Debugging Windows Applications with IDA WinDbg Plugin

Copyright 2011 Hex-Rays SA

Quick overview:

The Windbg debugger plugin is an IDA Pro debugger plugin that uses Microsoft's debugging engine (dbgeng) that is used by Windbg, Cdb or Kd.

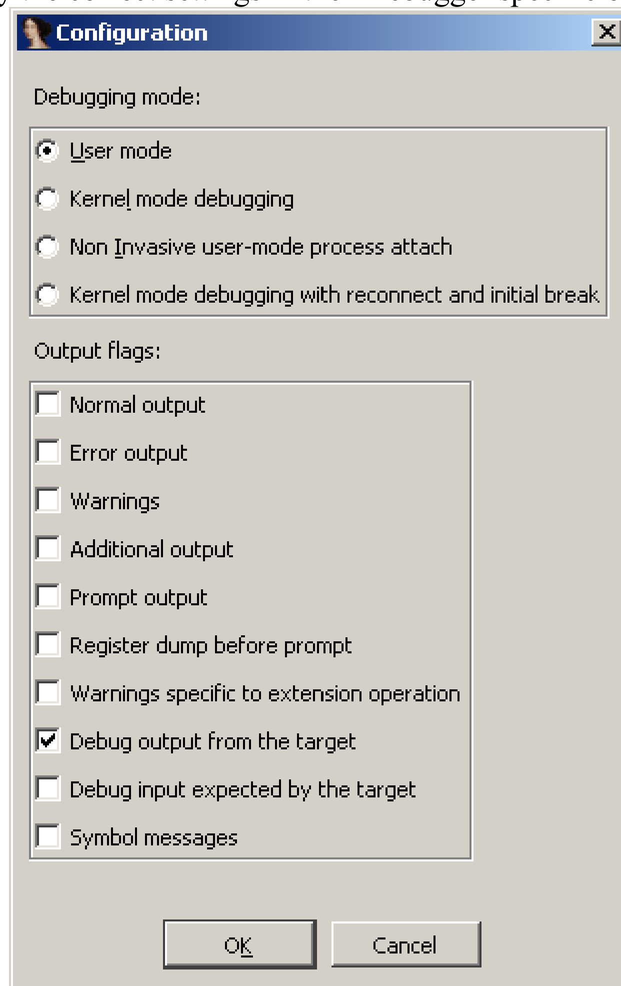
To get started, you need to install the latest Debugging Tools from Microsoft website:
https://msdn.microsoft.com/en-us/windows/hardware/hh852365

or from the Windows SDK / DDK package.

Please make sure you should install the **x86** version of the debugging tools which is used by both IDA Pro and IDA Pro 64. The x64 version will NOT work.

After installing the debugging tools, make sure you select « Debugger / Switch Debugger » and select the WinDbg debugger.

Also make sure you specify the correct settings in the “Debugger specific options” dialog:

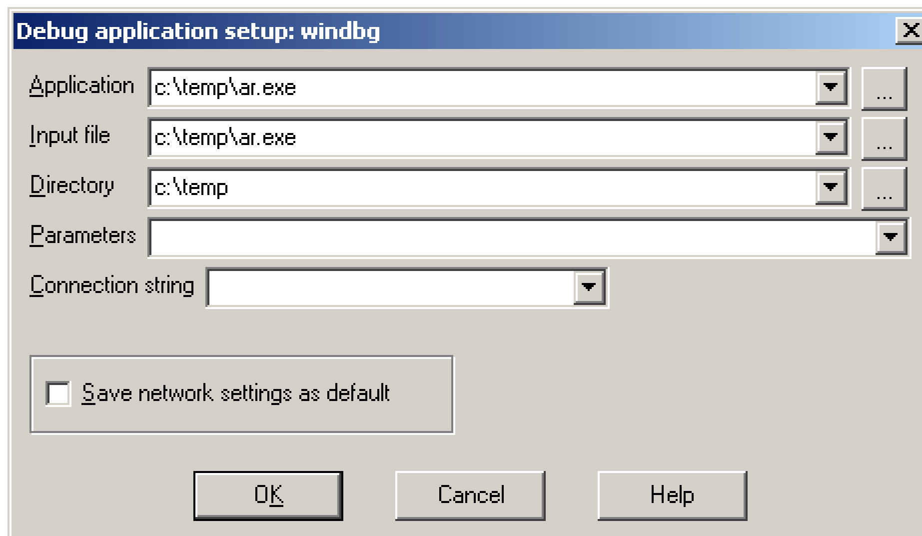


- **User mode:** Select this mode for user mode application debugging (default mode)
- **Kernel mode:** Select this mode to attach to a live kernel.
- **Non Invasive debugging:** Select this mode to attach to a process non-invasively
- **Output flags:** These flags tell the debugging engine which kind of output messages to display and which to omit
- **Kernel mode debugging with reconnect and initial break:** Select this option when debugging a kernel and when the connection string contains 'reconnect'. This option will assure that the debugger breaks as soon as possible after a reconnect.

