Copyright (c) 2021 Hex-Rays SA

Debugging Linux/Windows Applications with PIN Tracer module

Table of Contents

1. Introduction . . . 1
  1.1. PIN support for MacOSX . . . 1
2. Building the PIN tool . . . 1
  2.1. Building on Windows . . . 1
  2.2. Building on Linux . . . 2
3. Start process . . . 2
4. Attach to an existing process . . . 6
5. Remote debugging . . . 8
  5.1. Starting the remote PIN backend . . . 8
  5.2. Connecting IDA to the backend . . . 9

Last updated on March 31, 2021 — v0.2

1. 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.

1.1. PIN support for MacOSX

Recording traces on MacOSX target is not supported yet.

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

2. 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 of all download PIN from http://www.pintool.org , and unpack it on your hard drive.

IMPORTANT	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)**".

The building process of the PIN tool is different for Windows and Linux.

2.1. Building on Windows

1. Install Visual Studio. It is possible to build the PIN tool with the Express version of Visual Studio for C++.
2. 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 built with PIN version 3.0 and higher, for earlier versions of pintool you should use PIN build 65163.
3. Unpack the .zip file into /path/to/pin/source/tools/
4. 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.

Alternatively you can use GNU make:

1. Install GNU make as a part of cygwin or MinGW package

2. Unpack the .zip file into /path/to/pin/source/tools/
3. Prepare Visual Studio environment (e.g. %VCINSTALLDIR%\Auxiliary\Build\vcvars32.bat for 32-bit pintool or %VCINSTALLDIR%\Auxiliary\Build\vcvars64.bat for 64-bit one)
4. cd /path/to/pin/source/tools/idapin
5. make

2.2. Building on Linux

1. Install GCC 3.4 or later
2. Download the IDA pintool sources from: https://www.hex-rays.com/products/ida/support/freefiles/idapin$(IDAMAJMIN).zip (*)
3. Unpack the .zip file into /path/to/pin/source/tools/
4. Open a console, and do the following (only for versions of PIN prior to 3.0):
    a. cd /path/to/pin/ia32/runtime
    b. ln -s libelf.so.0.8.13 libelf.so
    c. cd /path/to/pin/intel64/runtime
    d. ln -s libelf.so.0.8.13 libelf.so
    e. cd /path/to/pin/source/tools/Utils
    f. ls testGccVersion 2>/dev/null || ln -s ../testGccVersion testGccVersion
5. cd /path/to/pin/source/tools/idapin

```
$ make TARGET=ia32
```

for building the x86 version, or

```
$ make TARGET=intel64
```

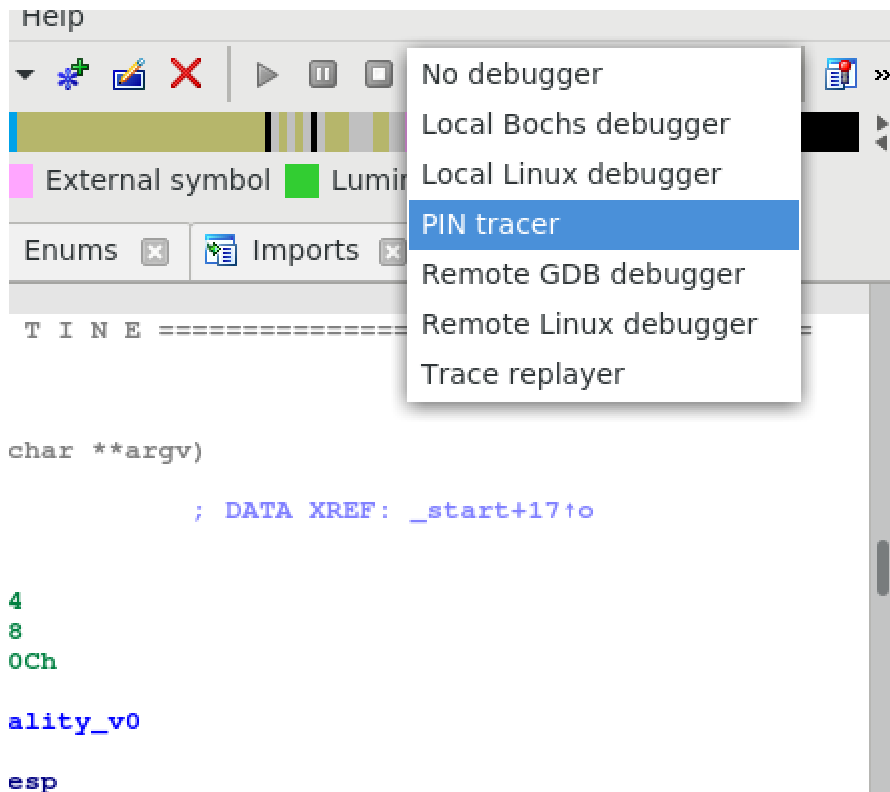
for the x64 version.

