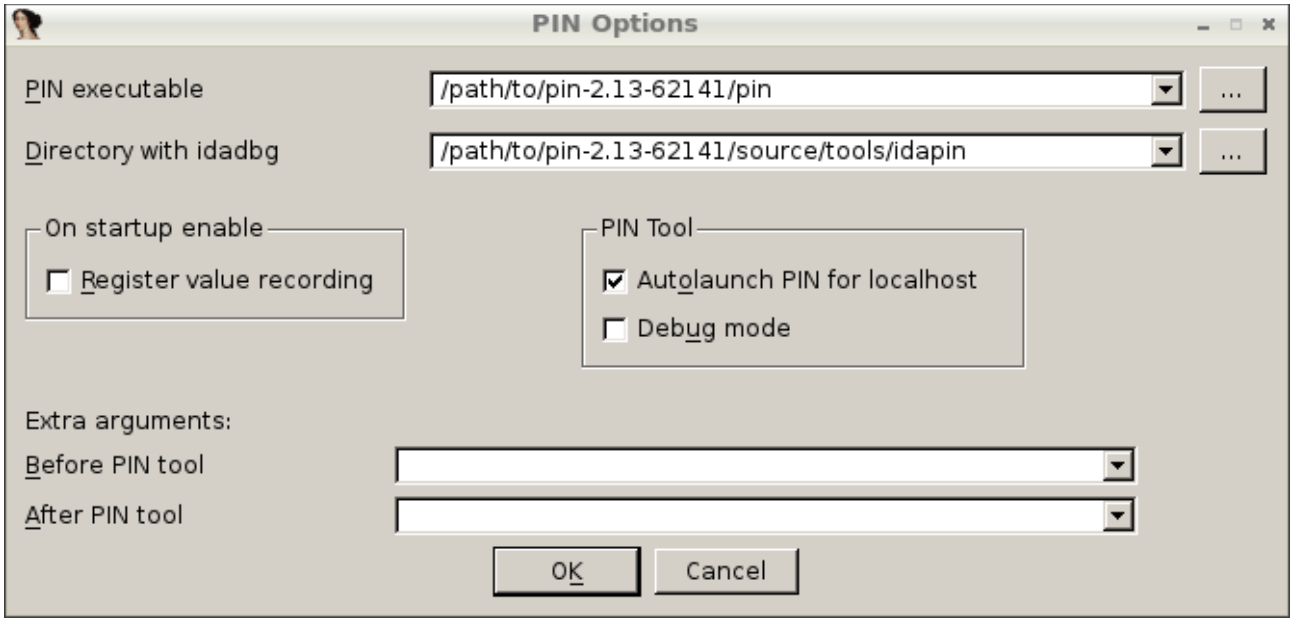
(*) Where '$(IDAMAJMIN)' is the IDA version major/minor. E.g., for IDA 6.9, the final URL would be: https://www.hex-rays.com/products/ida/support/freefiles/idapin69.zip
Pintool 6.9 is compatible with versions 6.5-6.8 of IDA so currently you can use it for them.

Using the PIN tracer

Once the PIN tool module is built we can use it in IDA. Let's take the “ls” binary from Ubuntu for this tutorial. Open the binary “ls” in IDA and wait for the initial analysis to finish. When it's done select the PIN tracer module from the debuggers drop down list or via Debugger → Select debugger:



After selecting the PIN tracer module select the menu Debugger → Debugger options → Set specific options. The following new dialog will be displayed:



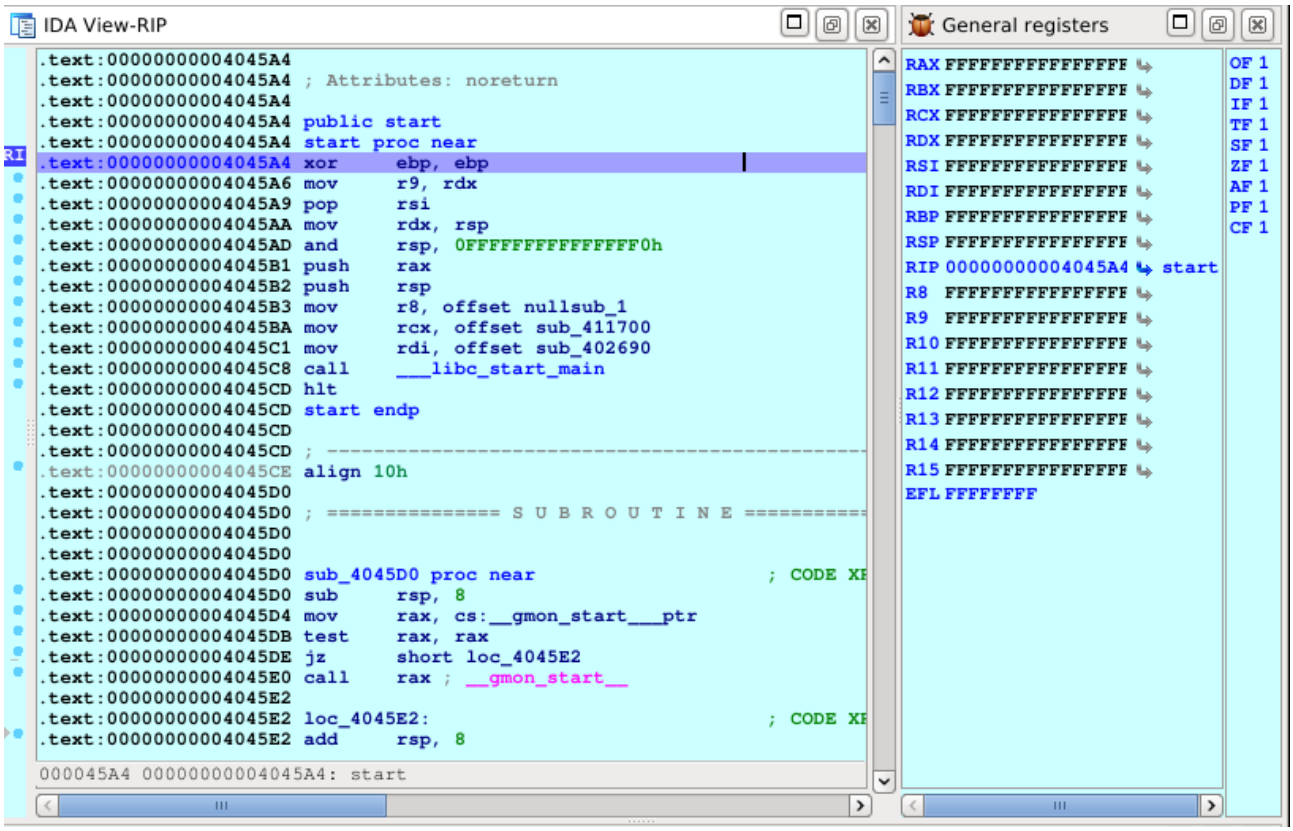
In this dialog at least the following options are mandatory:

