netmatze

.net Blog

About these ads

using windbg.exe and sos.dll to debug a .net 4.0 application

Posted: 24/08/2012 | **Author:** netmatze | **Filed under:** .net, Debug, windbg | **Tags:** .net 4.0, debug, sos, sos.dll, windbg |1 Comment
In this post i want to write about using windbg and sos.dll. I often have the problem that there is a bug at a client system and the only chance to debug it is windbg (with sos.dll). Every time i do so i have to search in many posts and forums to get all the information i need. So in this post i want to summerize the things you need to do find a bug in a .net 4 application with windbg. First of all the are a x86 and a x64 version of windbg (to find out what version you need read **Choosing the 32-Bit or 64-Bit Debugging Tools**). The next step is to download the windows sdk. The actual version is this one (**windows sdk**). After installing the sdk you can start the windbg.exe (the good thing is that you can copy the windbg.exe to every client system and start immediately with the debugging). Most of the time i have a scenario where there is a bug in the application on that specific client system and i need to find where the bug is. To do so there are several steps:

1. I need to attach windbg to the application i need to inspect. To do so i start the application and use File -> Attach to Process in windbg to attach windbg to the application.
2. Now i am attached to the application and can break (ctrl-break) and continue (F5) the execution of the client application.
3. Most of the time i am debugging managed applications so i need an extension to windbg. This extension is the sos.dll (son of strike). To load this extension the best way is to use:

   ```
   .loadby sos clr
   ```

   If you want to load the 32bit version of sos.dll for the .net framework 4 you have to use:

   ```
   .load C:\Windows\Microsoft.NET\Framework\v4.0.30319\sos.dll
   ```

   If you get no error from windbg every thing is ok.

4. When i use windbg most of the time i search for Exceptions that occur in the client application. When i have found the exception that occurs i have a clue where i have to search in my source code to find the bug. So i want that windbg stops the execution of the client application if the client application rises an exception. To do so i use Debug -> Event Filter and search for CLR exceptions and put the value Execution at Enabled. That means that windbg stops at every CLR exception.

After all that work is doen we can realy start to find the bug. To do so we need some sos commands.

We need:

1. `!threads`
   that shows us all threads in our application
2. `~[threadnumber]s`
   to switch to the thread the exception is thrown
3. `!pe`
   shows the last exception that occured at the active thread
4. `~*e !pe`
   shows the last exception that occured at any thread
5. `!CLRStack [-p] [-l]`
   to show the clr stack of the active thread, -p means to show the overtaken parameters, -l means to show the local variables
6. `!Dumpheap [-stat] [-mt [methodtable address of the object]]`
   shows the complete heap (could shut down your maschine when you have a big application), -stat groups the heap objects by there type, -mt shows the heap object with exactly that methodtable address.
7. `!DumpObj [address of the object]`
   shows the information about a specific clr object
8. `!DumpClass [EEClass address]`
   displays information about the EEClass structure
9. `!DumpMT [-MD] [methodetable address]`
   displays information about a specific methodtable, -MD shows all methodes in that methodtable
10. `!DumpMD [MethodDesc address]`
   displays information about a specific method
11. `!DumpIL [address of the methode]`
   shows the IL code of a methode
12. `!ProcInfo`
   shows infomation about your maschine

This are the commandos i use to find the bug in the application. A list of all sos.dll commands is shown at the **msdn**.
Now i want to show how i use this commands to find information that gives me a hint to the code that produces the bug. To do so i have implemented a simple program that raises Exceptions in the actual thread and in a different thread.

The class ExceptionThrower raises a custom Exception in the ThrowException method.

```
public class ExceptionThrower
{
    public static int staticValue = 1;
    public int value;

    public void ThrowException()
    {
        throw new WindbgCustomException();
    }

    public override string ToString()
    {
        return "ExceptionThrower";
    }
}
```

The class Processor calls the ThrowException method in the UI thread and in a new thread.

```
public class Processor
{
    private ExceptionThrower exceptionThrower = new ExceptionThrower();

    public void RaiseException()
    {
        try
        {
            exceptionThrower.ThrowException();
        }
        catch (Exception ex)
        {

        }
    }

    public void RaiseExceptionInDifferentThread()
    {
        Thread thread = new Thread(() =>
        {
            try
            {
                exceptionThrower.ThrowException();
            }
            catch (Exception ex)
            {

            }
        });
        thread.Start();
    }
}
```

The simples case is that you start windbg, attach it to the process raise the CLRException and type:

```
(11d8.fe4): CLR exception - code e0434352 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=003dec6c ebx=00000005 ecx=00000005 edx=00000000 esi=003ded18
edi=00843850
eip=762db727 esp=003dec6c ebp=003decbc iopl=0 nv up ei pl nz na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000206
KERNELBASE!RaiseException+0x58:
762db727 c9 leave
```