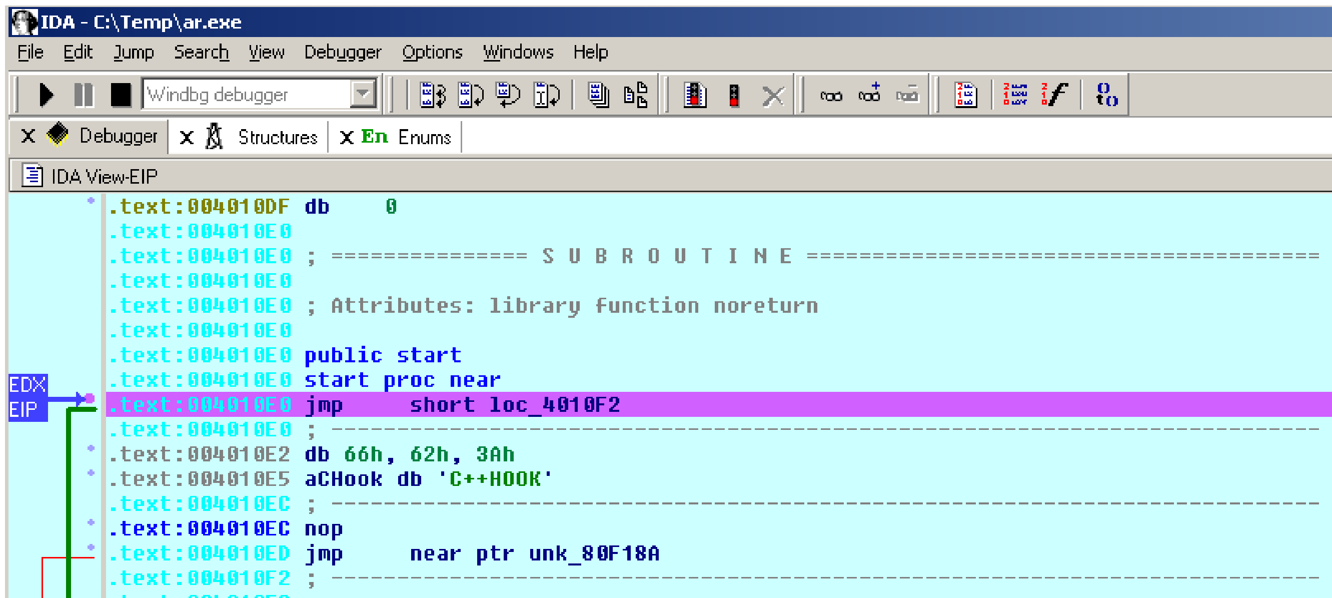
To make these settings permanent, please edit the IDA\cfg\dbg_windbg.cfg file.

To specify the debugging tools folder you may add to the PATH environment variable the location of Windbg.exe or edit %IDA%\cfg\ida.cfg and change the value of the DBGTOOLS key.

After the debugger is properly configured, edit the process options and leave the connection string value empty because we intend to debug a local user-mode application.

