Memory leaks

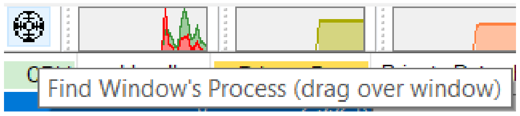
This lab focuses on discovering memory leaks.

In case of unexpected memory consumption, the first step is usually to ensure that the application is not simply consuming a lot of memory by design (large caches, server mode GC,…). Then, the application memory should be compared from time to time in order to see growing count of types instances. When unexpected instances are detected, the final step is to find out what is the “root” object that keeps a reference (possibly transitively) to these instances.

Reproducing the problem

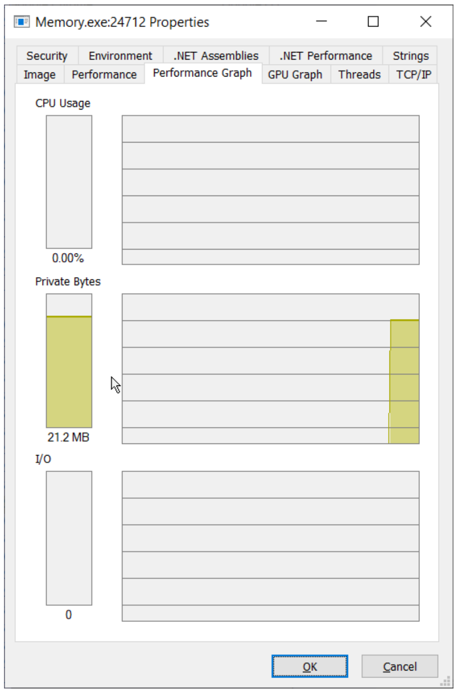
The Memory application allows you to open Mirror windows where the content of a text box in the main window appears with the letters in the opposite order. Let’s see if opening several Mirror windows leaks memory.

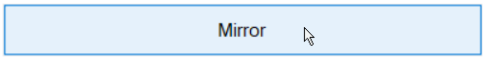
1. Make **Memory** as the startup project and run it (don’t debug it)

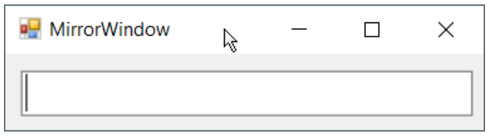
2. Open Process Explorer and select the **Memory** application
   Note: for UI applications, it might be easier to find the application by dragging the target toolbar button
   

   and dropping it into the Memory application window

3. Double-click the process line and look at the **Private Bytes** graph

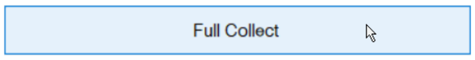


4. Click the  button and a new Mirror window should appear



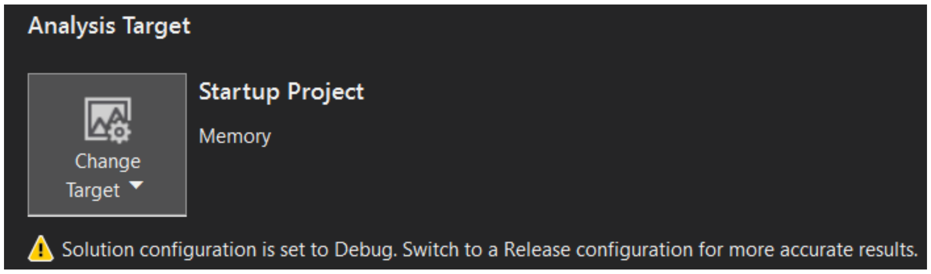
5. Check what is the value of the Private Bytes in Process Explorer: is there any increase? Is this a sign of memory leak?

6. Close the Mirror window and check the value of the Private Bytes in Process Explorer Is this a sign of memory leak?

7. Press the  button and check the value of Private Bytes in Process Explorer: is this a sign of memory leak?

Investigating with Visual Studio

1. Make **Memory** as the startup project and start a profiler session with Debug | Performance Profiler
Note: it is recommended to use a Release build (otherwise the following warning will be displayed)

