**.NET Debugging Demos Lab 3: Memory**

Todays debugging puzzle will be a Memory investigation. This time we are going to stress the BuggyBits site to create a high memory usage situation and figure out what caused it. This lab is a bit lengthier because I wanted to show various aspects of a memory investigation. Once you know the relationship between the numbers in the dump and perfmon etc. you can often skip some parts, but if it is possible to gather I would still recommend gathering performance logs for completeness.

**Set up a performance monitor log**

1. Start the app and browse to the Links page to start the iisexpress.exe process
2. Check that iisexpress.exe is up and running and ready to collect counters
3. >dotnet-counters ps
4. 1496 iisexpress C:\Program Files\IIS Express\iisexpress.exe
5. Start monitoring the System.Runtime counters
6. dotnet-counters monitor -p 1496 --refresh-interval 1 --counters System.Runtime

Watch the counters as you reproduce the problem

**Reproduce the problem**

* Run .\ToolsAndScripts\tinyget.ps1 -url https://localhost:44350/Links -numTimes 4000

**Note:** You can lower the amount of loops slightly if you find that your process crashes because of OutOfMemory exceptions, I set it this high so that we would get some kind of output in perfmon.

**Examine the counters**

* What are the values for GC Heap Size (MB), and Working Set (MB) after the test?
* Can you tell from this if it appears to be a native leak or a .net memory leak, or a leak on hte loader heap?
* Does the memory go down if you wait some time or stay the same?
* Looking at the counters for GC counts - do any collections occur after the test is finished? why or why not?

**Examine task manager**

1. Add the column Working Set in Task Manager
2. Compare the values for Memory and Working Set to the values for Working Set in the performance counters.

**Note:** since they are given in K in task manager you need to multiply these values by 1024 if you want to compare them

* What does Memory show?
* What does Working set show?

**Get a memory dump**

1. Capture a dump with procdump -ma iisexpress.exe or dotnet-dump -n iisexpress.exe

**Analyze the memory dump**

1. Open the dump file in windbg
2. Set up the symbol path and load sos (see the setup instructions for more info)

* How big is the dump? Look at the size in the file explorer.
* How does this compare to the different counters.

**Examine the .NET GC Heaps**

1. Run !eeheap -gc to examine the size of the .NET GC heaps
   * How many heaps do you have? Why?
   * How much memory is stored on the .net GC heaps? Compare to GC Heap Size
   * How much memory is on the large object heap? Hint: add up the sizes for all the Large Object heap segments. Compare to LOH Size in the perfmon counters.
2. Run !dumpheap -stat to dump out all .net objects in a statistical fashion (Note: you can run !help DumpHeap to get help for the !dumpheap command or check out [this post](https://www.tessferrandez.com/blog/2005/11/25/who-is-this-out-of-memory-guy.html) or [this post](https://www.tessferrandez.com/blog/2007/11/26/getting-started-with-windbg-part-ii.html))
   * Looking at the 5-10 object types that use most memory, what do you think is leaking?
   * What “size” does the size column show? I.e. what is included in this “size”? Hint: see the post [!dumpheap -stat explained](https://www.tessferrandez.com/blog/2005/11/25/dumpheap-stat-explained.html) for more info.

Normally I wouldn’t recommend looking at strings immediately since they will be at the bottom of the !dumpheap -stat output in pretty much every single dump you take, because of the following

* + The “size” for strings is the actual size of the contents of the string or a char[]. If you compare this to the size of a dataset for example, the size will only contain pointers to the rows and columns arrays but not include the memory for the rows and columns, so the size for a dataset object will always be very small (and constant) while the size of a string varies in size.
  + strings are leaf nodes in most objects, i.e. datasets contain strings, aspx pages contain strings, session vars contain strings, labels contain strings etc. etc. so you are bound to have mostly strings in your app.

However in this case the strings are so many and occupy so much memory that if we don’t find anything else that sticks out, we might just want to follow the string lead…

1. Dump out stats for various size char[] to find out if there is a pattern (this is a bit of trial and error so you have to try a few different sizes to figure out where the bulk of the strings are.

Get the method table (MT) for System.Char[] (first column in !dumpheap -stat)

!dumpheap -mt <char[] MT> -min 0n85000 -stat

!dumpheap -mt <char[] MT> -min 0n10000 -stat

!dumpheap -mt <char[] MT> -min 0n20000 -stat

!dumpheap -mt <char[] MT> -min 0n30000 -stat

!dumpheap -mt <char[] MT> -min 0n25000 -stat

* + In what range (between what sizes) do most of the char[] exist?

1. Dump out the strings in that range
2. !dumpheap -mt <char[] MT> -min 0n20000 -max 0n25000

In this case most of them will even be the exact same size which is a clue to what is going on

1. Dump out a few of them to find out what they contain
2. !do <address of char[] - first column in the !dumpheap -mt output>
   * What do these char[] contain?
3. Pick a few and find out where they are rooted (i.e. why they can’t be collected) Note: You may want to try a couple different ones.
4. !gcroot <address of char[]>
   * Where are they rooted? Why?

**Examine the finalizer queue and the finalizer thread**

1. Look at the finalizer queue
2. !finalizequeue
   * What objects are listed in the finalizequeue output? Hint: run !help finalizequeue
   * How many objects are “ready for finalization”? What does this mean?
3. Find the finalizer thread to determine what it is doing. Run !threads and look for the thread listed with (Finalizer)
4. Move to the finalizer thread and examine the managed and native call stack
5. ~5s # 5 should be substituted with the id of the finalizer thread
6. kb 2000
7. !clrstack
   * What object is it finalizing?
   * What is it doing? Why is this causing high memory usage?

**Examine the code for verification**

1. Open Link.cs to find the destructor/finalizer for the Link class to see the problematic code