1. PIN executable: This is the full path to the PIN binary (including the “pin.exe” or “pin.sh” file name). In old versions “pin.sh” may not exist – in this case you should use “pin” instead.
2. Directory with idadbg: This is the directory where the idadbg.so or idadbg.dll PIN tool resides. Please note that *only the directory must be specified*.

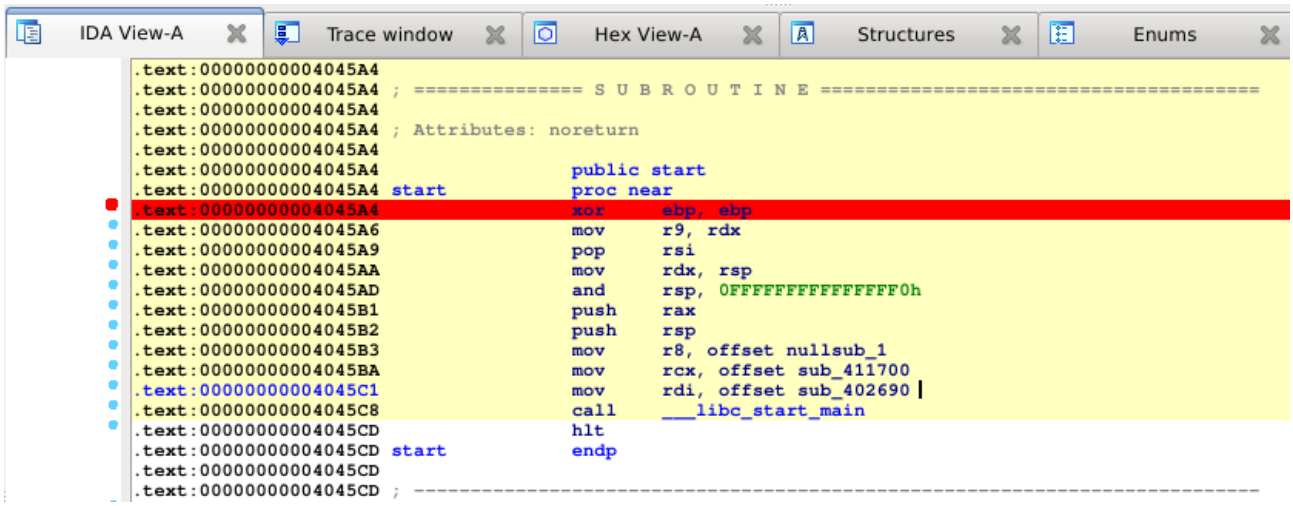
Fill the form with the correct paths and leave everything as is. Press OK in this dialog and put a breakpoint in the very first instruction of the entry point of the “ls” binary as in the following example:

```
.text:00000000004045A4 ; Attributes: noreturn
.text:00000000004045A4
.text:00000000004045A4                 public start
.text:00000000004045A4 start           proc near
.text:00000000004045A4                 xor     ebp, ebp
.text:00000000004045A6                 mov     r9, rdx
.text:00000000004045A9                 pop     rsi
.text:00000000004045AA                 mov     rdx, rsp
.text:00000000004045AD                 and     rsp, 0FFFFFFFFFFFFFFF0h
.text:00000000004045B1                 push    rax
.text:00000000004045B2                 push    rsp
.text:00000000004045B3                 mov     r8, offset nullsub_1
.text:00000000004045BA                 mov     rcx, offset sub_411700
.text:00000000004045C1                 mov     rdi, offset sub_402690
.text:00000000004045C8                 call    ___libc_start_main
.text:00000000004045CD                 hlt
.text:00000000004045CD start           endp
.text:00000000004045CD
```