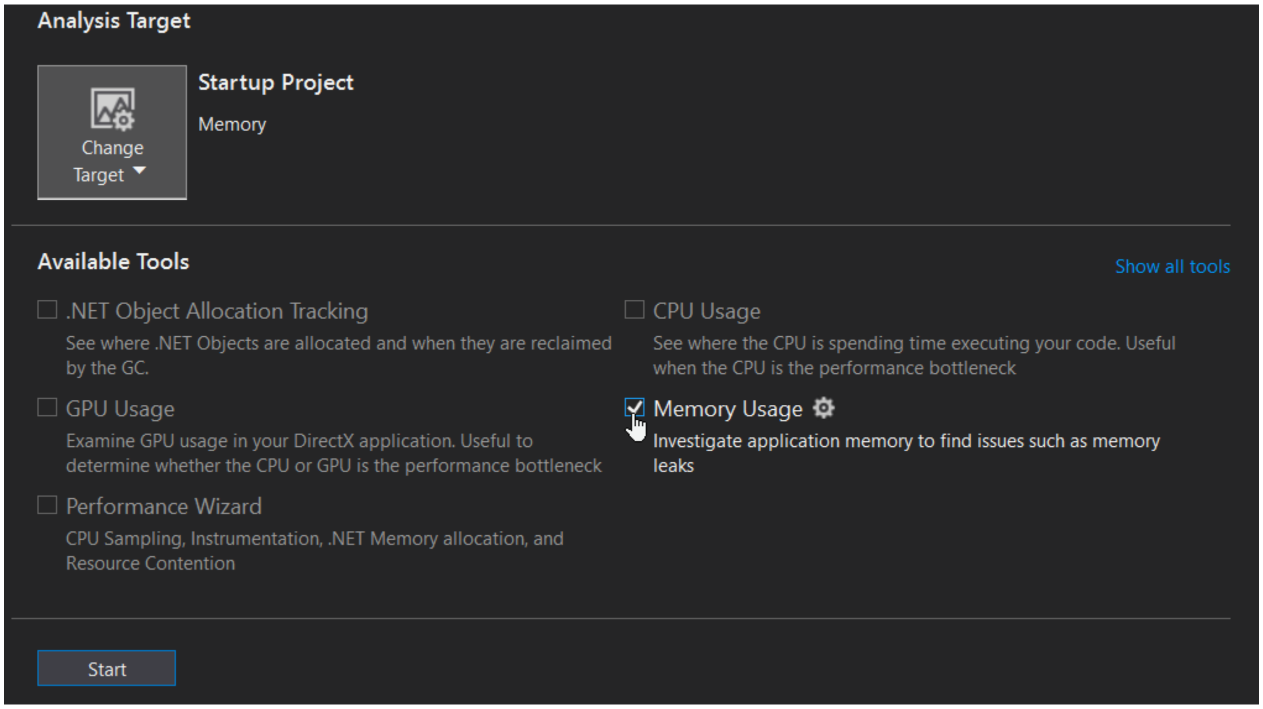


Startup Project

Memory

Solution configuration is set to Debug. Switch to a Release configuration for more accurate results.

2. Check the **Memory Usage** box



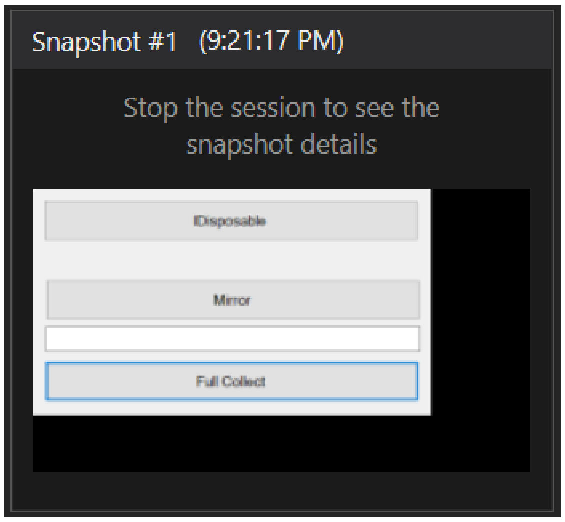
Startup Project

Memory

- [ ] .NET Object Allocation Tracking
  See where .NET Objects are allocated and when they are reclaimed by the GC.
- [ ] CPU Usage
  See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck
- [ ] GPU Usage
  Examine GPU usage in your DirectX application. Useful to determine whether the CPU or GPU is the performance bottleneck
- [✓] Memory Usage
  Investigate application memory to find issues such as memory leaks
- [ ] Performance Wizard
  CPU Sampling, Instrumentation, .NET Memory allocation, and Resource Contention

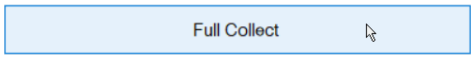
and click the **Start** button.