(*) Where '$(IDAMAJMIN)' is the IDA version major/minor. E.g., for IDA 7.6, the final URL would be: https://www.hex-rays.com/products/ida/support/freefiles/idapin76.zip

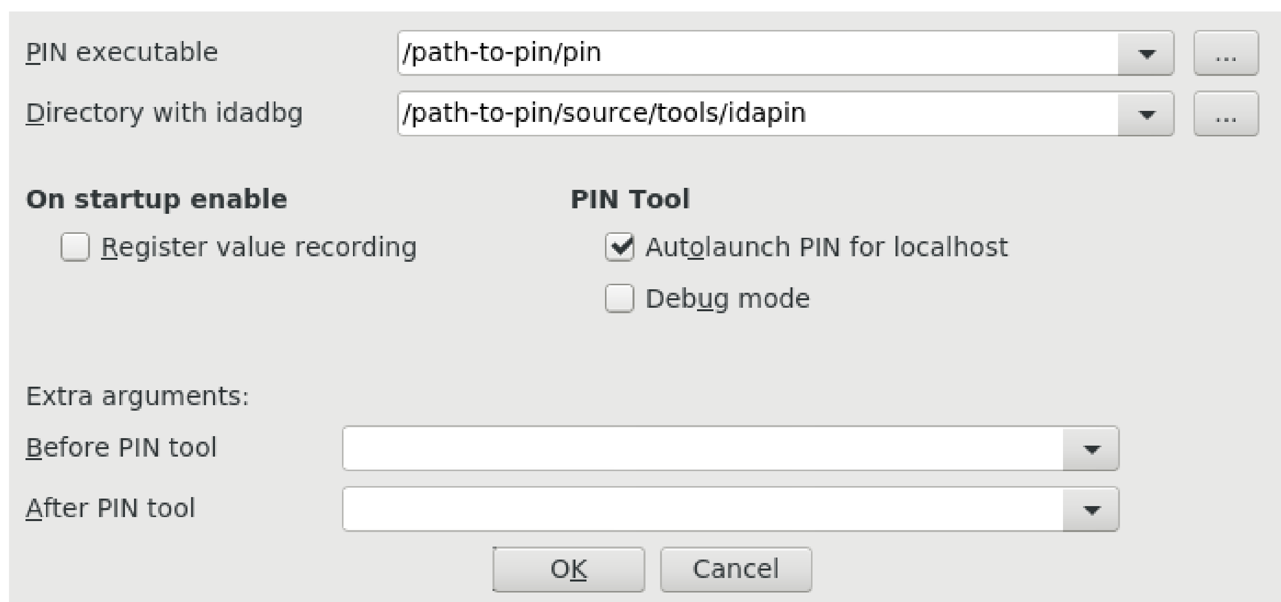
Pintool 6.9 and higher are compatible with versions 6.5-6.8 of IDA so currently you can use them.

3. Start process

Once the PIN tool module is built we can use it in IDA. Open a binary 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" file name). In some versions "pin.sh" may exist – in this case you should use it.
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 press OK in this dialog and enable option **Autolaunch PIN for localhost**.

We can interact with the PIN tracer like with any other debugger module: add breakpoints and step into or step over functions by pressing F7 or F8 alternatively.

Now we put a breakpoint in the very first instruction of function **main**

```
.text:08048492
.text:08048492 ; =============== S U B R O U T I N E =======================================
.text:08048492
.text:08048492 ; Attributes: bp-based frame
.text:08048492
.text:08048492 ; int __cdecl main(int argc, char **argv)
.text:08048492                 public main
.text:08048492 main            proc near               ; DATA XREF: _start+17↑o
.text:08048492
.text:08048492 char_in_main    = byte ptr -5
.text:08048492 foo             = dword ptr -4
.text:08048492 argc            = dword ptr  8
.text:08048492 argv            = dword ptr  0Ch
.text:08048492
.text:08048492 ; __unwind { // ___gxx_personality_v0
.text:08048492                 push    ebp
.text:08048493                 mov     ebp, esp
.text:08048495                 sub     esp, 18h
.text:08048498                 mov     [ebp+char_in_main], 62h ; 'b'
.text:0804849C                 lea     eax, [ebp+char_in_main]
00000492 08048492: main (Synchronized with Hex View-1)
```