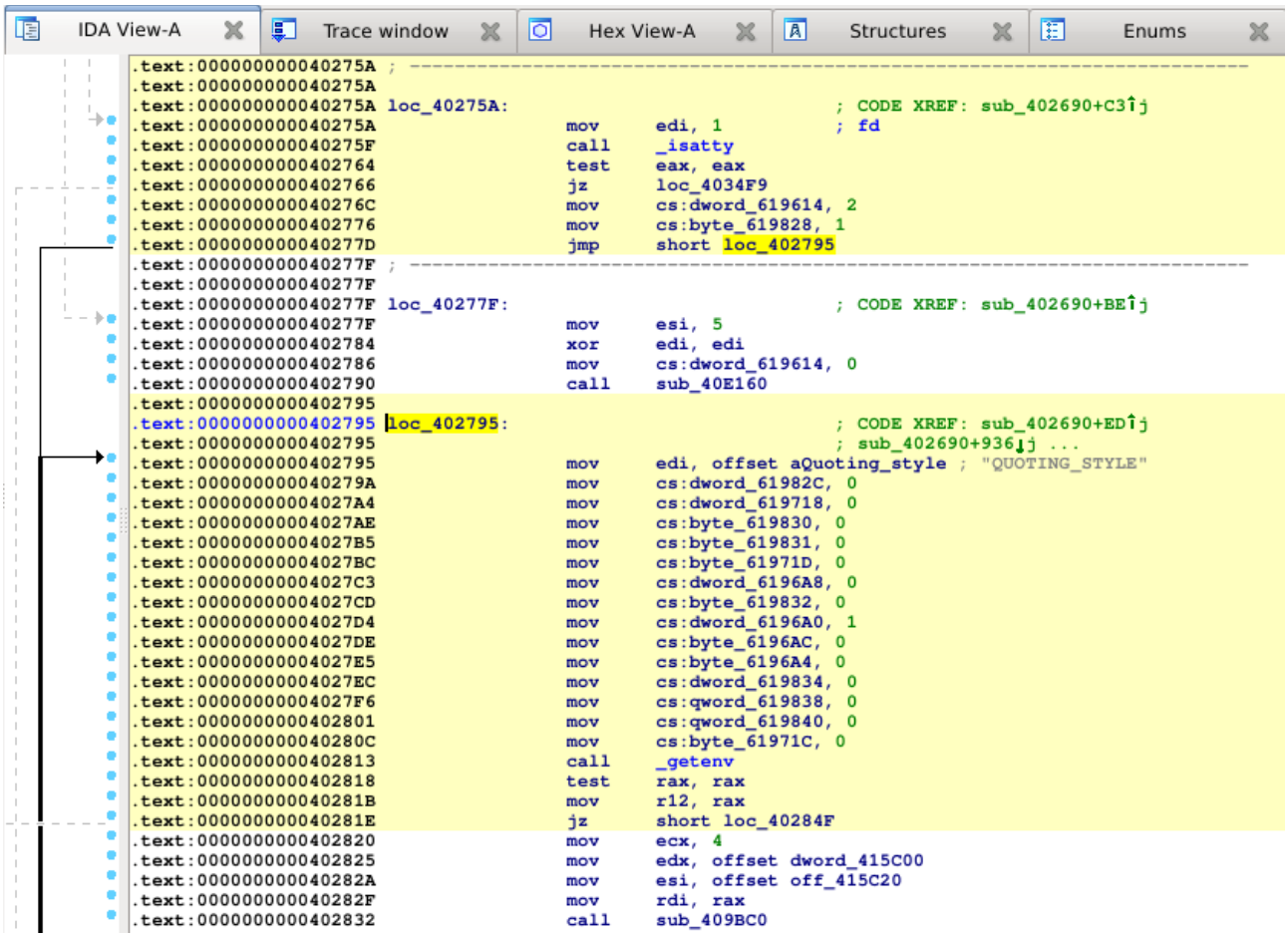
Once the breakpoint is set press F9 or select Debugger → Start process. The process will be suspended at the 1st instruction in function “start”:



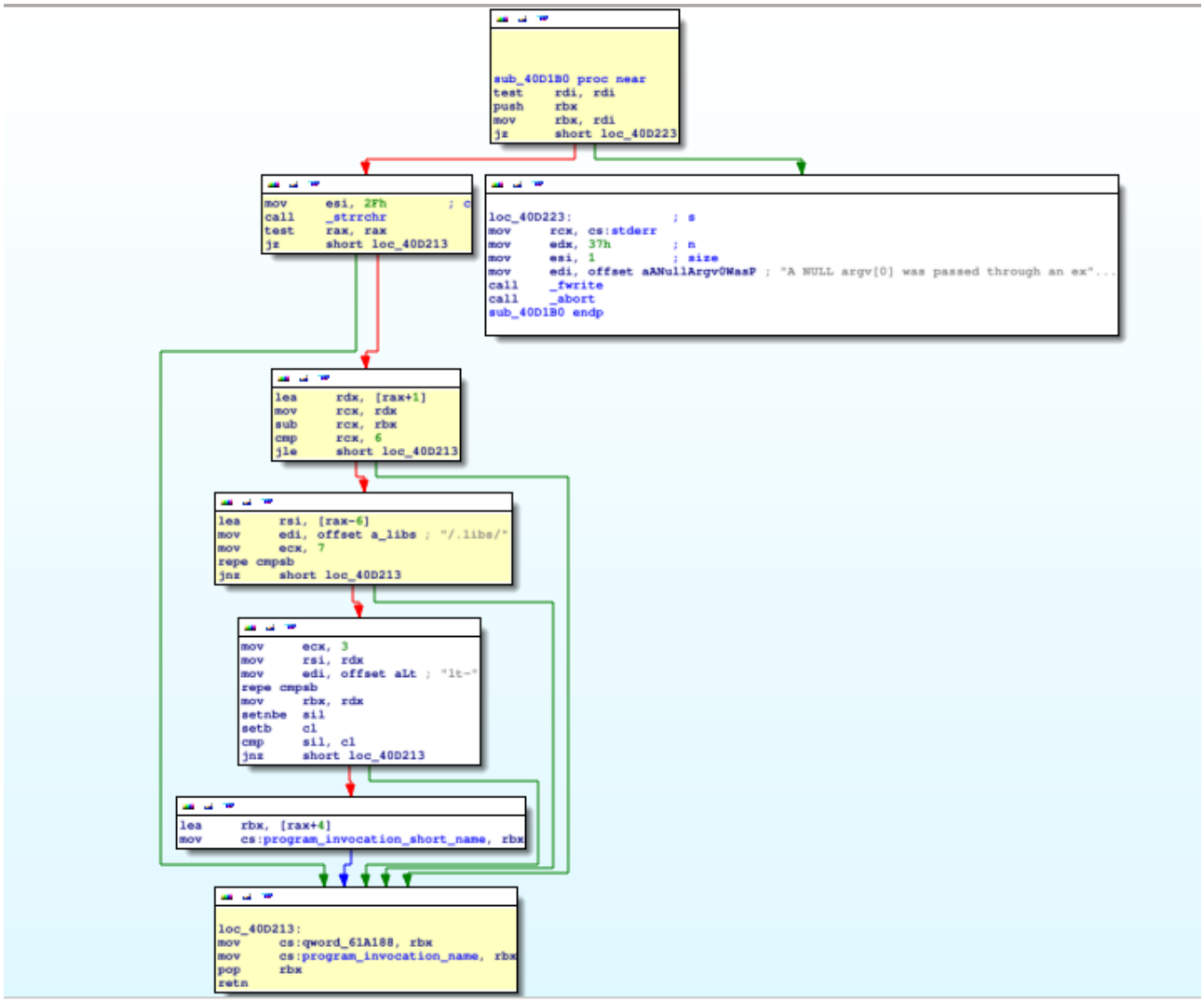
Now that we're debugging the process we can interact with the PIN tracer like with any other debugger module and step into or step over functions by pressing F7 or F8 alternatively. Simply let the application run and finish by pressing F9 again. After a while the process will terminate and IDA will display a dialog telling us that is reading the recorded trace. Once IDA reads the trace the debugger will stop and the instructions executed will be highlighted (like with the built-in tracing engine) as in the following picture:



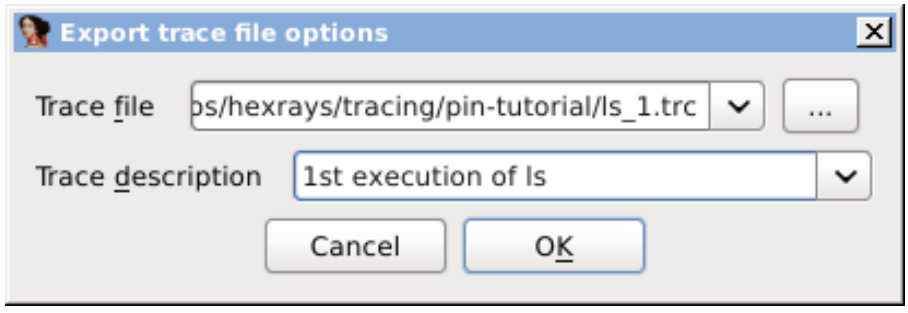
We can now inspect all the functions that were executed and see the exact basic blocks where execution passed:



Also, we can see in the graph view mode the complete path the application took in some specific function by switching to the graph view pressing space bar and then pressing “w” to zoom out:

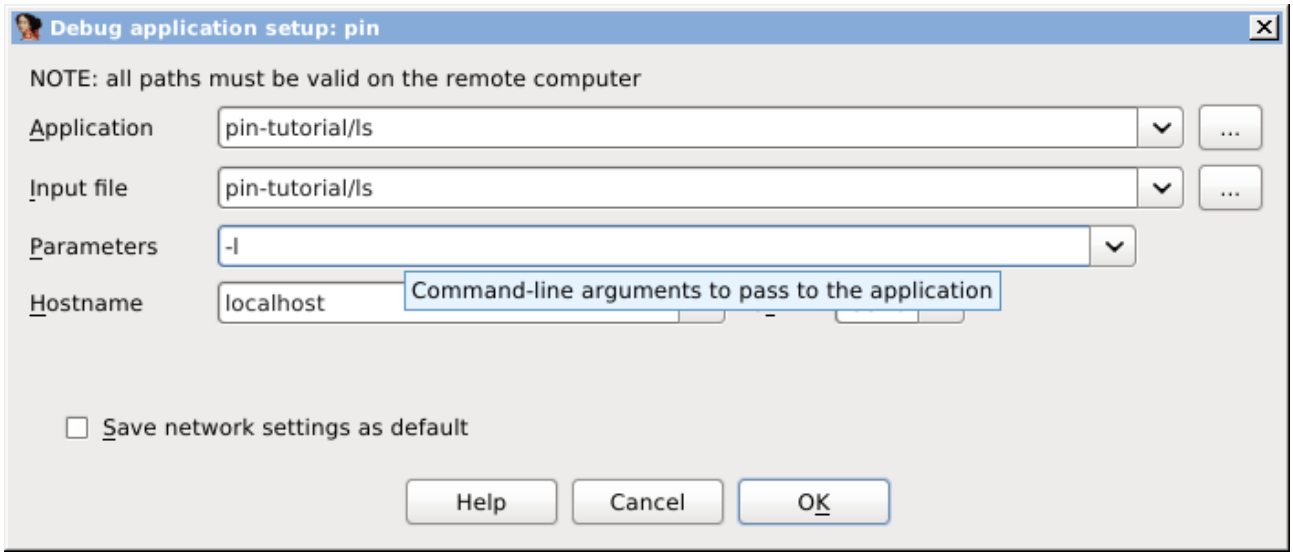


Let's save this trace by going to Debugger → Tracing →Trace window, then right click and select from the pop up menu "Save trace". Select the desired path to save the trace file, like in the next image:



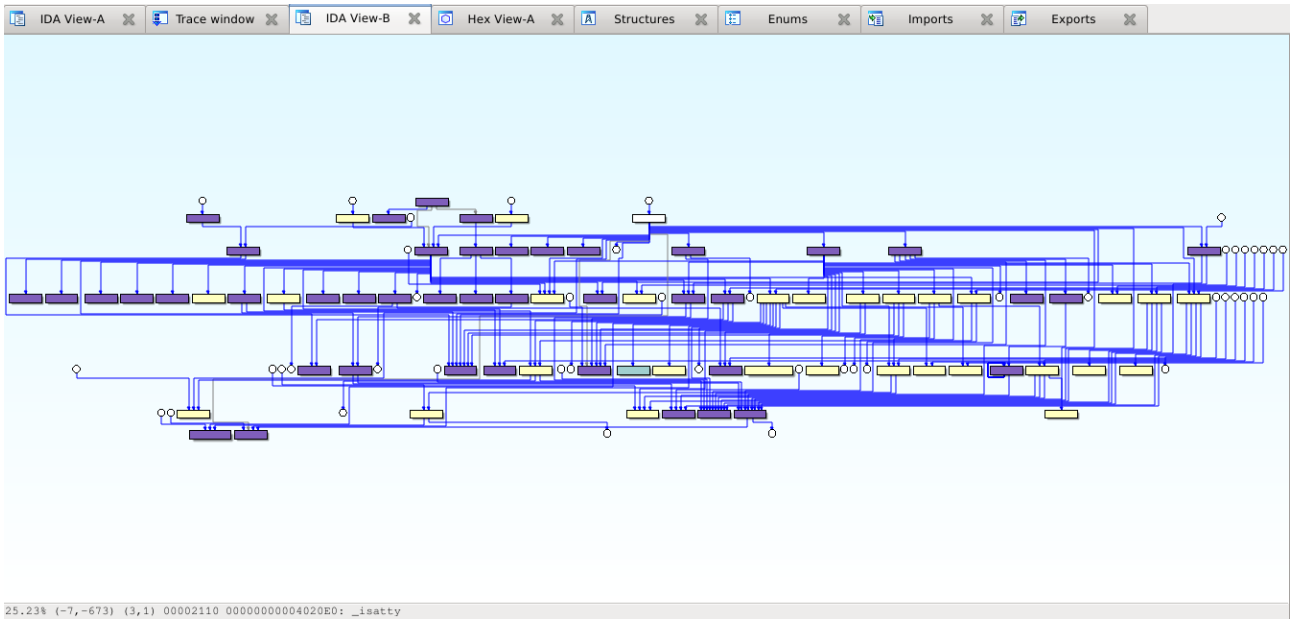
We have saved our trace. Let's clear the current trace via right click, "Clear Trace" in the "Trace Window", remove the previously added breakpoint and record another trace. This time we will add the command line argument "-l" to "ls". Select Debugger → Process options and add the argument

“-l” (without quotes) in the configuration dialog:



Press OK and execute the application again by pressing F9. The process will be executed, IDA will read the corresponding trace and stop debugging. Another trace have been recorded. To spot the differences between this execution and the previous one go to the “Trace Window”, right click, “Overlay” and select ”Load overlay”. The previously recorded trace will appear in the file selection dialog; press OK. Then go again to the “Overlay” submenu and select “Subtract overlay”.

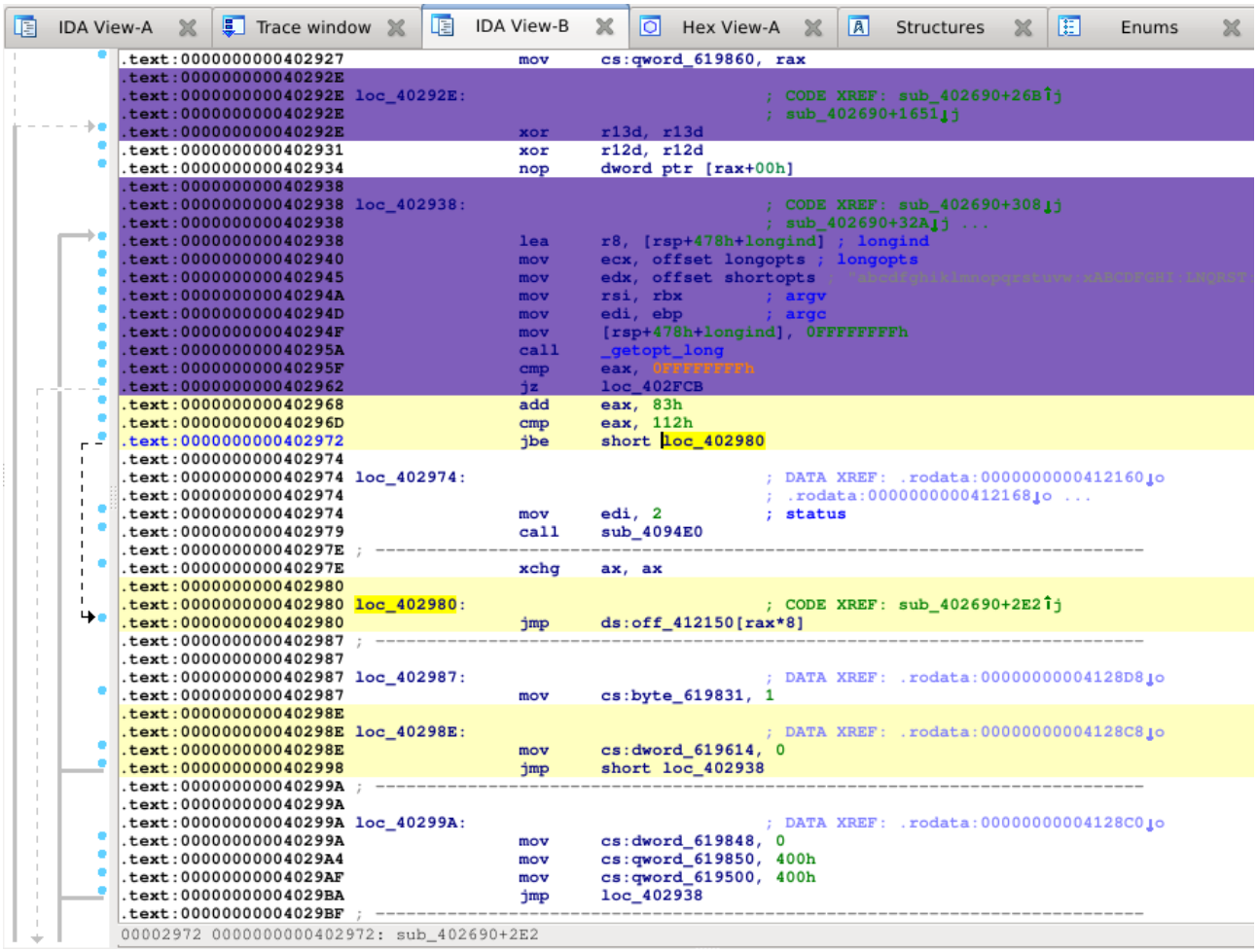
We can see the differences between both executions in a call graph to quickly discover which new functions were executed: right click in the “Trace Window” and select “Show trace call graph”. A graph similar to the next one will be displayed:



In this picture, the nodes highlighted in “yellow” are the new functions in the last trace. The other

color displayed in this graph is used to determine which functions are shared by both executions. In case there was any function executed in the 1st trace but not in the second, it would appear in pink.

We can also see the differences at assembly level between both executions as shown bellow:

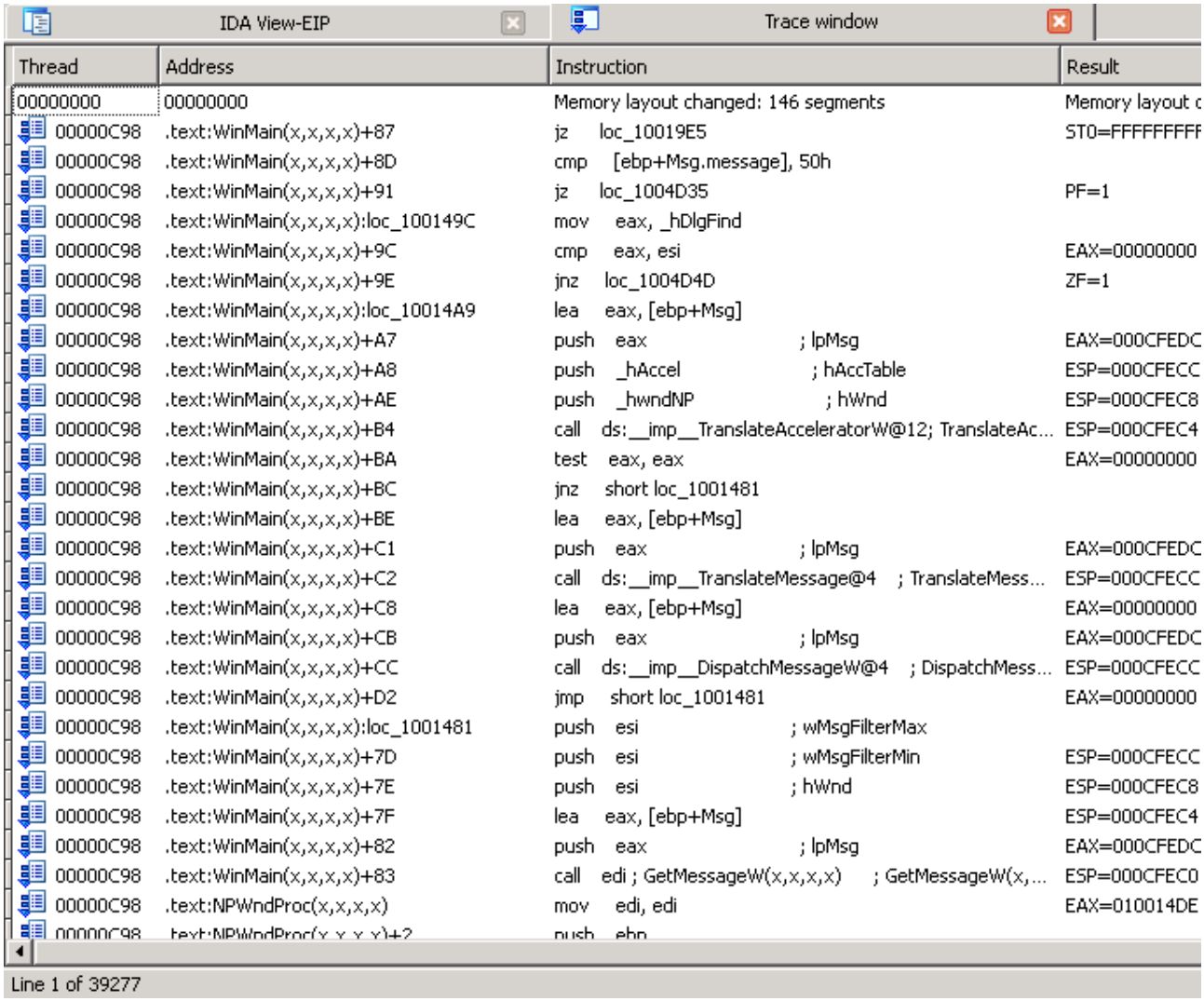


Differential debugging

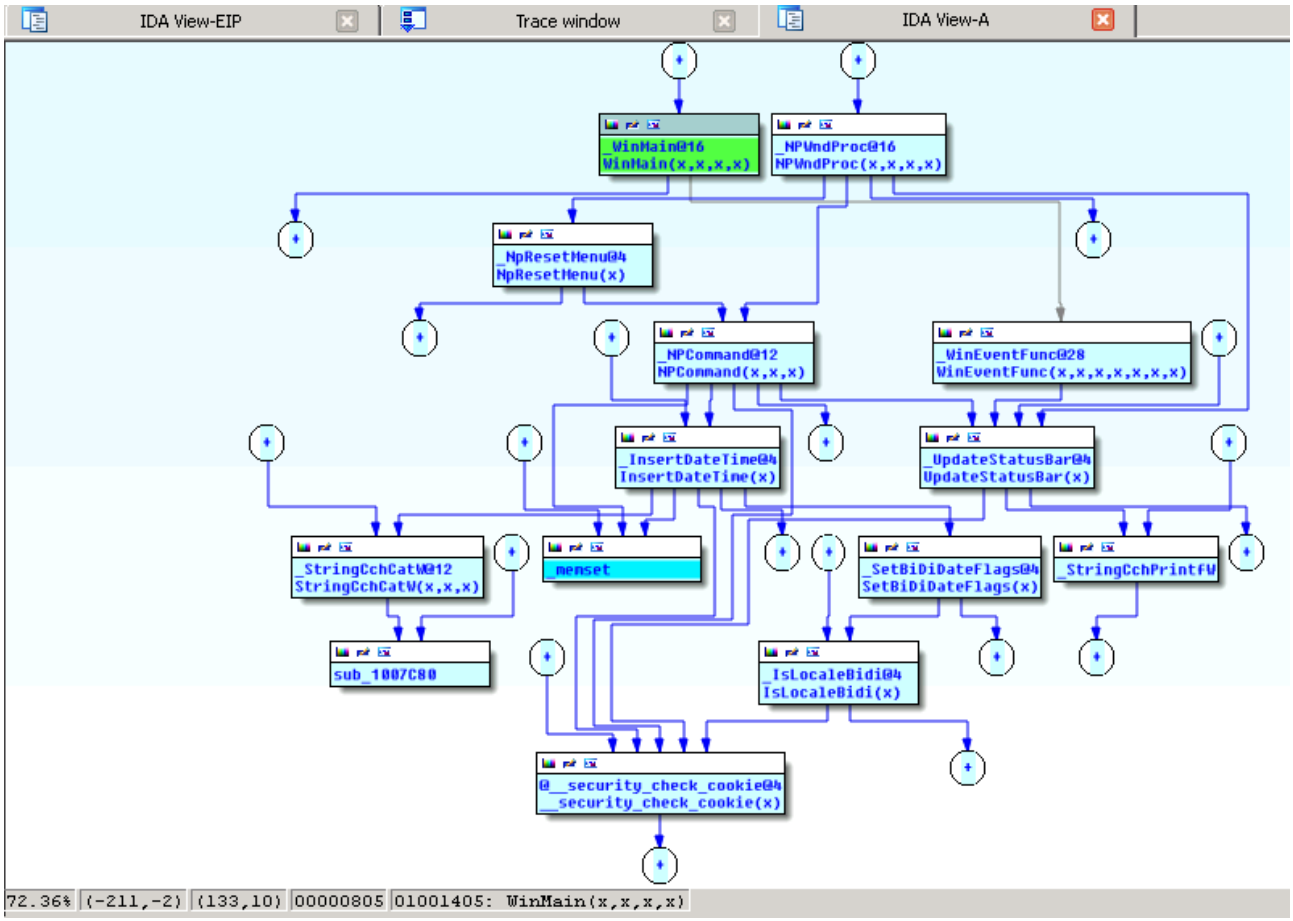
Differential debugging is the ability for a tool to discover new functionality in a target application recording traces and discarding every non new function, basic block or instruction (functions, basic blocks or instructions that were previously recorded in other traces).

Let's see how the PIN tracer can be used for differential debugging. In this case we will use the Microsoft Windows notepad.exe executable (x86). As with the previous one, open it in IDA and wait for the initial auto-analysis to finish. When done, select the PIN tracer module from the debuggers drop down list and configure its options as in the previous example with the only difference that we're going to use “Instruction tracing” instead of “Basic block tracing”. Once PIN is configured we need to set one more option: go to Debugger → Tracing options and check the menu item “Only add new instructions”. Then, press F9 or select Debugger → Start process.

