SOS – “Son of Strike”

By Mark Smith

One of the things DevelopMentor is known for is deep dives into technology. In that vein, this month I’d like to focus on one of the things you would learn if you attended our intense Guerrilla .NET class or the 9-5 Effective .NET class we offer for onsite delivery – “Son of Strike”.

For those of you who are not familiar with SOS.DLL, it’s a debugging extension used primarily with WinDBG, but also loadable into Visual Studio 2005. It gives you an “inner view” into the CLR, allowing you to peek around and explore the data structures of .NET.

In our Guerrilla .NET and Effective .NET class, we talk about memory leaks and how to debug them – here I’ll present an ASP.NET application that is leaking memory by holding references to page objects after they have already returned their content. I set this example up by having the page class hook up an event handler to some global event and then never remove the handler – this causes the page to be “kept alive” and gives me the super-size application I am looking for. This, of course, is bad form because the System.Web.UI.Page object is intended to be a transient object - it goes away at the end of the request - in production code, I would really bind the event to a handler in **global.asax** instead – and that’s why we have to debug it!

Generally, when debugging a memory exception, the first thing you want to know is what is taking up all the memory? Here, I’ll run the web page a few times, note that the working set continually increases and decide I’ve got a memory leak. So, to debug that, the first thing I need to do is take a hang dump of the process. You do that through the **ADPLUS.VBS** script which is supplied with the Windows Debugging Tools.

I can then load this dump up into WinDBG and started poking around. First, I’ll load SOS into memory:

```
0:000> .load sos
```

Next, let’s take a look at the managed heap and see what we’ve got in memory:

```
0:000> !DumpHeap -stat
total 36955 objects
Statistics:
      MT    Count    TotalSize Class Name
7b4ecd7c        1           12
System.Windows.Forms.ButtonInternal.ButtonPopupAdapter
7b481f00        1           12
System.Windows.Forms.LinkLabel+LinkComparer
7b475ca8        1           12
System.Windows.Forms.FormCollection
7b474f8c        1           12
System.Windows.Forms.Layout.DefaultLayout
7b4749e0        1           12
System.Windows.Forms.ClientUtils+WeakRefCollection
7b473ca8        1           12
System.Windows.Forms.Layout.ArrangedElementCollection
7a755834        1           12
System.Diagnostics.PerformanceCounterCategoryType
7a753394        1           12
System.Diagnostics.TraceOptions
7a71a710        1           12 System.Net.TimeoutValidator
.......
00166030      891       169744      Free
054d24d4     3128       187680 System.Web.UI.LiteralControl
0548cbd4      519       197220 ASP.default_aspx
791242ec     1545       297960
System.Collections.Hashtable+bucket[]
79124670     1185      1090500 System.Char[]
79124228    11961      1279380 System.Object[]
790fa3e0    19149      1561392 System.String
Total 110069 objects
```

From this, I can see that we've got 519 ASP.default_aspx objects in memory – I'm going to guess that these are actually the page objects for default.aspx (the name kinda gives it away).

So, next let's use **DumpHeap** to just look at this specific type by giving it a metadata token:

```
0:000> !DumpHeap -mt 0548cbd4
```

```
 Address       MT     Size
.....
01854ff0 0548cbd4      380
01860130 0548cbd4      380
0186b2b4 0548cbd4      380
018773f8 0548cbd4      380
01882538 0548cbd4      380
0188d6bc 0548cbd4      380
01898840 0548cbd4      380
018a39c4 0548cbd4      380
018aeb48 0548cbd4      380
total 519 objects
Statistics:
      MT    Count    TotalSize Class Name
0548cbd4      519       197220 ASP.default_aspx
Total 519 objects
```

With this output, I can see how much total size is being taken up, but more importantly, I see a list of all the objects and their address. With a single instance, I can use the *very* cool **GCRoot** command to determine why a page instance was still rooted and therefore not collectable.

Note: it appears that the GCRoot command doesn't work well inside VS.NET 2005 - apparently the SOS debugging extension is using some debugger API which isn't fully supported in VS.NET, so you need to familiarize yourself with WinDBG to do this.

```
0:000> !gcroot 018aeb48
Note: Roots found on stacks may be false positives. Run
"!help gcroot" for
more info.
Scan Thread 0 OSTHread 3a8
Scan Thread 2 OSTHread e8
Scan Thread 3 OSTHread 1a8
Scan Thread 6 OSTHread 7d4
Scan Thread 7 OSTHread 2b4
Scan Thread 8 OSTHread fdc
Scan Thread 9 OSTHread eac
DOMAIN(001E5E08):HANDLE(Pinned):12312f0:Root:0226c498(Syste
m.Object[])->
```

```
018af940(System.EventHandler)->
0186c0ac(System.Object[])->
018af920(System.EventHandler)->
018aeb48(ASP.default_aspx)
```

With this output, I can tell that my default_aspx object is being kept alive through an EventHandler as I expected. The interesting thing about this output is I cannot tell which event handler is keeping it alive - i.e. there is nothing in this root list that points to a specific object holding it other than a object[]. That essentially means this is a static event and there isn't an actual object around on the heap for it. This just makes the debugging exercise more interesting - after all if it were an instance event then I would see the class name at the top of the root list and we could stop right here.

