Memory Leak (.NET Heap)

Sometimes the process size constantly grows but there is no difference in the process heap size. In such cases we need to check whether the process uses Microsoft .NET runtime (CLR). If one of the loaded modules is mscorwks.dll or mscorsvr.dll then it is most likely. Then we should check CLR heap statistics.

In .NET world dynamically allocated objects are garbage collected (GC) and therefore simple allocate-and-forget memory leaks are not possible. To simulate that I created the following C# program:

```
using System;

namespace CLRHeapLeak
{
    class Leak
    {
        private byte[] m_data;


        public Leak()
        {
            m_data = new byte[1024];
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Leak leak = new Leak();

            while (true)
            {
                leak = new Leak();
                System.Threading.Thread.Sleep(100);
            }
        }
    }
}
```

If we run it the process size will never grow. GC thread will collect and free unreferenced Leak classes. This can be seen from inspecting memory dumps taken with userdump.exe after the start, 2, 6 and 12 minutes. The GC heap never grows higher than 1Mb and the number of CLRHeapLeak.Leak and System.Byte[] objects always fluctuates between 100 and 500. For example, on 12th minute we have the following statistics:

```
0:000> .loadby sos mscorwks
```

```
0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x0147160c
generation 1 starts at 0x0147100c
generation 2 starts at 0x01471000
ephemeral segment allocation context: (0x014dc53c, 0x014dd618)
 segment    begin allocated     size
004aedb8 790d7ae4  790f7064 0x0001f580(128384)
01470000 01471000  014dd618 0x0006c618(443928)
Large object heap starts at 0x02471000
 segment    begin allocated     size
02470000 02471000  02473250 0x00002250(8784)
Total Size   0x8dde8(581096)
------------------------------
GC Heap Size   0x8dde8(581096)

0:000> !dumpheap -stat
total 2901 objects
Statistics:
Count    TotalSize Class Name
    1           12 System.Security.Permissions.SecurityPermission
    1           24 System.OperatingSystem
    1           24 System.Version
    1           24 System.Reflection.Assembly
    1           28 System.SharedStatics
    1           36 System.Int64[]
    1           40 System.AppDomainSetup
    3           60 System.RuntimeType
    5           60 System.Object
    2           72 System.Security.PermissionSet
    1           72 System.ExecutionEngineException
    1           72 System.StackOverflowException
    1           72 System.OutOfMemoryException
    1          100 System.AppDomain
    7          100      Free
    2          144 System.Threading.ThreadAbortException
    4          328 System.Char[]
  418         5016 CLRHeapLeak.Leak
    5         8816 System.Object[]
 2026       128632 System.String
  418       433048 System.Byte[]
Total 2901 objects
```

However, we can make Leak objects always referenced by introducing the following changes into the program:

```
using System;

namespace CLRHeapLeak
{
    class Leak
    {
        private byte[] m_data;
        private Leak m_prevLeak;
```

```
        public Leak()
        {
            m_data = new byte[1024];
        }

              public Leak(Leak prevLeak)
                              {
                       m_prevLeak = prevLeak;
                      m_data = new byte[1024];
                              }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Leak leak = new Leak();

            while (true)
            {
                leak = new Leak(leak);
                System.Threading.Thread.Sleep(100);
            }
        }
    }
}
```

Then, if we run the program, we would see in Task Manager that it grows over time. Taking consecutive memory dumps after the start, 10 and 16 minutes, shows that Win32 heap segments have always the same size:

```
0:000> !heap 0 0
Index   Address  Name      Debugging options enabled
  1:   00530000
    Segment at 00530000 to 00630000 (0003d000 bytes committed)
  2:   00010000
    Segment at 00010000 to 00020000 (00003000 bytes committed)
  3:   00520000
    Segment at 00520000 to 00530000 (00003000 bytes committed)
  4:   00b10000
    Segment at 00b10000 to 00b50000 (00001000 bytes committed)
  5:   001a0000
    Segment at 001a0000 to 001b0000 (00003000 bytes committed)
  6:   00170000
    Segment at 00170000 to 00180000 (00008000 bytes committed)
  7:   013b0000
    Segment at 013b0000 to 013c0000 (00003000 bytes committed)
```

