

Debugging Windows Applications with IDA WinDbg Plugin

Copyright 2010 Hex-Rays SA

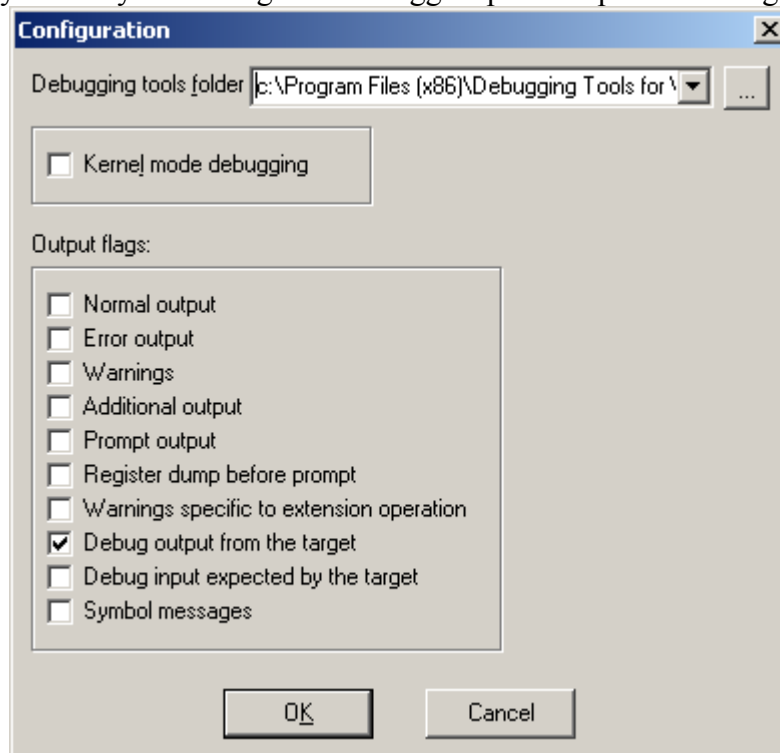
Quick overview:

In IDA 5.4, we created a new debugger plugin that uses Microsoft's Debugging Engine. The debugging engine is also used by Microsoft's WinDbg debugger to debug user mode applications and live kernels.

To get started, you need to install the latest Debugging Tools from Microsoft website:

<http://www.microsoft.com/whdc/devtools/debugging/default.msp>

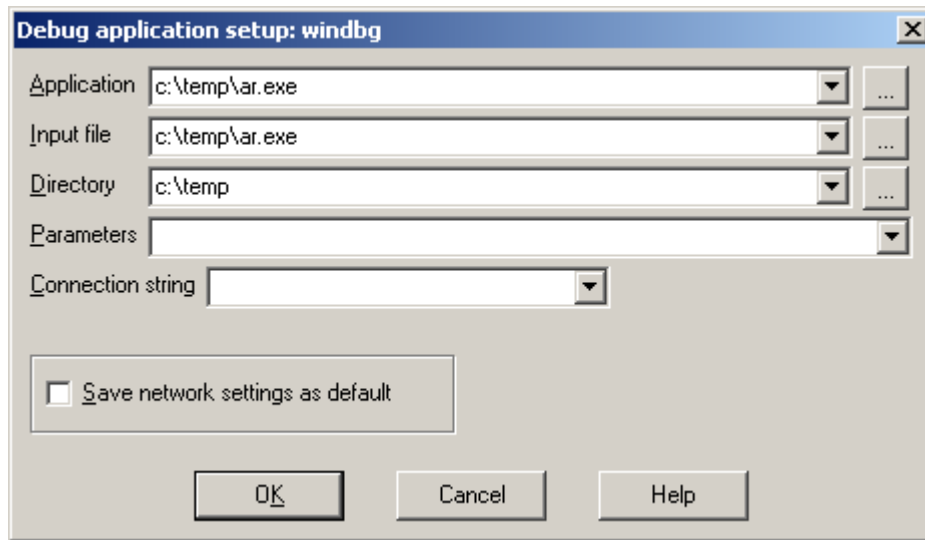
After installing it, make sure you « Switch Debugger » and select the WinDbg debugger. Also make sure that you verify the settings in “Debugger specific options” dialog:



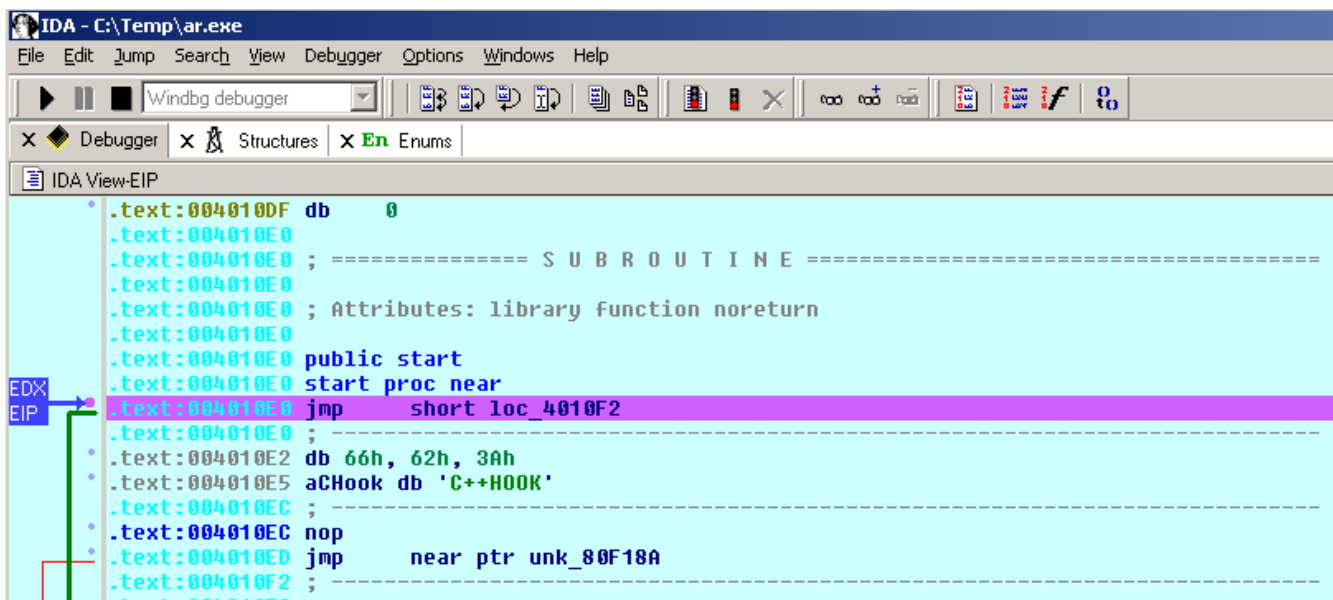
- **Debugging tools folder:** It is where the debugging tools are installed. IDA will try to guess (by searching the PATH environment variable and Program Files folder) and if it fails will use “dbgeng.dll” found in the system directory (which could be outdated).
- **Kernel mode:** Toggle this switch if you want to attach to a live kernel.
- **Output flags:** These flags tell the debugging engine which kind of output messages to display and which to omit

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

Now that we configured the debugger for this session, we will edit the process options and leave the connection string value empty because we intend to debug a local user mode application.