and launch the debugger by pressing the F9 key or by clicking the **Start** button in the debugger toolbar.

```
.text:08048492
.text:08048492 ; =============== S U B R O U T I N E =============================
.text:08048492
.text:08048492 ; Attributes: bp-based frame
.text:08048492
.text:08048492 ; int __cdecl main(int argc, char **argv)
.text:08048492 public main
.text:08048492 main proc near                              ; DATA XREF: _start+17↑o
.text:08048492
.text:08048492 char_in_main= byte ptr -5
.text:08048492 foo= dword ptr -4
.text:08048492 argc= dword ptr  8
.text:08048492 argv= dword ptr  0Ch
.text:08048492
.text:08048492 ; __unwind { // ___gxx_personality_v0
.text:08048492 push    ebp
.text:08048493 mov     ebp, esp
.text:08048495 sub     esp, 18h
.text:08048498 mov     [ebp+char_in_main], 62h ; 'b'
.text:0804849C lea     eax, [ebp+char_in_main]
.text:0804849F mov     [esp+4], eax                        ; in_char_ptr
.text:080484A3 mov     dword ptr [esp], 7Ah ; 'z'          ; in_char
.text:080484AA call    _Z10return_intcPc                   ; return_int(char,char *)
.text:080484AF mov     [ebp+foo], eax
.text:080484B2 mov     eax, [ebp+foo]
.text:080484B5 leave
.text:080484B6 retn
.text:080484B6 ; } // starts at 8048492
00000492 08048492: main (Synchronized with EIP)
```