3. Click the **Take Snaphot** button to create a baseline

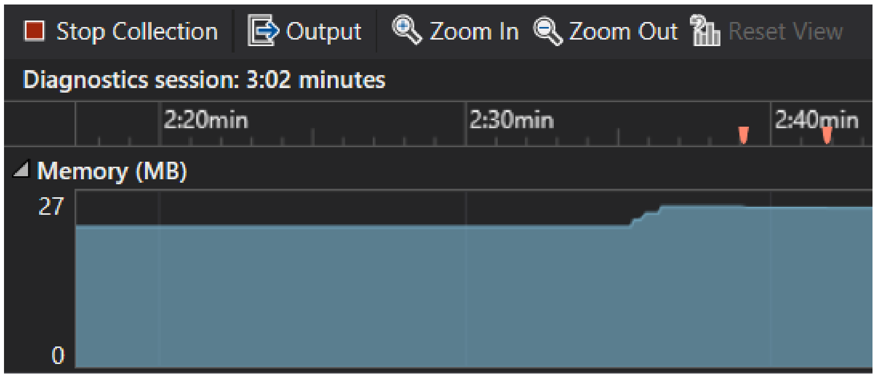


Snapshot #1 (9:21:17 PM)

Stop the session to see the snapshot details

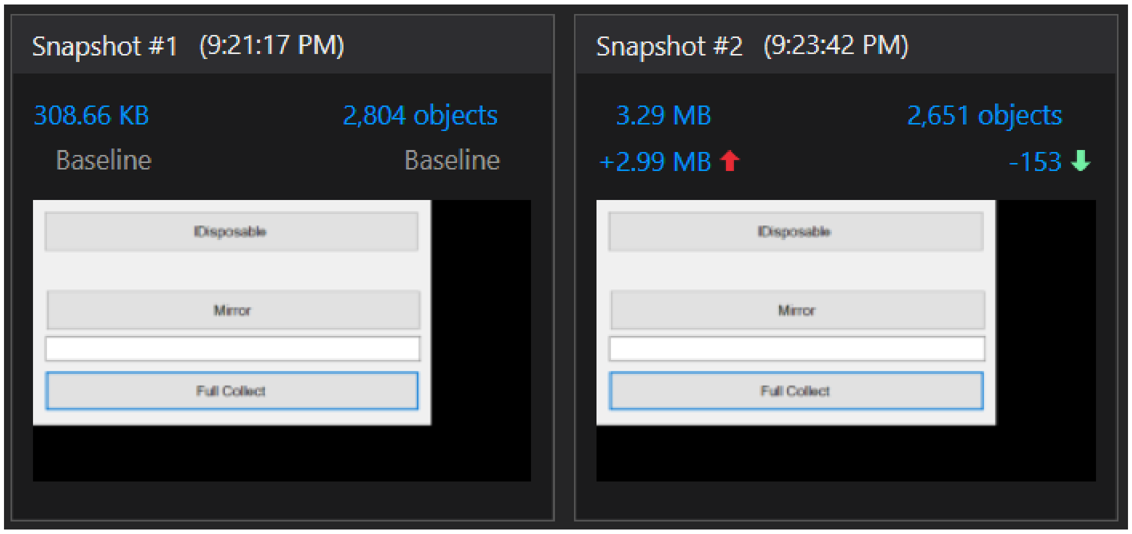
4. Create 3 Mirror windows, close them and press the 

button.
Note: try to reproduce the leaking workflow 3 or 5 times because it will be easier to find instances count patterns of multiples of 3 or 5.

5. Click the **Take Snapshot** again to get the state of the application after the 3 windows creation


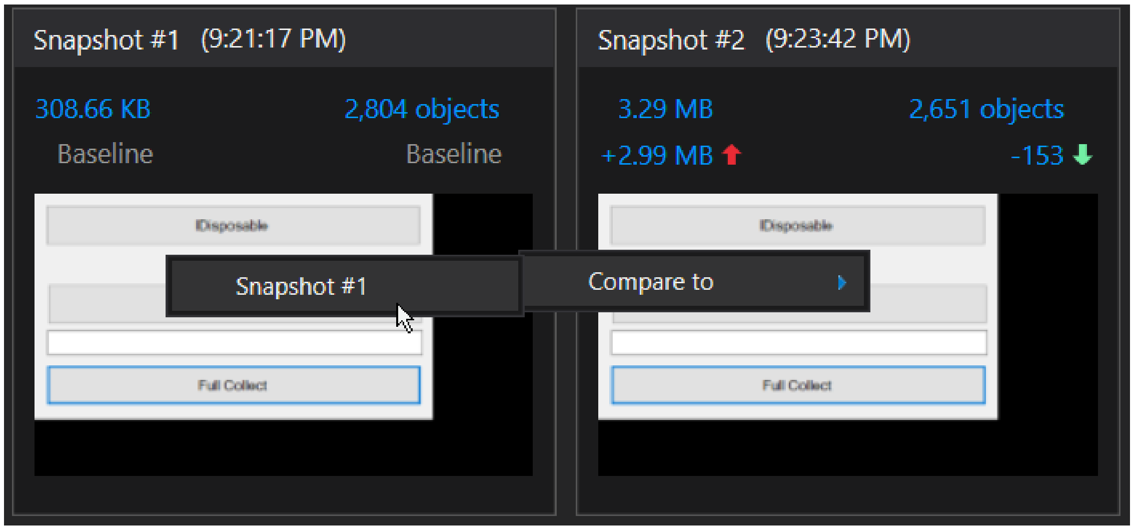
Note: you can see the 3 small bumps in the memory consumption corresponding to the Mirror windows. Also two red triangles are visible at the right of the timeline: each represents a garbage collector event. The first one is triggered by the Full Collect button and the second one by the profiler to ensure that the objects in memory are all referenced.