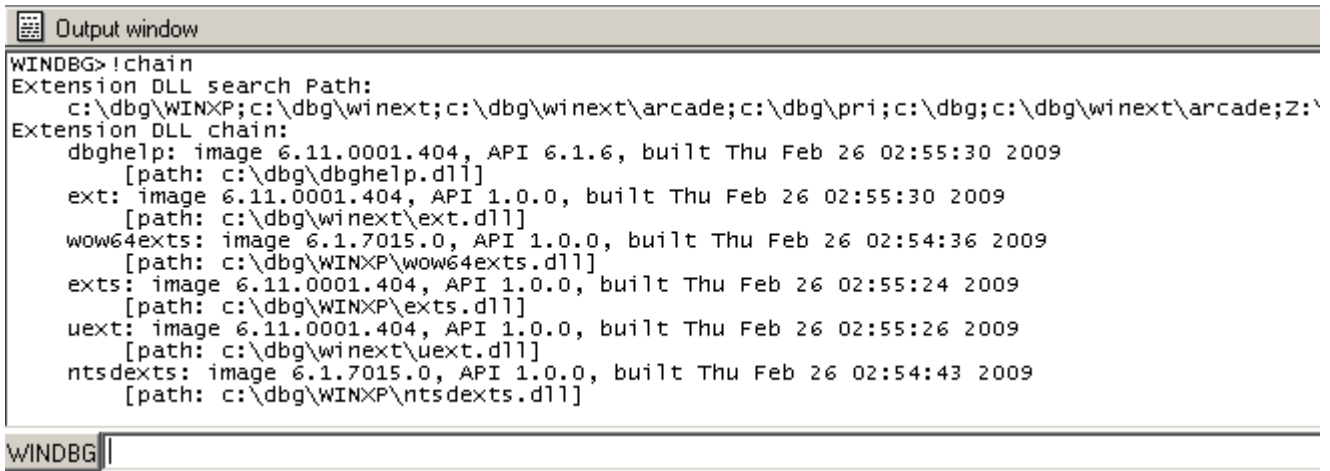


Let us put a breakpoint at the beginning and start the debugger:



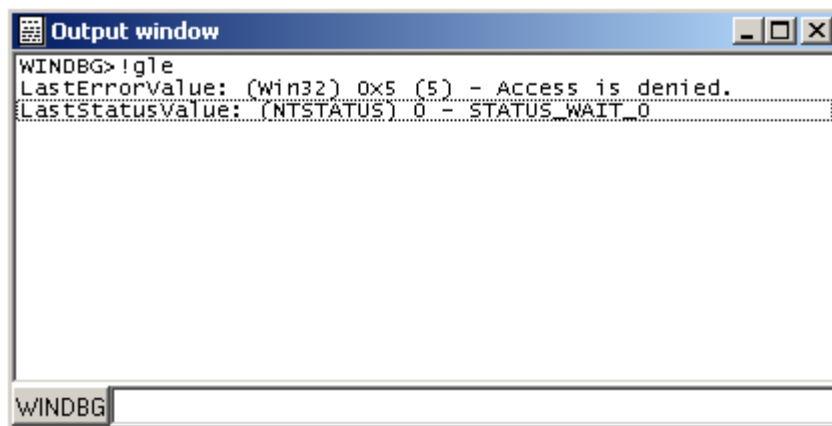
The Windbg plugin is very similar to IDA's Win32 debugger plugin, nonetheless by using the former, one can benefit from the command line facilities and the extensions that ship with the debugging tools.

For example, we can type “!chain” to see the registered windbg extensions:



```
Output window
WINDBG>!chain
Extension DLL search Path:
  c:\dbg\WINXP;c:\dbg\winext;c:\dbg\winext\arcade;c:\dbg\pri;c:\dbg;c:\dbg\winext\arcade;Z:\
Extension DLL chain:
  dbghelp: image 6.11.0001.404, API 6.1.6, built Thu Feb 26 02:55:30 2009
    [path: c:\dbg\dbghelp.dll]
  ext: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:55:30 2009
    [path: c:\dbg\winext\ext.dll]
  wow64exts: image 6.1.7015.0, API 1.0.0, built Thu Feb 26 02:54:36 2009
    [path: c:\dbg\WINXP\wow64exts.dll]
  exts: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:55:24 2009
    [path: c:\dbg\WINXP\exts.dll]
  uext: image 6.11.0001.404, API 1.0.0, built Thu Feb 26 02:55:26 2009
    [path: c:\dbg\winext\uext.dll]
  ntsdexts: image 6.1.7015.0, API 1.0.0, built Thu Feb 26 02:54:43 2009
    [path: c:\dbg\WINXP\ntsdexts.dll]
WINDBG
```