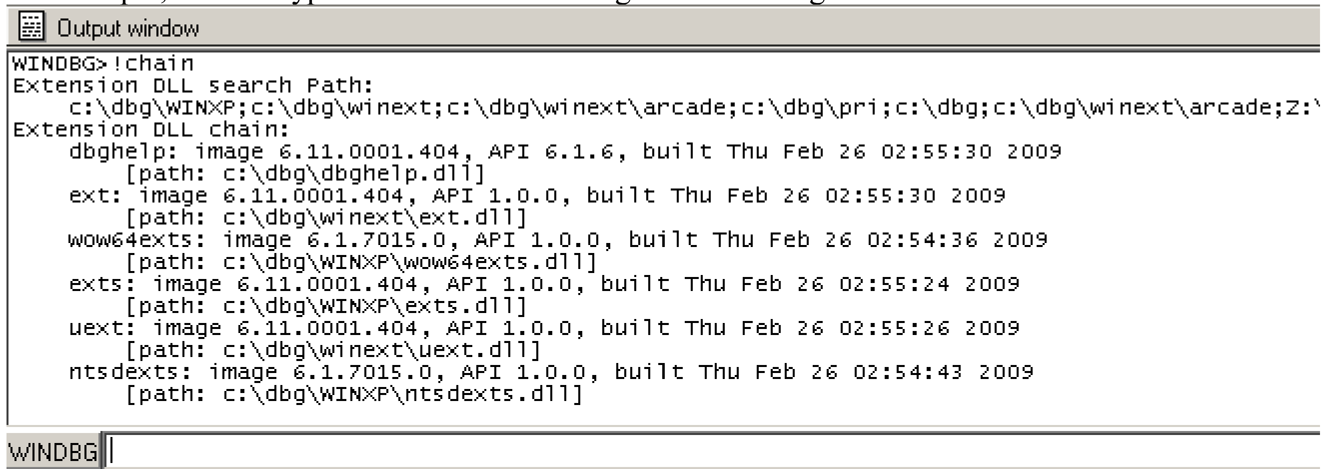


Now hit F9 to start debugging:

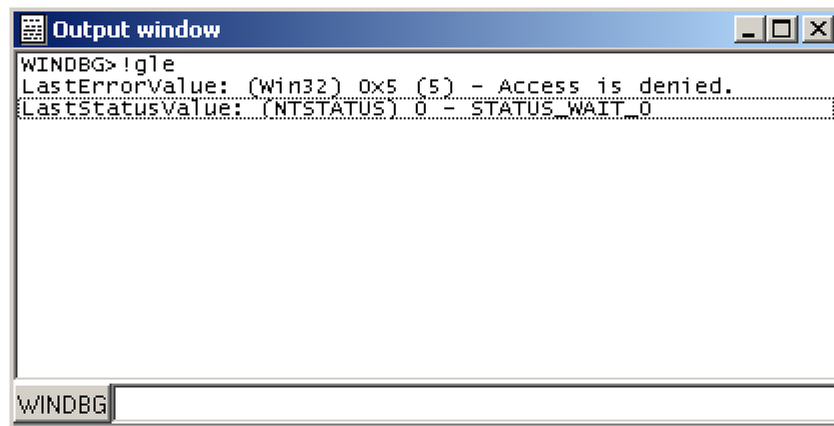


The Windbg plugin is very similar to IDA Pro's Win32 debugger plugin, nonetheless by using the former, one can benefit from the command line facilities and the extensions that ship with the debugging tools.

For example, one can type “!chain” to see the registered Windbg extensions:

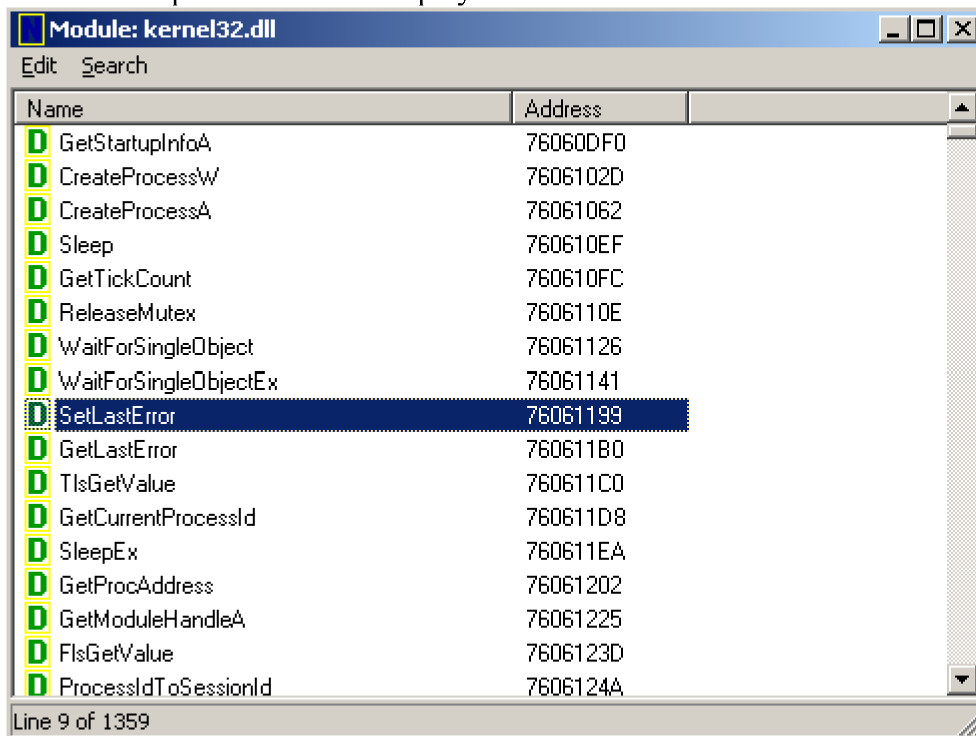


“!gle” is another command to get the last error value of a given Win32 API call.



Another benefit of using the Windbg debugger plugin is the use of symbolic information.

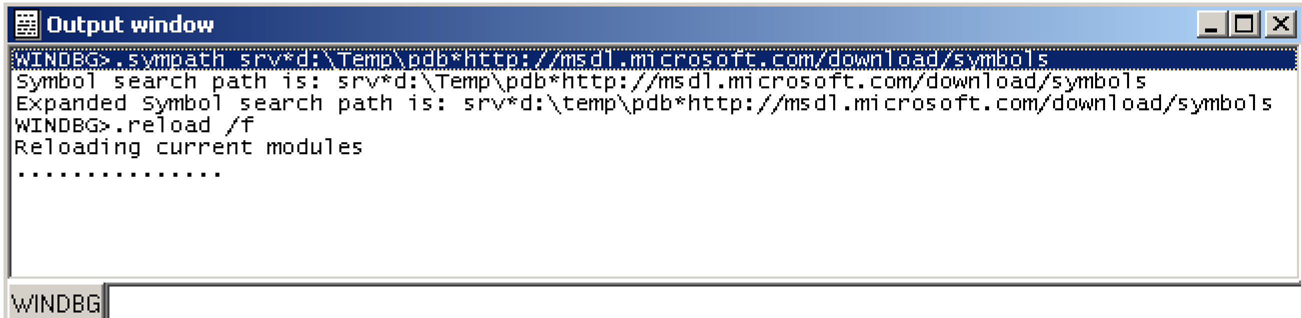
Normally, if the debugging symbols path is not set, then the module window will only show the exported names. For example kernel32.dll displays 1359 names:



Let us configure a symbol source by adding this environment variable before running IDA:

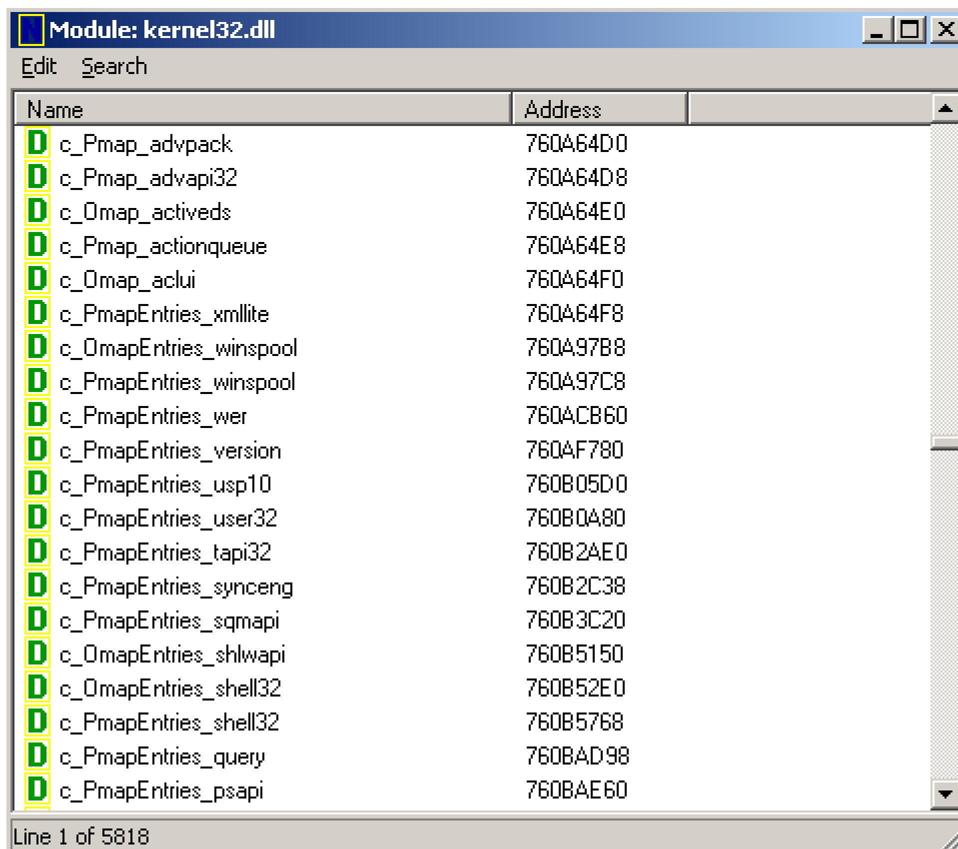
set _NT_SYMBOL_PATH=srv***C:\Temp\pdb***http://msdl.microsoft.com/download/symbols