With the pe command we can get information about the last raised exception.

```
!pe
Exception object: 02471a3c
Exception type: WinformsWindbgApplication.WindbgCustomException
Message:
InnerException:
StackTrace (generated):
StackTraceString:
HResult: 80131500
```

Now we have the information about the Exception. With

```
!CLRStack
OS Thread Id: 0xfe4 (0)
Child SP IP Call Site
003dedc4 762db727 [HelperMethodFrame: 003dedc4]
003dee14 001b06c6
WinformsWindbgApplication.ExceptionThrower.ThrowException()
003dee24 001b0652 WinformsWindbgApplication.Processor.RaiseException()
003dee58 001b05ec
WinformsWindbgApplication.frmMain.btnRaiseException_Click(System.Object,
System.EventArgs)
...
```

we see the call stack and can see in what method the exception is raised. If we want more information about the object that raised the Exception we can execute

```
!CLRStack -p
OS Thread Id: 0xfe4 (0)
Child SP IP Call Site
003dedc4 762db727 [HelperMethodFrame: 003dedc4]
003dee14 001b06c6
WinformsWindbgApplication.ExceptionThrower.ThrowException()
PARAMETERS:
this (0x003dee18) = 0x0244c518
003dee24 001b0652 WinformsWindbgApplication.Processor.RaiseException()
PARAMETERS:
this (0x003dee2c) = 0x0244c50c
003dee58 001b05ec
WinformsWindbgApplication.frmMain.btnRaiseException_Click(System.Object,
System.EventArgs)
```

```
PARAMETERS:
this (0x003dee58) = 0x0244c3bc
sender (0x003dee5c) = 0x0244f3d4
```

that shows us the overtaken parameter. Here we can take the address of the this pointer to find the information about the   ExceptionThrower object.

```
!dumpobj 0x0244c518
Name: WinformsWindbgApplication.ExceptionThrower
MethodTable: 00146c08
EEClass: 001c0624
Size: 12(0xc) bytes
File: C:\Users\mathiass\Desktop\WinformsWindbgApplication.exe
Fields:
MT Field Offset Type VT Attr Value Name
6e1f2978 4000002 4 System.Int32 1 instance 0 value
6e1f2978 4000001 24 System.Int32 1 static 1 staticValue
```

If we want information about the Processor object we can use the this pointer from the second entry of the CLRStack.

```
!dumpobj 0x0244c50c
Name: WinformsWindbgApplication.Processor
MethodTable: 00146b74
EEClass: 001c055c
Size: 12(0xc) bytes
File: C:\Users\mathiass\Desktop\WinformsWindbgApplication.exe
Fields:
MT Field Offset Type VT Attr Value Name
00146c08 4000007 4 ....ExceptionThrower 0 instance 0244c518
exceptionThrower
```

If we want information about the class the ExceptionThrower object is generated of we can use the dumpClass command. (Here we use the EEClass address of the ExceptionThrower object that is marked strong)

```
!dumpclass 001c0624
Class Name: WinformsWindbgApplication.ExceptionThrower
mdToken: 02000002
File: C:\Users\mathiass\Desktop\WinformsWindbgApplication.exe
Parent Class: 6ded3ef8
Module: 00142e9c
Method Table: 00146c08
Vtable Slots: 4
Total Method Slots: 6
Class Attributes: 100001
Transparency: Critical
NumInstanceFields: 1
NumStaticFields: 1
MT Field Offset Type VT Attr Value Name
6e1f2978 4000002 4 System.Int32 1 instance value
6e1f2978 4000001 24 System.Int32 1 static 1 staticValue
```

If we want to look at the methodes (the methodeTable) of the object we need the DumpMT command and the address of the methodeTable. That address can be found with Dumpobj or Dumpclass.

```
!DumpMT -MD 0x00146c08
EEClass: 001c0624
Module: 00142e9c
Name: WinformsWindbgApplication.ExceptionThrower
mdToken: 02000002
File: C:\Users\mathiass\Desktop\WinformsWindbgApplication.exe
BaseSize: 0xc
ComponentSize: 0x0
Slots in VTable: 7
Number of IFaces in IFaceMap: 0
--------------------------------------
MethodDesc Table
Entry MethodDesc JIT Name
0014c0a5 00146bf0 NONE
WinformsWindbgApplication.ExceptionThrower.ToString()
6e0fe2e0 6ded493c PreJIT System.Object.Equals(System.Object)
6e0fe1f0 6ded495c PreJIT System.Object.GetHashCode()
6e181600 6ded4970 PreJIT System.Object.Finalize()
001b01e0 00146bf8 JIT WinformsWindbgApplication.ExceptionThrower..ctor()
001b01b0 00146c00 JIT WinformsWindbgApplication.ExceptionThrower..cctor()
001b0690 00146be4 JIT WinformsWindbgApplication.ExceptionThrower.
ThrowException()
```

