**.NET Debugging Demos Lab 6: Memory Leak**

**Problem description**

We have started getting out of memory exceptions on the buggy bits site and we have been able to determine a scenario in which we think we are leaking memory but we can’t seem to figure out where the memory is going.

The leak seems to be occurring on our ProductInfo page for example https://localhost:44350/Products/Details/Bugspray and we can reproduce it by stress testing.

It seems like it is leaking just a small bit every time but since it is something that customers look at a lot and over time the process will crash with an out of memory exception.

**Reproduce the issue and gather data**

1. Start the application and browse to the [product details page](https://localhost:44350/Products/Details/Bugspray)
2. Monitor the .net performance counters
3. dotnet-counters monitor -n iisexpress
4. Stress the application
5. .\tinyget.ps1 -url https://localhost:44350/Products/Details/Bugspray -numTimes 1000
6. After tinyget has finished, capture a memory dump
7. dotnet-dump collect -n iisexpress

or with procdump

procdump64.exe -ma iisexpress.exe

**Review the performance counters to figure out what we are leaking**

1. Compare the Working Set to GC Heap Size
   * Do the numbers seem to match or do they diverge? Based on this, can you tell if the issue we are facing is a virtual bytes leak, a native leak or a .NET leak?
2. Look at the Number of Assemblies Loaded counter
   * Should this counter stay flat or is it ok for this counter to increase like this? What does it mean?

**Debug the memory dump**

If there is a big discrepancy between Working Set and GC Heap Size, and they don’t seem to follow each other, we either have a **native leak** which means that we have a native component that is leaking (in which case debug diag would be the next step), or we have an assembly leak.

1. Open the memory dump, load up the symbols and load sos.dll (see information and setup instructions for more info)
   * What is the size of the memory dump (on disk)?
2. Run !eeheap -gc and !dumpheap -stat
   * What is the size of the .NET heap according to !eeheap -gc, why is it different from GC Heap Size?

We saw from performance monitor that we appeared to be leaking assemblies, so the next step is to determine where these assemblies are created and why we are leaking them.

1. Run !dumpdomain to look at the assemblies loaded in the process
   * Which domain has most assemblies loaded? Note: this question makes more sense on a server where you have multiple sites running
   * Are these dynamic assemblies or assemblies loaded from disk? (is there a path associated with them)
2. Dump the module contents using !dumpmodule <moduleaddress> where module address is the address given right after Module Name on one or a few of the dynamic assemblies. eg. in the example below you would run !dumpmodule 00007ffcfe532d58
3. Assembly: 0000025392f4b4f0 (Dynamic) []
4. ClassLoader: 000002539340E710
5. Module Name
6. 00007ffcfe532d58 Dynamic Module
7. Run dc <MetaDataStart> <MetaDataEnd> to dump out the metadata for the module and find out what is implemented in this dynamic assembly. eg. in the example below you would run dc 114d09e4 114d09e4+0n4184

**Note:** We use the start address + 0n4184 because the metadata is 4148 bytes and the 0n stands for decimal

0:000> !dumpmodule 11b7e900

Name: gyq9ceq2, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

Attributes: PEFile

Assembly: 158770d0

LoaderHeap: 00000000

TypeDefToMethodTableMap: 16e2a2c4

TypeRefToMethodTableMap: 16e2a2dc

MethodDefToDescMap: 16e2a33c

FieldDefToDescMap: 16e2a3a8

MemberRefToDescMap: 16e2a3d4

FileReferencesMap: 16e2a4c4

AssemblyReferencesMap: 16e2a4c8

MetaData start address: 114d09e4 (4184 bytes)

**Note:** If your assembly does not show MetaData start address - you have to take a bit of a leap with me and find out from the raw memory where the metadata starts - the example below is for a module at the address 00007ffcfe532d58

0:000> dp 00007ffcfe532d58

00007ffc`fe532d58 00007ffd`5c570f88 00000253`93187ffa

00007ffc`fe532d68 00000253`92f4b470 00000000`00000000

00007ffc`fe532d78 00000002`00200831 00000000`00000000

00007ffc`fe532d88 00000253`92f4b4f0 00000000`26000000

00007ffc`fe532d98 ffffffff`ffffffff 00000000`ffffffff

00007ffc`fe532da8 00000000`00000000 00000000`00000000

00007ffc`fe532db8 00000000`020007d0 00000000`c0000000

00007ffc`fe532dc8 ffffffff`ffffffff 00000000`ffffffff

Take the 2nd address 0000025393187ffa - this is where the metadata is stored, and print out the contents from there and 1000 bytes on

dc 0000025393187ffa 0000025393187ffa+0n1000

* + What type of assembly was this? What is it used for? How is it generated?

**Putting it all together and determining the cause of the assembly leak**

If we look at the [documentation for XmlSerializer](https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer?redirectedfrom=MSDN&view=net-5.0) we get the following information about dynamically generated assemblies related to XmlSerialization

**Dynamically Generated Assemblies** To increase performance, the XML serialization infrastructure dynamically generates assemblies to serialize and deserialize specified types. The infrastructure finds and reuses those assemblies. This behavior occurs only when using the following constructors:

* XmlSerializer..::.XmlSerializer(Type)
* XmlSerializer..::.XmlSerializer(Type, String)

If you use any of the other constructors, multiple versions of the same assembly are generated and never unloaded, which results in a memory leak and poor performance. The easiest solution is to use one of the previously mentioned two constructors. Otherwise, you must cache the assemblies in a Hashtable…

From this, and the fact that our performance logs and dump shows that we are continuously generating new XML serialization assemblies we can conclude that it is very likely that we are not using one of the standard constructors. Search the project code for new XmlSerializer or use reflector like in this example to determine where we are generating these dynamic assemblies.

* What method / line of code is causing the problem?

**Resolve the issue and rerun the test to verify the solution**

1. Resolve the issue by caching the XmlSerializer using the sample in the documentation for XmlSerializer.
2. Rerun the test to verify that the assembly “leak” no longer exists.