It is also possible to set the symbol path directly while debugging:



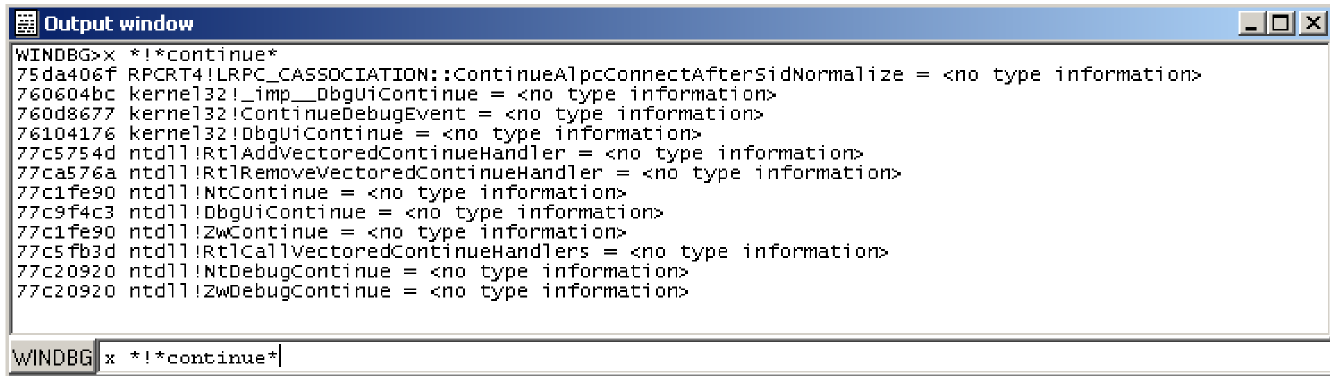
and then typing ".reload /f" to reload the symbols.

Now we try again and notice that more symbol names are retrieved from kernel32.dll:



Now we have 5818 symbols instead!

It is also possible to use the “x” command to quickly search for symbols:



(Looking for any symbol in any module that contains the word “continue”)

Debugging a remote process:

We have seen how to debug a local user mode program, now let us see how to debug a remote process. First let us assume that “pcA” is the target machine (where we will run the debugger server and the debugged program) and “pcB” is the machine where IDA Pro and the debugging tools are installed.

To start a remote process:

- On “pcA”, type:
  dbgsrv -t tcp:port=5000
  (change the port number as needed)
- On “pcB”, setup IDA Pro and Windbg debugger plugin:
  - “Application/Input file”: these should contain a path to the debuggee residing in “pcA”
  - Connection string: **tcp:port=5000,server=pcA**

Now run the program and debug it remotely.

To attach to a remote process, use the same steps to setup “pcA” and use the same connection string when attaching to the process.

More about connection strings and different protocols (other than TCP/IP) can be found in “debugger.chm” in the debugging tools folder.

Debugging the kernel with VMWare:

We will now demonstrate how to debug the kernel through a virtual machine.

In this example we will be using VMWare 6.5 and Windows XP SP3.