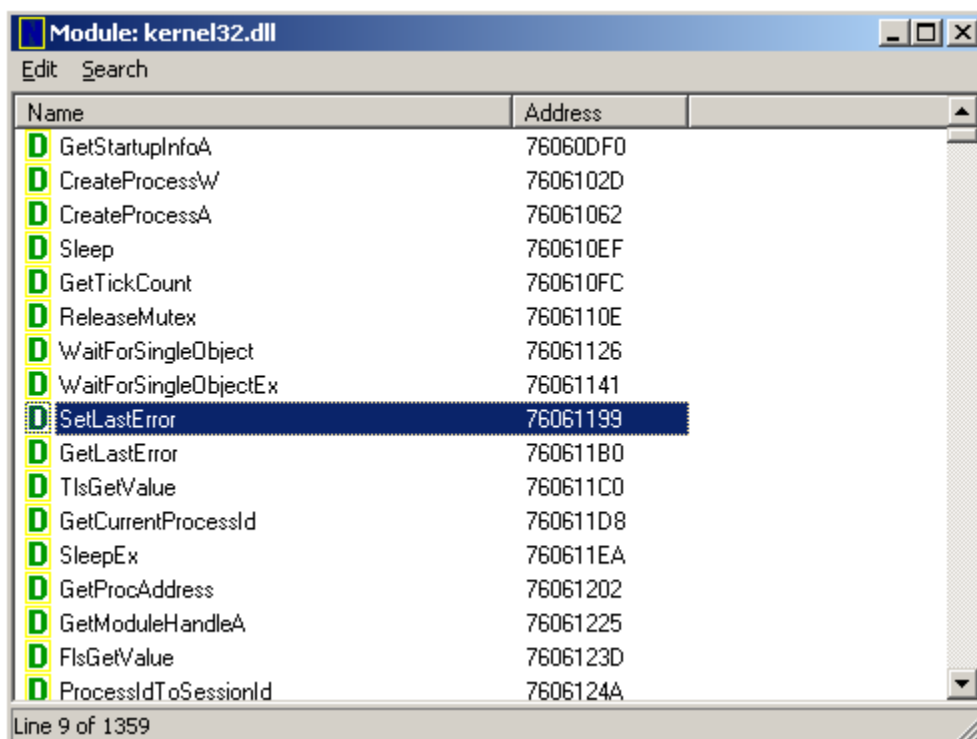
We can use the “!gle” command to get the last error value of a given Win32 API call.



```
Output window
WINDBG>!gle
LastErrorValue: (Win32) 0x5 (5) - Access is denied.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0
WINDBG
```

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

Normally, if the debug symbol source is not set, then the plugin will show only exported names, for example kernel32.dll displays 1359 names:



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

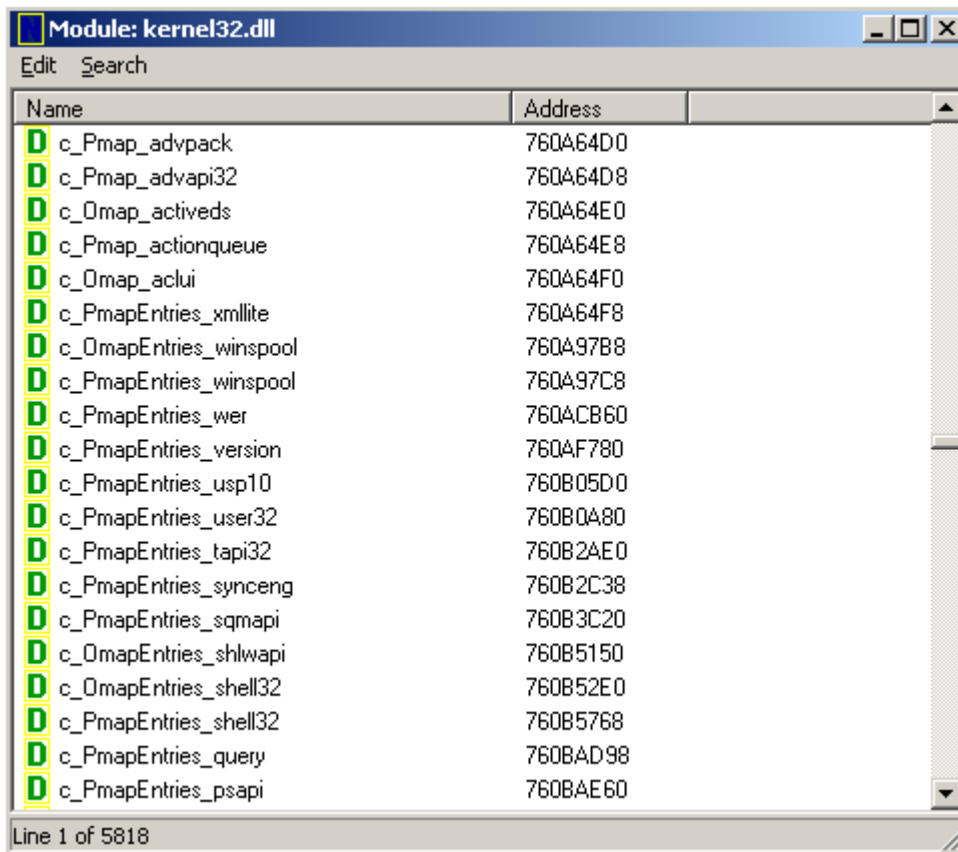
```
set _NT_SYMBOL_PATH=svr*C:\Temp\pdb*http://msdl.microsoft.com/download/symbols
```