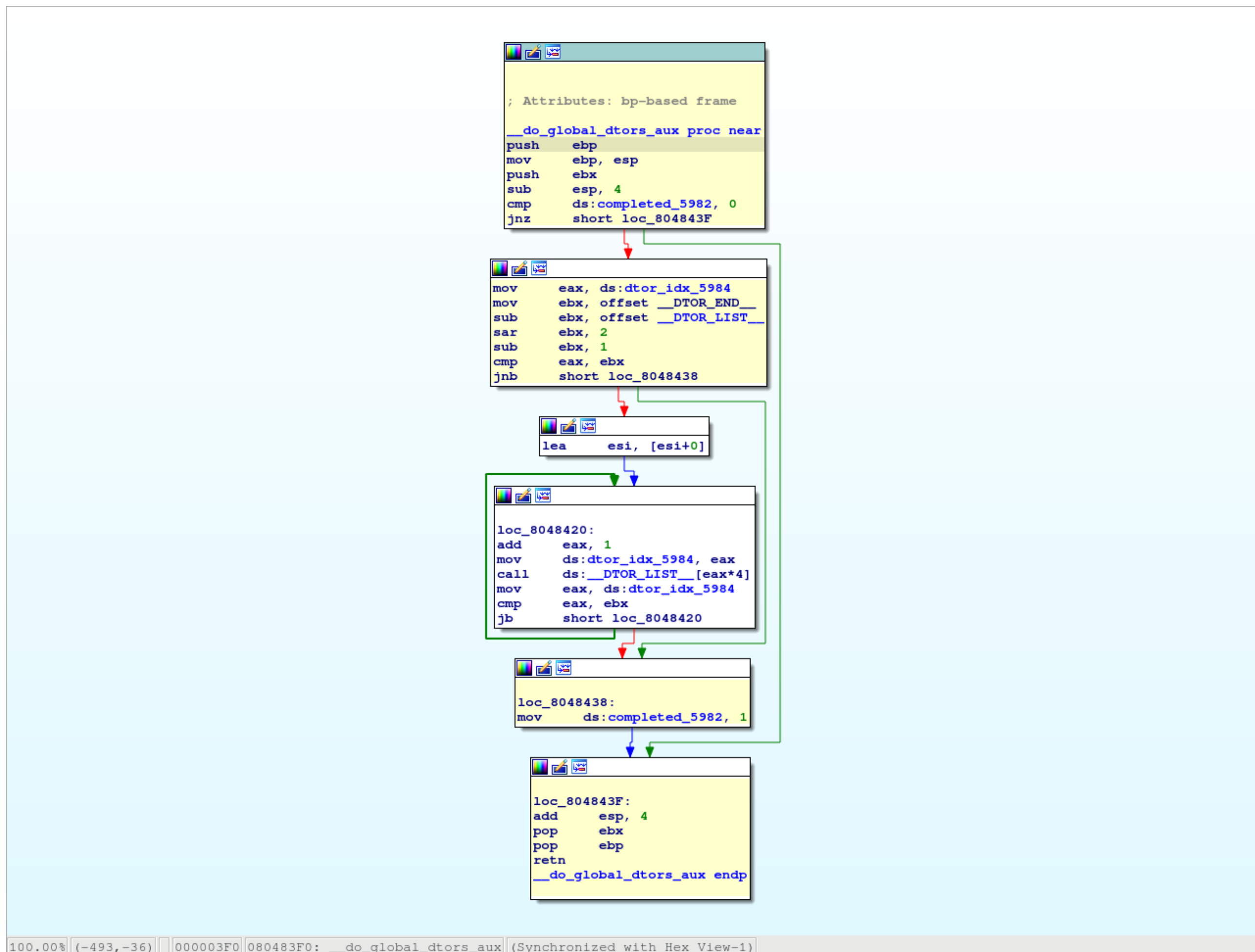
Make several steps by pressing F8. We can see all the instructions that were executed changed their color:

```
.text:08048492 argc= dword ptr  8
.text:08048492 argv= dword ptr  0Ch
.text:08048492
.text:08048492 ; __unwind { // ___gxx_personality_v0
.text:08048492 push    ebp
.text:08048493 mov     ebp, esp
.text:08048495 sub     esp, 18h
.text:08048498 mov     [ebp+char_in_main], 62h ; 'b'
.text:0804849C lea     eax, [ebp+char_in_main]
.text:0804849F mov     [esp+4], eax                ; in_char_ptr
.text:080484A3 mov     dword ptr [esp], 7Ah ; 'z'  ; in_char
.text:080484AA call    _Z10return_intcPc           ; return_int(char,char *)
.text:080484AF mov     [ebp+foo], eax
.text:080484B2 mov     eax, [ebp+foo]
.text:080484B5 leave
.text:080484B6 retn
.text:080484B6 ; } // starts at 8048492
.text:080484B6 main endp
.text:080484B6
.text:080484B6 ; ---------------------------------------------------------------
.text:080484B7 align 10h
.text:080484C0
.text:080484C0 ; =============== S U B R O U T I N E ==========================
.text:080484C0
.text:080484C0 ; Attributes: bp-based frame
.text:080484C0
.text:080484C0 public __libc_csu_fini
.text:080484C0 __libc_csu_fini proc near              ; DATA XREF: _start+B↑o
00000492 08048492: main (Synchronized with EIP)
```

Now 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:

```
.text:080483F0
.text:080483F0 ; Attributes: bp-based frame
.text:080483F0
.text:080483F0 __do_global_dtors_aux proc near        ; CODE XREF: _term_proc+13↓p
.text:080483F0                 push    ebp
.text:080483F1                 mov     ebp, esp
.text:080483F3                 push    ebx
.text:080483F4                 sub     esp, 4
.text:080483F7                 cmp     ds:completed_5982, 0
.text:080483FE                 jnz     short loc_804843F
.text:08048400                 mov     eax, ds:dtor_idx_5984
.text:08048405                 mov     ebx, offset __DTOR_END__
.text:0804840A                 sub     ebx, offset __DTOR_LIST__
.text:08048410                 sar     ebx, 2
.text:08048413                 sub     ebx, 1
.text:08048416                 cmp     eax, ebx
.text:08048418                 jnb     short loc_8048438
.text:0804841A                 lea     esi, [esi+0]
.text:08048420
.text:08048420 loc_8048420:                           ; CODE XREF: __do_global_dtors_aux+46↓j
.text:08048420                 add     eax, 1
000003F0 080483F0: __do_global_dtors_aux (Synchronized with Hex View-1)
```

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:



4. Attach to an existing process

Instead of launching a new process we could attach to a running process and debug it. For that we could have selected the "**Debugger > Attach to process...**" menu item. IDA will display a list of active processes.