Configuring the virtual machine:

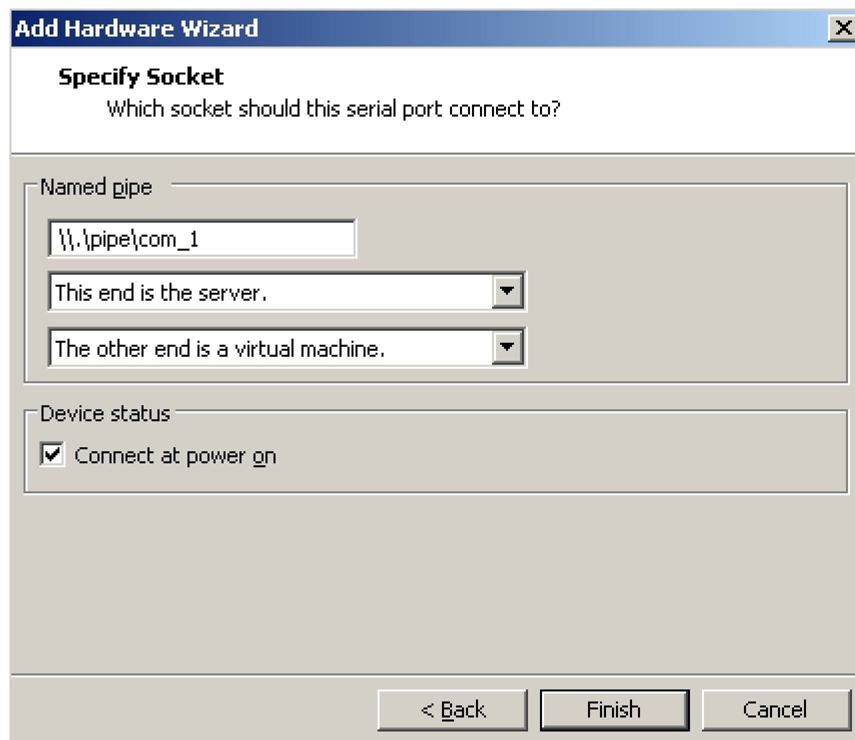
Run the VM and then edit "c:\boot.ini" file and add one more entry (see in bold):

```
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=optin /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Local debug" /noexecute=optin /fastdetect /debug /debugport=com1 /baudrate=115200
```

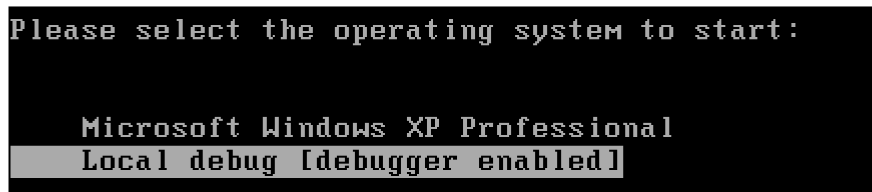
For MS Windows Vista please see: http://msdn.microsoft.com/en-us/library/ms791527.aspxp

Actually the last line is just a copy of the first line but we added the "/debug" switch and some configuration values.

Now shutdown the virtual machine and edit its hardware settings and add a new serial port with option "use named pipes":

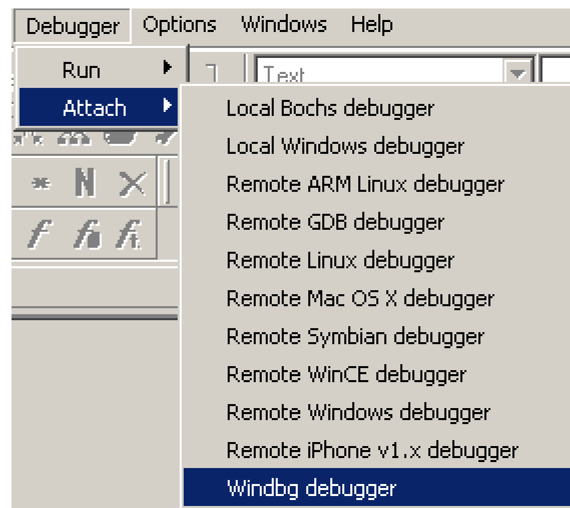


Press “Finish” and start the VM. At the boot prompt, select “Local debug” from the boot menu:



Configuring Windbg debugger plugin:

Now run IDA Pro and select Debugger / Attach / Windbg



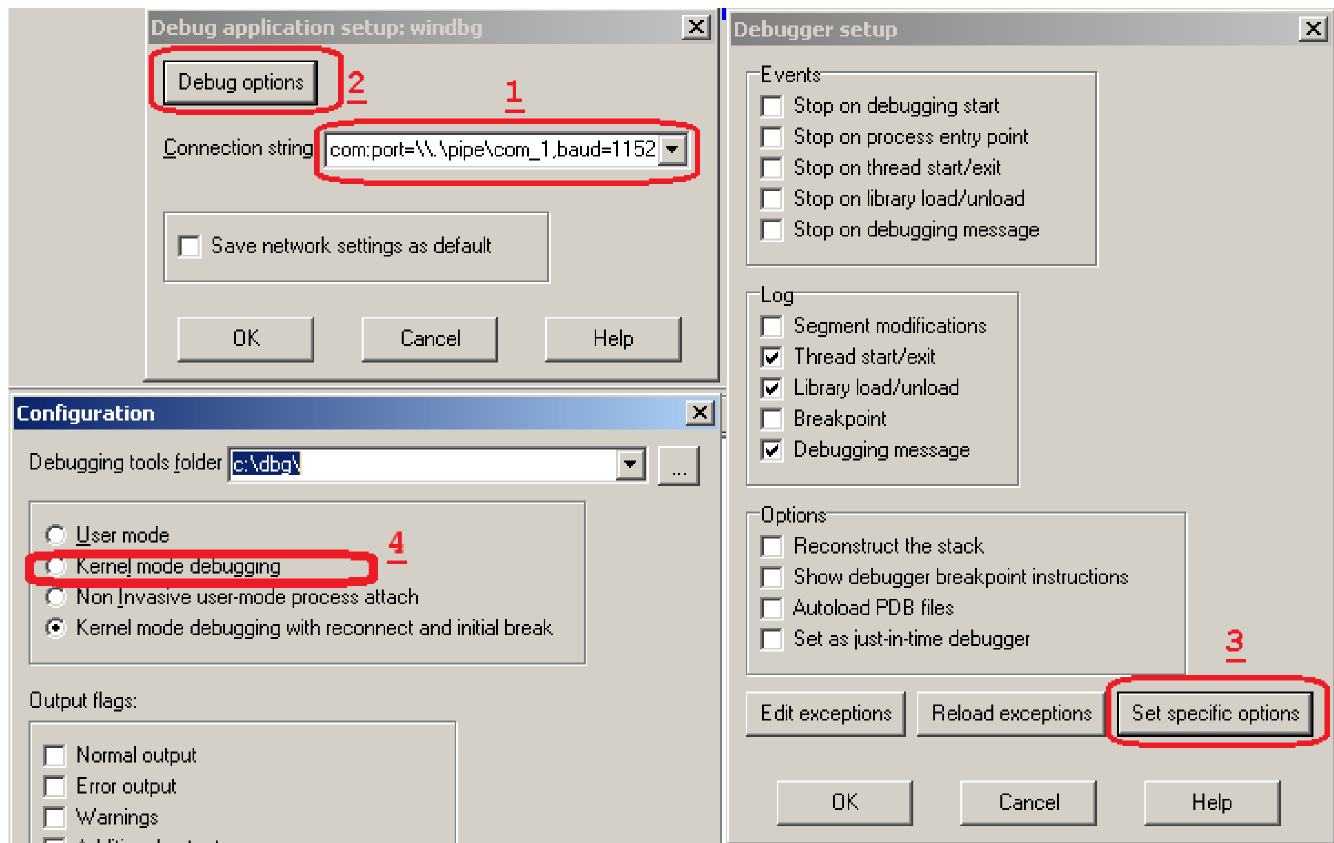
Then configure it to use “Kernel mode” debugging and use the following connection string:

com:port=\\.**\pipe\com_1**,baud=115200,pipe

It is possible to use the 'reconnect' keyword in the connection string:

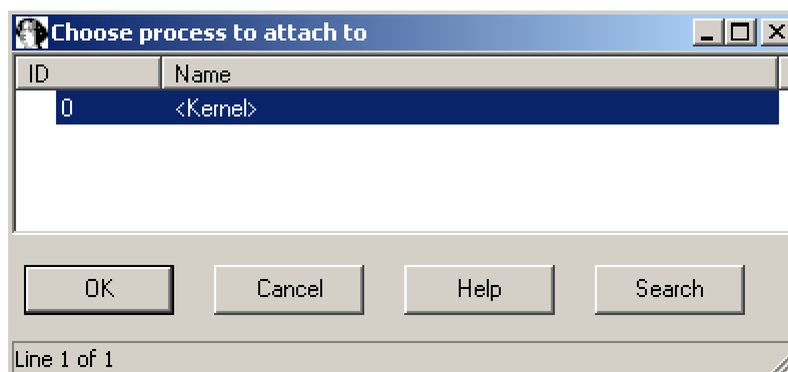
com:port=\\.**\pipe\com_1**,baud=115200,pipe,reconnect

Also make sure the appropriate option is selected from the debugger specific options.