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



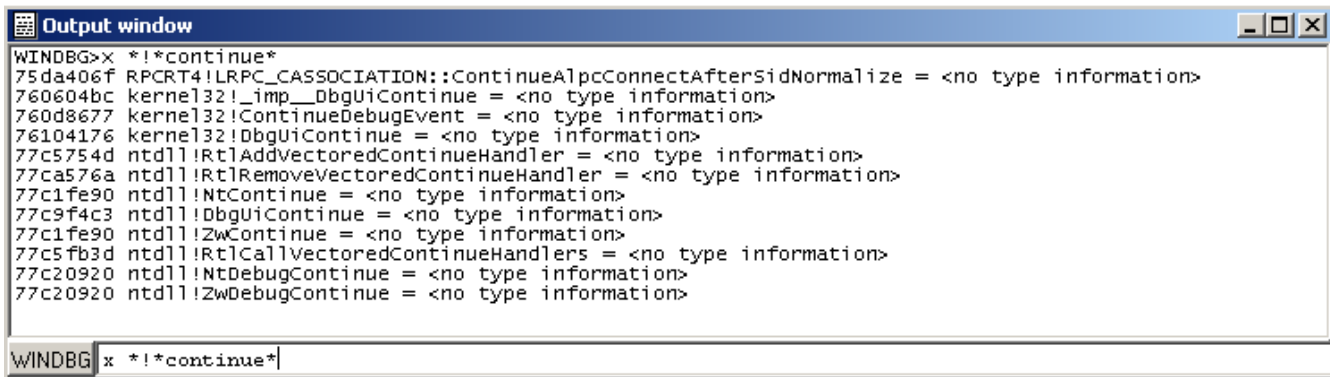
(And typing “.reload /f” to reload the symbols)

Now let us try again and see how many debug names will be 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 with IDA and the debugger installed.

To start a remote process:

- On “pcA” we type:
dbgsrv -t tcp:port=5000
(you can change the port number)
- On “pcB” we setup IDA and the 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 we can run the program and debug it remotely.

To attach to a remote process, we 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:

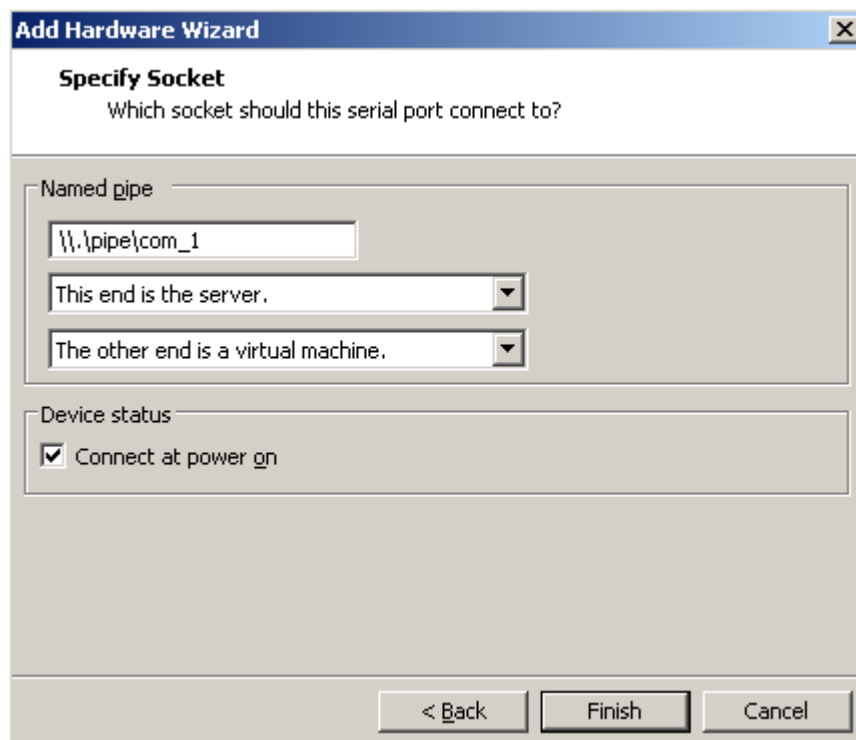
We run the VM and then edit “boot.ini” file and add one more entry (see in bold):

```
[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
```