So, my next step is to dump the event handler to try to identify what method it is wrapping - I could then search the code for this method and find out where it is being bound. I dump out the **EventHandler** object using the handy dandy dump object command (do):

```
0:000> !do 018af920
Name: System.EventHandler
MethodTable: 7910d61c
EEClass: 790c3a7c
Size: 32(0x20) bytes

(C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e
089\mscorlib.dll)
Fields:
      MT    Field   Offset                 Type VT     Attr
Value Name
790f9c18  40000f9        4        System.Object  0 instance
018aeb48 _target
79109208  40000fa        8 ...ection.MethodBase  0 instance
00000000 _methodBase
790fe160  40000fb        c        System.IntPtr  0 instance
88962888 _methodPtr
790fe160  40000fc       10        System.IntPtr  0 instance
0 _methodPtrAux
790f9c18  4000106       14        System.Object  0 instance
00000000 _invocationList
```

```
790fe160  4000107       18        System.IntPtr  0 instance
0 _invocationCount
```

From this I get the internals of the EventHandler object. It gives me some specific information:

1. **_target** is the object the delegate is holding onto - my default_aspx in this case. The value *better* match up with my original object!
2. **_methodBase** is used for dynamic code and is null here.
3. **_methodPtr** is the method associated with the instance, or possibly a dynamically generated thunk for static methods. This is what I'm interested in.
4. **_methodPtrAux** is used for static methods; this holds the method descriptor and the **_methodPtr** is a dynamically generated block of code to remove the **this** reference. Noting that this is null, I can infer that this is an instance method bound to the delegate.
5. **_invocationList** and **_invocationCount** are used for Multicast Delegates -- these are both zero indicating that this is a single delegate and there is no chain to follow.

My next step is to try to get a valid method name from the **_methodPtr** so I convert that to hex and pass it into IP2MD.

```
0:000> !ip2md 0n88962888
Failed to request MethodData, not in JIT code range
```

This is a pretty common error and simply means that the method may not have been JITted yet, or may be a dynamic block of code which never went through the JIT compiler. In .NET 1.1 we could start dumping method descriptors and trying to match up the address (DumpClass -md) for this class and it's base class.

However, under .NET 2.0 this rarely works anymore - the address doesn't appear to ever match up to a valid descriptor. However, it clearly is part of the managed heap due to its address. So, failing to locate this address, I first try to disassemble the code:

```
0:000> !u 0n88962888
Unmanaged code
054d7748 e862289b74      call
mscorwks!LogHelp_TerminateOnAssert+0x3f5f (79e89faf)
```

```
054d774d 5e              pop     esi
054d774e cc              int     3
054d774f cc              int     3
054d7750 38c8            cmp     al,cl
054d7752 48              dec     eax
054d7753 05a0774d05      add     eax,54D77A0h
054d7758 0100            add     dword ptr [eax],eax
054d775a 0011            add     byte ptr [ecx],dl
054d775c 0000            add     byte ptr [eax],al
```

This doesn't even look like valid code to me – this looks like random data, so let's dump it out to get a better idea of what it is using the dump data (dd) command:

```
0:000> dd 0n88962888
054d7748  9b2862e8 cccc5e74 0548c838 054d77a0
054d7758  11000001 90000000 054d77a0 11000002
054d7768  90000004 00000000 054d77a0 00000000
```

The third and fourth DWORD look interesting because they appear to fall in the managed heap as well -- so let's dump them out to try to figure out what they are – through trial and error, we'll find out that the third DWORD is actually a method descriptor:

```
0:000> !dumpmd 0548c838
Method Name: _Default.OnDatabaseHasChanged(System.Object,
System.EventArgs)
Class: 054ab574
MethodTable: 0548c86c
mdToken: 06000004
Module: 0548c35c
IsJitted: no
m_CodeOrIL: ffffffff
```

This is the real method bound to the delegate instance. The other DWORD appears to be a metadata reference to the event owner itself:

```
0:000> !dumpmt 054d77a0
EEClass: 0551940c
Module: 048ac9ec
```

```
Name: DatabaseMonitor
mdToken: 02000002
(C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Temporary
ASP.NET
Files\leakypage\2a399ab5\b1e04c63\App_Code.onwg1zqj.dll)
BaseSize: 0xc
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 8
```

From here I can see the module that this code is defined in (the dynamically generated App_Code directory) and the name (**DatabaseMonitor**). This gives me enough information to stop here and begin looking at the code itself - specifically where default.aspx binds it's OnDatabaseHasChanged method to the DatabaseMonitor static class defined somewhere in the App_Code directory. If I didn't have the source code available, I could locate the module and then save it out to a file through the **savemodule** command:

```
0:000> lm m App_Code_onwg1zqj
start    end        module name
04de0000 04de8000   App_Code_onwg1zqj C (no symbols)
0:000> !savemodule 04de0000 c:\appcode.dll
3 sections in file
section 0 - VA=2000, VASize=504, FileAddr=200, FileSize=600
section 1 - VA=4000, VASize=2c8, FileAddr=800, FileSize=400
section 2 - VA=6000, VASize=c, FileAddr=c00, FileSize=200
```

I could then ILDasm or Reflector the generated assembly and look for my bug from that. This is just a quick look at some of the exercises we take students through in our classes – I would encourage you to learn WinDBG and SOS as they are extremely powerful tools to have in your repertoire! If you have any questions or comments, I'd love to hear them – email me at marksm@develop.com!