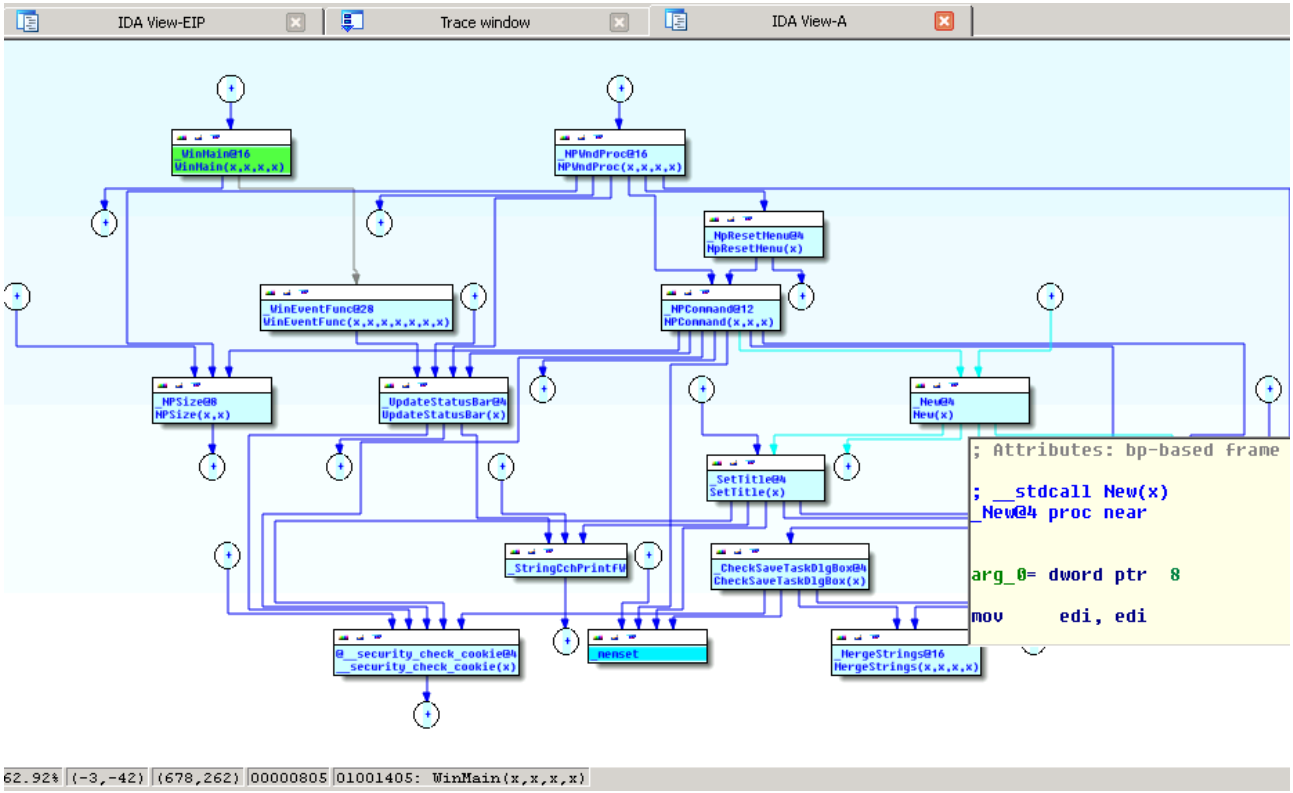
When the notepad window opens move, minimize and restore it, change its size, open the menu bar without selecting any menu items and, also, write some text. When done, return back to IDA and press the pause button. What we did is just to be sure that we recorded as many GUI paint events as possible so those new events doesn't appear in new traces. Clear this trace as we aren't interested in GUI events by selecting Debugger → Debugging options → Clear trace and press F9 again. Return back to notepad and select Edit → Time/Date. After this, switch to IDA, press the pause button and go to the “Trace Window”: only the new instructions and functions will be displayed, as in the following picture:



Like we did in the previous example, we can display the call graph of this specific trace via right click, “Other options” and then selecting “Show trace call graph”:



We can see the function name _InsertDateTime@4 at the center of the graph as well as all the other functions that were executed when one selects Edit → Date/Time in notepad. Let's discover now what code is executed when the menu File → New is selected in notepad. As before, clear the trace and press F9 to let notepad continue. Then, select File → New, answer “No” when asked to save the current document. After this, go back to IDA, press the pause button and check the trace window: only the new instructions responsible of executing the code to open a new document in notepad will be displayed. We can see those functions in a call graph as we did in almost all the previous examples:


IDA View-EIP
Trace window
IDA View-A
_WinMain@16
WinMain(x,x,x,x)
_WPWndProc@16
WPWndProc(x,x,x,x)
_WpResetMenu@4
WpResetMenu(x)
_WinEventFunc@28
WinEventFunc(x,x,x,x,x,x,x)
_WPCommand@12
WPCommand(x,x)
_WPSize@8
WPSize(x,x)
_UpdateStatusBar@4
UpdateStatusBar(x)
_New@4
New(x)
_SetTitle@4
SetTitle(x)
_StringCchPrintfW
_CheckSaveTaskDlgBox@4
CheckSaveTaskDlgBox(x)
@__security_check_cookie@4
__security_check_cookie(x)
_memset
_MergeStrings@16
MergeStrings(x,x,x,x)
; Attributes: bp-based frame
; __stdcall New(x)
_New@4 proc near
arg_0= dword ptr 8
mov edi, edi
62.92%
(-3,-42)
(678,262)
00000805
01001405: WinMain(x,x,x,x)

Performance considerations

Although we did our best in making the PIN debugger plugin as fast as possible it isn't as fast as running the application without any control. On the other hand, the PIN tracer is considerably faster compared to the built-in tracing engine. In any case, there are some options that can be configured to make the tracing experience faster, as explained in the next paragraphs.

Instruction vs Basic block vs Function tracing

While instruction level tracing is the tracing method which gives more information about what the target application did it's inevitably the slower method. Basic block level is faster than instruction tracing as not all instructions executed are being recorded, only some instructions of every basic block as seen by PIN, not by IDA. This method is the default tracing option for the PIN tracer.

Often, it isn't needed to trace neither instructions nor basic blocks as it's unknown which functions are those interesting. In this case, function tracing is the best option we can select and is the faster tracing option.

Tracing Basic blocks

When tracing basic blocks, on the PIN tool side, only some instructions are being recorded, not all instructions in the basic block. The instructions recorded are the following:

1. The 1st and last instructions of every basic block.
2. Procedure calls and returns from procedures.
3. All branches.
4. Instructions that may cause control flow to change because of exceptions (like UD2).

Those instructions are recorded on the PIN tool side and send through the network to IDA. Then, IDA highlights all of those instructions and, also, all the previous instructions not recorded in the same basic block. IDA does this task because of some differences between basic blocks in IDA and in PIN. This operation is expensive and makes IDA to take longer to display a recorded trace. We can make IDA faster by specifying that we don't want to add those extra instructions by unchecking the option “Log basic block instructions” in Debugger → Tracing options.

Reading traces

Traces recorded in PIN side are being read in IDA after one of the following events happens:

1. Execution is paused because of a breakpoint or because the pause button was pressed.
2. The trace is full.

The trace, in the PIN side, is considered full when a number of instructions is executed. This number is configurable in Debugger → Tracing options → Trace buffer size. For large applications, 1.000.000 of events is the recommended limit and the default. However, a bigger number of events may make the tracing engine to go faster, if there is enough memory to hold such a big trace, as

IDA doesn't need to pause the application, read the trace and resume it again.

Summary

And that's all! We hope you enjoy this new feature of IDA 6.4.