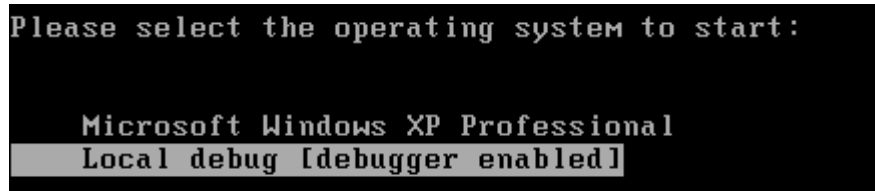
For MS Windows Vista please see: <http://msdn.microsoft.com/en-us/library/ms791527.aspx>

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

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

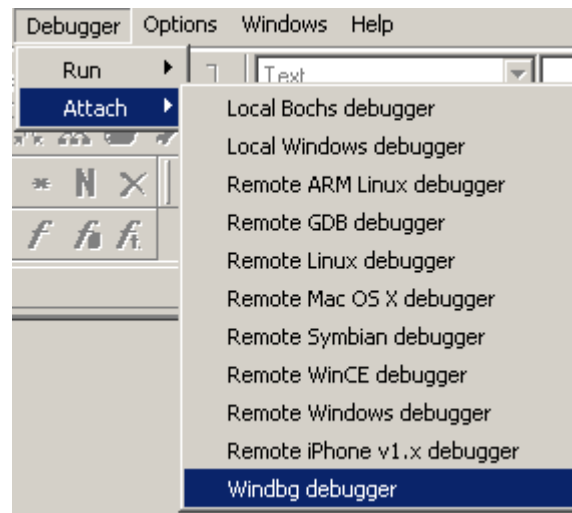


We finished configuring the virtual machine, now let us start it and make sure we select “Local debug” from the boot menu:

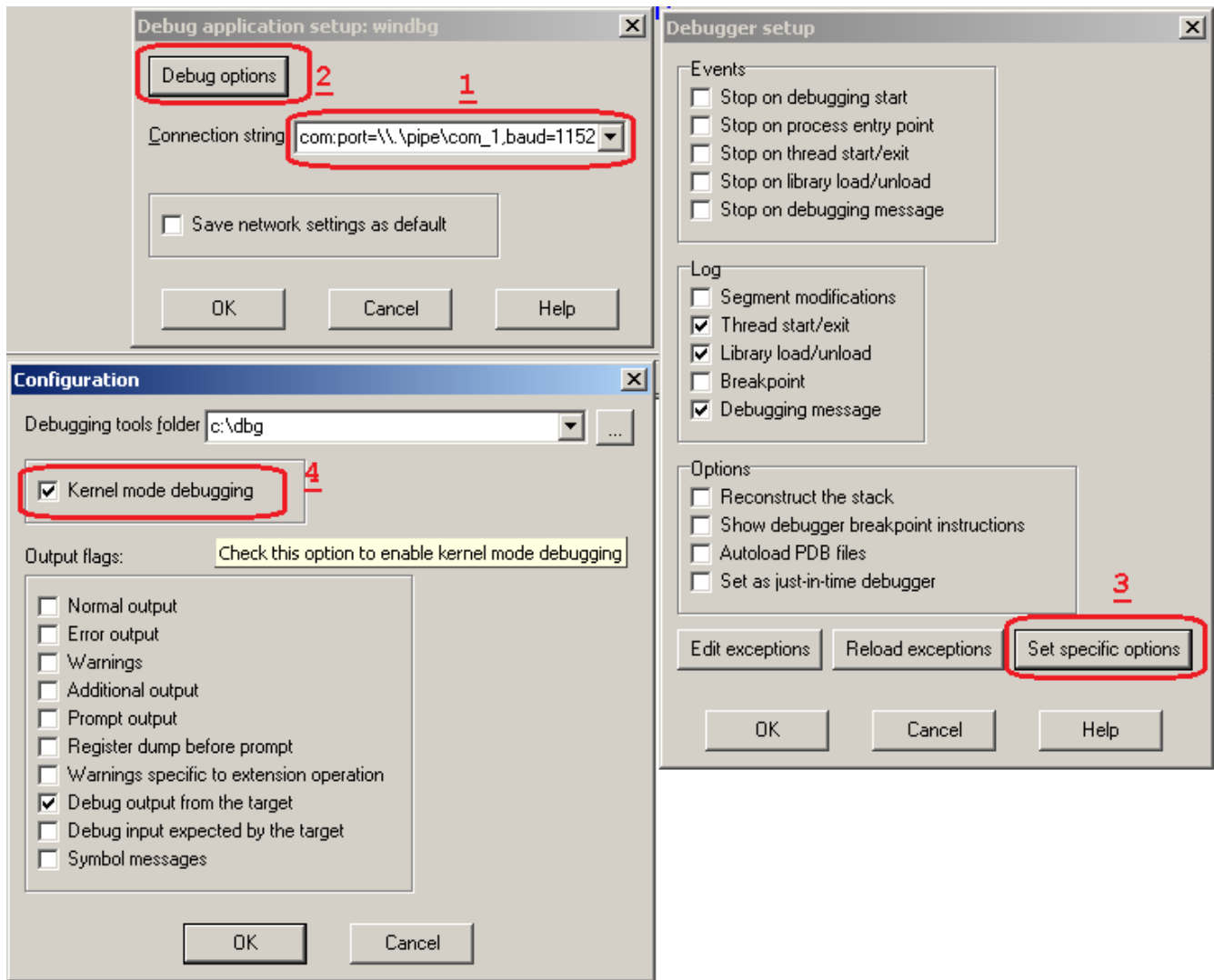


Configuring Windbg debugger plugin:

Now we run IDA, and select Debugger / Attach / Windbg

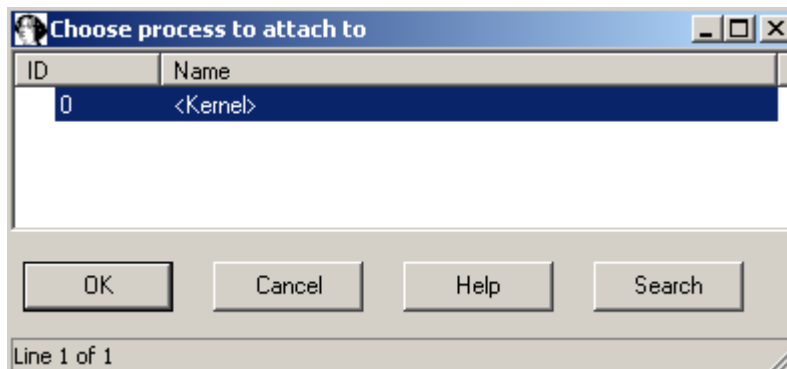


Then we configure it to use “Kernel mode” debugging and use the following connection string:
“com:port=\\.\pipe\com_1,baud=115200,pipe”



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

Now we can 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 ; -----
```

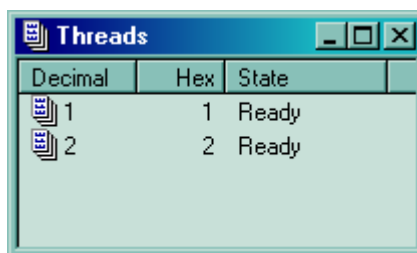
If you do not see named labels it is probably because you did not setup the debug symbols properly. (Please refer to the beginning of this article to learn how)

Now we are ready to start debugging the kernel.

In kernel mode IDA will display “virtual” threads. Each thread corresponds to a processor. For example a two processor configuration yields:

Device	Summary
Memory	256 MB
Processors	2

VMWare configuration



Decimal	Hex	State
1	1	Ready
2	2	Ready

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_IoDeleteDevice
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”.

Remember that you can send commands to the debugger engine in the same way you would do that if you were using WinDbg.