6. Click the

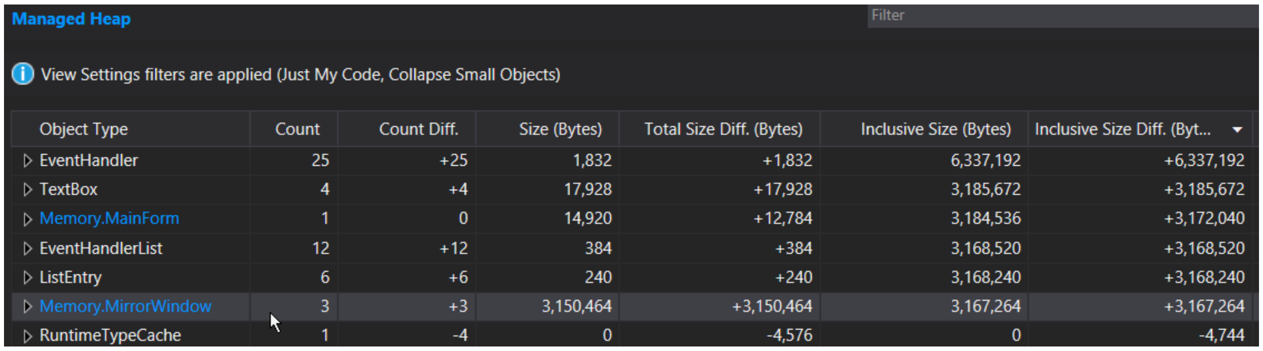
button and you should see 2 snapshots


Snapshot #1 (9:21:17 PM)		Snapshot #2 (9:23:42 PM)	
308.66 KB	2,804 objects	3.29 MB	2,651 objects
Baseline	Baseline	+2.99 MB ⬆	-153 ⬇

7. Right click the second one and select the Snapshot #1



8. The new **Managed Heap** panel shows count and size of types instances



View Settings filters are applied (Just My Code, Collapse Small Objects)

Object Type	Count	Count Diff.	Size (Bytes)	Total Size Diff. (Bytes)	Inclusive Size (Bytes)	Inclusive Size Diff. (Byt...
EventHandler	25	+25	1,832	+1,832	6,337,192	+6,337,192
TextBox	4	+4	17,928	+17,928	3,185,672	+3,185,672
Memory.MainForm	1	0	14,920	+12,784	3,184,536	+3,172,040
EventHandlerList	12	+12	384	+384	3,168,520	+3,168,520
ListEntry	6	+6	240	+240	3,168,240	+3,168,240
Memory.MirrorWindow	3	+3	3,150,464	+3,150,464	3,167,264	+3,167,264
RuntimeTypeCache	1	-4	0	-4,576	0	-4,744

In the **xxxDiff** columns:

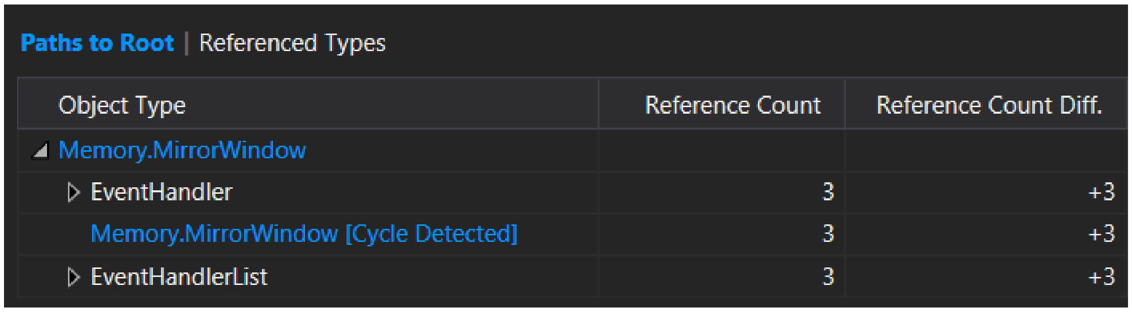
- <number> : less than the baseline

+ <number> : more than the baseline

Is it expected to see +3 for Memory.MirrorWindow in **Count Diff**?