but GC heap and the number of Leak and System.Byte[] objects in it were growing significantly:

```
Process Uptime: 0 days 0:00:04.000

0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x013c1018
```

```
generation 1 starts at 0x013c100c
generation 2 starts at 0x013c1000
ephemeral segment allocation context: (0x013cd804, 0x013cdff4)
 segment    begin allocated     size
0055ee08 790d7ae4  790f7064 0x0001f580(128384)
013c0000 013c1000  013cdff4 0x0000cff4(53236)
Large object heap starts at 0x023c1000
 segment    begin allocated     size
023c0000 023c1000  023c3250 0x00002250(8784)
Total Size   0x2e7c4(190404)
------------------------------
GC Heap Size   0x2e7c4(190404)

0:000> !dumpheap -stat
total 2176 objects
Statistics:
Count    TotalSize Class Name
...
...
...
   46          736 CLRHeapLeak.Leak
    5         8816 System.Object[]
   46        47656 System.Byte[]
 2035       129604 System.String
Total 2176 objects

Process Uptime: 0 days 0:09:56.000

0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x018cddbc
generation 1 starts at 0x01541ec4
generation 2 starts at 0x013c1000
ephemeral segment allocation context: (0x0192d668, 0x0192ddc8)
 segment    begin allocated     size
0055ee08 790d7ae4  790f7064 0x0001f580(128384)
013c0000 013c1000  0192ddc8 0x0056cdc8(5688776)
Large object heap starts at 0x023c1000
 segment    begin allocated     size
023c0000 023c1000  023c3240 0x00002240(8768)
Total Size  0x58e588(5825928)
------------------------------
GC Heap Size  0x58e588(5825928)

0:000> !dumpheap -stat
total 12887 objects
Statistics:
Count    TotalSize Class Name
...
...
...
    5         8816 System.Object[]
 5403         86448 CLRHeapLeak.Leak
```

```
 2026       128632 System.String
 5403       5597508 System.Byte[]
Total 12887 objects

Process Uptime: 0 days 0:16:33.000

0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01c59cb4
generation 1 starts at 0x0194fd20
generation 2 starts at 0x013c1000
ephemeral segment allocation context: (0x01cd3050, 0x01cd3cc0)
 segment    begin allocated     size
0055ee08 790d7ae4  790f7064 0x0001f580(128384)
013c0000 013c1000  01cd3cc0 0x00912cc0(9514176)
Large object heap starts at 0x023c1000
 segment    begin allocated     size
023c0000 023c1000  023c3240 0x00002240(8768)
Total Size  0x934480(9651328)
------------------------------
GC Heap Size  0x934480(9651328)

0:000> !dumpheap -stat
total 20164 objects
Statistics:
Count    TotalSize Class Name
    5         8816 System.Object[]
 2026       128632 System.String
 9038        144608 CLRHeapLeak.Leak
   9038       9363368 System.Byte[]
Total 20164 objects
```

This is not the traditional memory leak because we have the reference chain. However, uncontrolled memory growth can be considered as a memory leak too, caused by poor application design, bad input validation or error handling, etc.

There are situations when customers think there is a memory leak but it is not. One of them is unusually big size of a process when running it on a multi-processor server. If dllhost.exe hosting typical .NET assembly DLL occupies less than 100Mb on a local workstation starts consuming more than 300Mb on a 4 processor server than it can be the case that the server version of CLR uses per processor GC heaps:

```
0:000> .loadby sos mscorsvr

0:000> !EEHeap -gc
generation 0 starts at 0x05c80154
generation 1 starts at 0x05c7720c
generation 2 starts at 0x102d0030
generation 0 starts at 0x179a0444
generation 1 starts at 0x1799b7a4
generation 2 starts at 0x142d0030
generation 0 starts at 0x0999ac88
generation 1 starts at 0x09990cc4
generation 2 starts at 0x182d0030
```

```
generation 0 starts at 0x242eccb0
generation 1 starts at 0x242d0030
generation 2 starts at 0x1c2d0030
...
...
...
GC Heap Size  0x109702ec(278332140)
```

or if this is CLR 1.x the old extension will tell us the same too:

```
0:000> !.\clr10\sos.eeheap -gc
Loaded Son of Strike data table version 5 from
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\mscorsvr.dll"
Number of GC Heaps: 4
------------------------------
Heap 0 (0x000f9af0)
generation 0 starts at 0x05c80154
generation 1 starts at 0x05c7720c
generation 2 starts at 0x102d0030
...
...
...
Heap Size  0x515ed60(85,323,104)
------------------------------
Heap 1 (0x000fa070)
generation 0 starts at 0x179a0444
generation 1 starts at 0x1799b7a4
generation 2 starts at 0x142d0030
...
...
...
Heap Size  0x37c7bf0(58,489,840)
------------------------------
Heap 2 (0x000fab80)
generation 0 starts at 0x0999ac88
generation 1 starts at 0x09990cc4
generation 2 starts at 0x182d0030
...
...
...
Heap Size  0x485de34(75,882,036)
------------------------------
Heap 3 (0x000fb448)
generation 0 starts at 0x242eccb0
generation 1 starts at 0x242d0030
generation 2 starts at 0x1c2d0030
...
...
...
Heap Size  0x41ea570(69,117,296)
------------------------------
Reserved segments:
------------------------------
GC Heap Size  0x1136ecf4(288,812,276)
```

The more processors we have the more heaps are contributing to the overall VM size. Although the process occupies almost 400Mb if it doesn't grow constantly over time beyond that value then it is normal.