Please note that the connection string (in step 1) refers to the named pipe we set up in the previous steps.

Finally, press OK to attach and start debugging.



When IDA attaches successfully, it will display something like this:

```
nt:8052A854
nt:8052A854 nt_RtlpBreakWithStatusInstruction:      ; Trap to Debugger
nt:8052A854                      int     3
nt:8052A855                      retn    4
nt:8052A855 ; ---------------------------------------------------------------
```

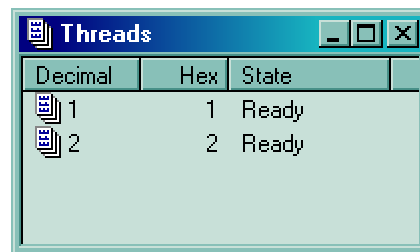
If you do not see named labels then try checking your debugging symbols settings.

Note: In kernel mode IDA Pro will display one entry in the threads window for each processor.

For example a two processor configuration yields:

Device	Summary
Memory	256 MB
Processors	2

VMWare configuration



Threads in IDA

This screenshot shows how we are debugging the kernel and changing the disassembly listing (renaming stack variables, or using structure offsets):

```
nt:804F1808 ; int __stdcall nt_IoDeleteDevice(int pDeviceObject)
nt:804F1808 nt_IoDeleteDevice proc near
nt:804F1808
nt:804F1808 pDeviceObject   = dword ptr  8
nt:804F1808
nt:804F1808                 mov     edi, edi
nt:804F180A                 push    ebp
nt:804F180B                 mov     ebp, esp
nt:804F180D                 cmp     nt_IopVerifierOn, 0
nt:804F1814                 push    esi
nt:804F1815                 mov     esi, [ebp+pDeviceObject]
nt:804F1818                 jz      short loc_804F1820
nt:804F181A                 push    esi
nt:804F181B                 call    near ptr nt_IovDeleteDevice
nt:804F1820
nt:804F1820 loc_804F1820:                           ; CODE XREF: nt_IoDeleteDevice+10↑j
nt:804F1820                 test    byte ptr [esi+(DEVICE_OBJECT.Flags+1)], 8
nt:804F1824                 jz      short loc_804F182C
nt:804F1826                 push    esi
nt:804F1827                 call    near ptr nt_IoUnregisterShutdownNotification
nt:804F182C
nt:804F182C loc_804F182C:                           ; CODE XREF: nt_IoDeleteDevice+1C↑j
nt:804F182C                 push    edi
nt:804F182D                 mov     edi, [esi+DEVICE_OBJECT.Timer]
nt:804F1830                 test    edi, edi
nt:804F1832                 jz      short loc_804F1842
nt:804F1834                 push    edi
nt:804F1835                 call    near ptr nt_IopRemoveTimerFromTimerList
nt:804F183A                 push    0
nt:804F183C                 push    edi
nt:804F183D                 call    near ptr nt_ExFreePoolWithTag
nt:804F1842
```

At the end you can detach from the kernel and resume it or detach from the kernel and keep it suspended.

To detach and resume, simply select the "Debugger / Detach", however to detach and keep the kernel suspended select "Debugger / Terminate Process".

Debugging the kernel through kdsrv.exe

In some cases, when debugging a 64bit kernel using a 1394 cable then 64bit drivers are needed, thus dbgeng (32bits) will not work. To workaround this problem we need to run the kernel debugger server from the x64 debugging tools folder and connect to it:

- Go to “Debugging Tools (x64)” installation
- Run kdsrv.exe (change the port number/transport appropriately):
  - kdsrv -t tcp:port=6000
- Now run ida64 and specify the following connection string (change the transport value appropriately):
  - kdsrv:server=@{tcp:port=6000,server=127.0.0.1},trans=@{**com:port=\\.\pipe\com_3,baud=115200,pipe**}