Here we see all Methodes of the ExceptionThrower class (including the JIT status). If we want informatioin to a specific method we can use the DumpMD with the MethodDesc address.

```
!DumpMD 00146be4
Method Name: WinformsWindbgApplication.ExceptionThrower.ThrowException()
Class: 001c0624
MethodTable: 00146c08
mdToken: 06000001
Module: 00142e9c
IsJitted: yes
CodeAddr: 001b0690
Transparency: Critical
```

We can also see display the IL code of the method with the DumpIL command the MethodDesc address of the method.

```
!DumpIL 00146be4
ilAddr = 00102050
IL_0000: nop
IL_0001: newobj WinformsWindbgApplication.WindbgCustomException::.ctor
IL_0006: throw
```

Here we see the creation of the WindbgCustomException object and the throw of that exception object. If the exception is raised in another thread sometimes windbg does not jump to the correct thread, so we have to find the thread and make that thread the active thread.

With the ~*e !pe command we can find all exceptions at all threads.

```
~*e !pe
There is no current managed exception on this thread
The current thread is unmanaged
There is no current managed exception on this thread
The current thread is unmanaged
The current thread is unmanaged
Exception object: 024771f0
Exception type: WinformsWindbgApplication.WindbgCustomException
Message:
InnerException:
StackTrace (generated):
StackTraceString:
HResult: 80131500
```

As we see there is are no exceptions at the first and second managed threads but one at the third managed thread. So we use the threads command shows us all threads.

```
!threads
ThreadCount: 4
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 1
Hosted Runtime: no
PreEmptive GC Alloc Lock
ID OSID ThreadOBJ State GC Context Domain Count APT Exception
0 1 fe4 00843850 6020 Enabled 00000000:00000000 0083d858 1 STA
2 2 109c 0084fba8 b220 Enabled 00000000:00000000 0083d858 0 MTA
(Finalizer)
XXXX 3 0088d9b8 19820 Enabled 00000000:00000000 0083d858 0 Ukn
5 4 1288 0088f3a0 b020 Enabled 00000000:00000000 0083d858 0 MTA
WinformsWindbgApplication.WindbgCustomException (024771f0)
```

The threads with the number 0, 2 and 5 are managed threads so we make the thread with the number 5 our active thread. Therefor we use the `~5s` command. The we use the `!pe` command again.

```
!pe
Exception object: 024771f0
Exception type: WinformsWindbgApplication.WindbgCustomException
Message:
InnerException:
StackTrace (generated):
StackTraceString:
HResult: 80131500
```

And now we are at the right thread and can search for the bug. There are two useful command left the `!dumpheap -stat` command. This command shows all different objects on the heap (and there number).

```
!dumpheap -stat
total 0 objects
```

```
Statistics:
MT Count TotalSize Class Name
6e1f65a4 1 12 System.Nullable`1[[System.Boolean, mscorlib]]
6e1f5f90 1 12 System.Security.HostSecurityManager
6e1f59b8 1 12
System.Collections.Generic.ObjectEqualityComparer`1[[System.Type,
mscorlib]]
6e1f4bc4 1 12 System.Security.Permissions.ReflectionPermission
6e1f4ae0 1 12 System.Security.Permissions.FileDialogPermission
6e1f4a00 1 12 System.Security.PolicyManager
6e1f4918 1 12 System.Security.SecurityDocument
```

And the

```
!ProcInfo
---------------------------------------
Environment
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\mathiass\AppData\Roaming
CLIENTNAME=MATHIASS
CommonProgramFiles=C:\Program Files (x86)\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
COMPUTERNAME=TERMINALTEST
...
```

command that shows information about your maschine.
So i often used windbg to find errors at maschines i could not install Visual Studio or a remote debugger and with this few commands i found most of them.

One Comment on “using windbg.exe and sos.dll to debug a .net 4.0 application”

1. *Certificate pinning: translating Chrome settings into EMET | Random Oracle* says:
   13/05/2013 at 18:07
   […] A good way to observe this is running the above command line under a debugger such as windbg, with CLR exception handling enabled to inspect uncaught managed exceptions. In the above example, the command reports zero entries […]

   Reply

Follow “netmatze”