ID	Name
10332	[64] /usr/bin/scl enable devtoolset-6 -- bash --init-file /home/pa...
10333	[64] /bin/bash /var/tmp/scl84SCEu
10336	[64] bash --init-file /home/pavel/bin/devenv_ver
10564	[64] /usr/libexec/gvfsd-trash --spawner :1.23 /org/gtk/gvfs/exec...
105688	[64] -bash
10576	[64] /usr/libexec/gvfsd-network --spawner :1.23 /org/gtk/gvfs/e...
105792	[64] vim pavel-slides.txt
105810	[64] vim block.cpp
10597	[64] /usr/libexec/gvfsd-dnssd --spawner :1.23 /org/gtk/gvfs/exe...
10735	[64] /usr/bin/scl enable devtoolset-6 -- bash --init-file /home/pa...
10736	[64] /bin/bash /var/tmp/scl7YAwAU
10739	[64] bash --init-file /home/pavel/bin/devenv_ver
107830	[64] -bash
108023	[64] -bash
110281	[64] -bash
110356	[64] /usr/bin/scl enable devtoolset-6 -- bash --init-file /home/pa...
110357	[64] /bin/bash /var/tmp/sclYfN77N
110360	[64] bash --init-file /home/pavel/bin/devenv_ver
110495	[64] vim allmake.mak
110588	[64] -bash
110663	[64] /usr/bin/scl enable devtoolset-6 -- bash --init-file /home/pa...
110664	[64] /bin/bash /var/tmp/sclRPvZe3
110667	[64] bash --init-file /home/pavel/bin/devenv_ver
111220	[64] '/opt/google/chrome/chrome --type=renderer --field-trial-h...

Line 1 of 185

OK | Cancel | Search | Help

We just select the process we want to attach to. IDA will then attach to the selected process, and leave it suspended at the place it was when it was attached to:

Copyright (c) 2021 Hex-Rays SA

5. Remote debugging

In case of remote debugging you can run IDA and PIN backend on different platforms.

5.1. Starting the remote PIN backend

The first thing to do, is to start the PIN debugging backend on the target machine. Command line depends of bitness of the target application.

```
$ <path-to-pin> -t <path-to-pintool> -p <port> -- <application> <application-options>
```

For example, a 64-bit application **ls** would be started for debugging by the following comand:

```
$ /usr/local/pin/pin \
  -t /usr/local/pin/source/tools/idapin/obj-intel64/idadbg64.so \
  -p 23947 -- \
  /bin/ls
```

whereas a 32-bit one **hello32** as follows:

```
/usr/local/pin/pin \
  -t /usr/local/pin/source/tools/idapin/obj-ia32/idadbg.so \
  -p 23947 -- \
  ./hello32
```

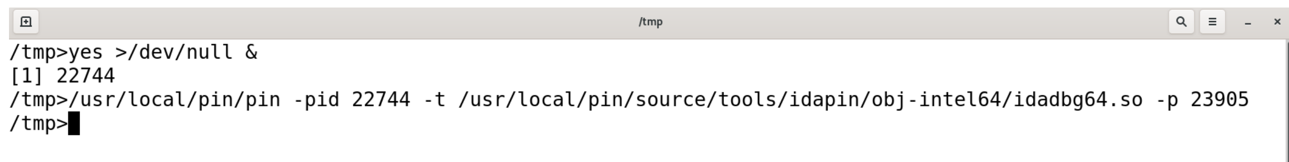
there is a more complicated way to start an application regardless bitness:

```
/usr/local/pin/pin \
  -t64 /usr/local/pin/source/tools/idapin/obj-intel64/idadbg64.so \
  -t /usr/local/pin/source/tools/idapin/obj-ia32/idadbg.so \
  -p 23947 -- \
  /usr/bin/ls
```

Also you can attach to already running programs:

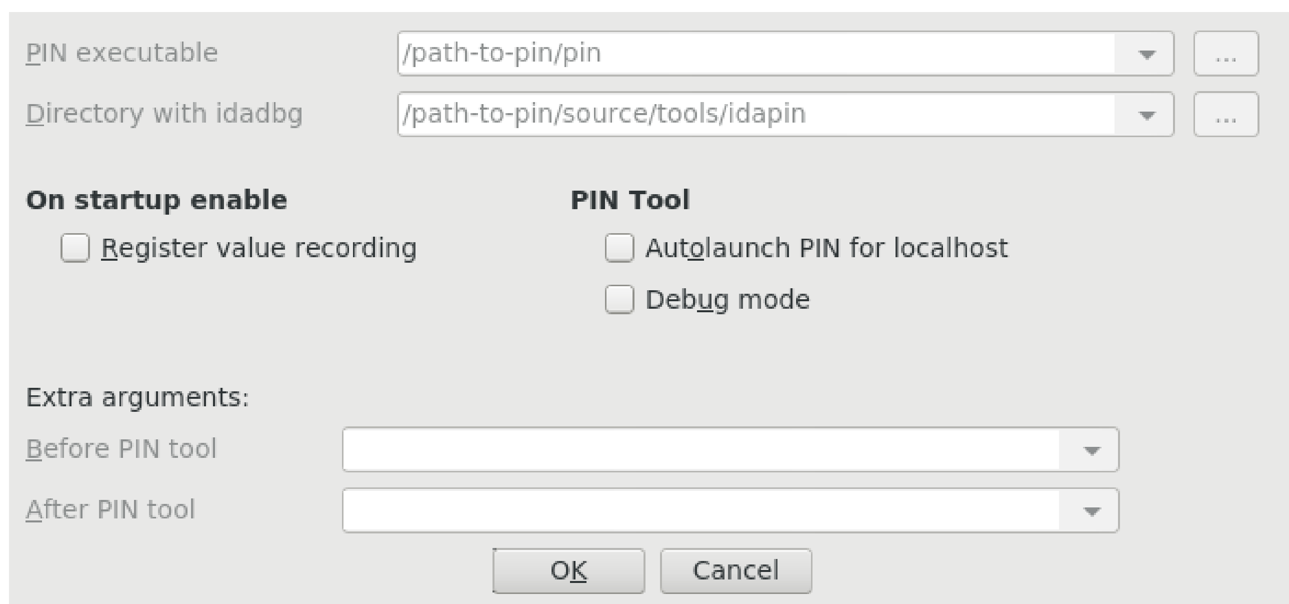
```
$ <path-to-pin> -pid <pid-to-attach> -t <path-to-pintool> -p <port> --
```

For example:

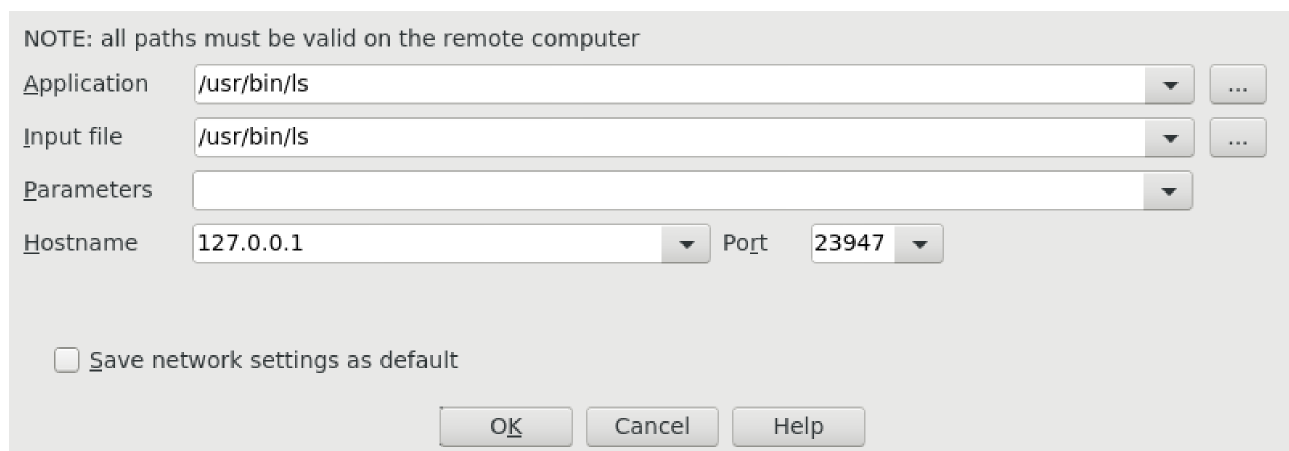


5.2. Connecting IDA to the backend

The next step is to select PIN tracer module in IDA via **Debugger > Select debugger** and switch IDA to remote PIN backend. For this you should disable option **Autolaunch PIN for localhost** in the PIN options dialod (**Debugger > Debugger options > Set specific options**):



and then tell IDA about the backend endpoint, through the menu action **Debugger > Process options…**



Copyright (c) 2021 Hex-Rays SA

Once IDA knows what host to contact (and on what port), debugging an application remotely behaves exactly the same way as if you were debugging it locally.