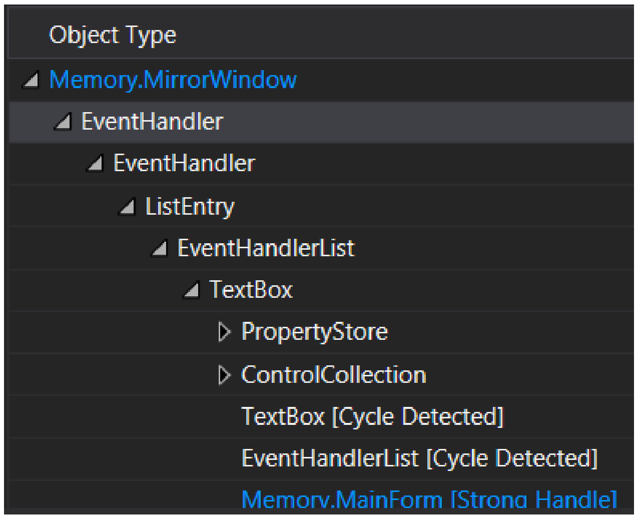
Is it expected to see +3,150,464 for Memory.MirrorWindow in **Total Size Diff (Bytes)**?

9. Click the **Memory.MirrorWindow** line and look at the pane below



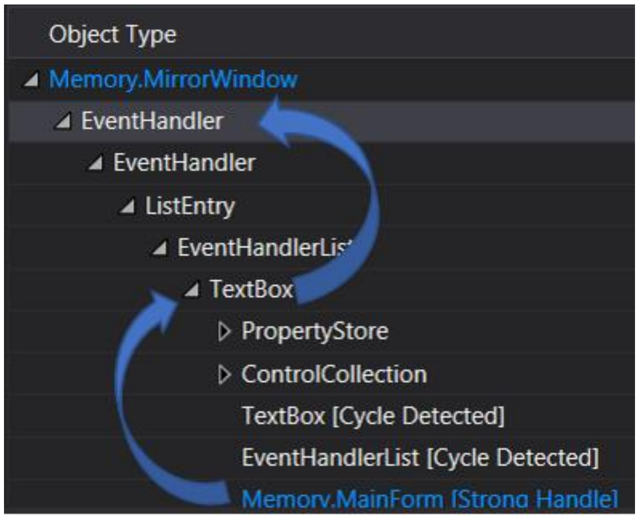
Object Type	Reference Count	Reference Count Diff.
Memory.MirrorWindow		
EventHandler	3	+3
Memory.MirrorWindow [Cycle Detected]	3	+3
EventHandlerList	3	+3

This view is misleading so expand the ▷ EventHandler at the top to see the whole reference tree



```
Object Type
◢ Memory.MirrorWindow
  ◢ EventHandler
    ◢ EventHandler
      ◢ ListEntry
        ◢ EventHandlerList
          ◢ TextBox
            ▷ PropertyStore
            ▷ ControlCollection
              TextBox [Cycle Detected]
              EventHandlerList [Cycle Detected]
              Memory.MainForm [Strong Handle]
```

10. The references should be read from the bottom up to the top



```
Object Type
◢ Memory.MirrorWindow
  ◢ EventHandler
    ◢ EventHandler
      ◢ ListEntry
        ◢ EventHandlerList
          ◢ TextBox
            ▷ PropertyStore
            ▷ ControlCollection
              TextBox [Cycle Detected]
              EventHandlerList [Cycle Detected]
              Memory.MainForm [Strong Handle]
```

The **TextBox** of the **MainForm** is referencing the **MirrorWindow** through an event.
What is the reason of this reference?
How to fix the problem?

Post-mortem analysis with Visual Studio Enterprise

If Visual Studio cannot be used (secured server environment or customer machines for example), it is still possible to troubleshoot this kind of problem with the following steps:

1. Save the application memory to dump files as memory consumption grows over time
2. Open the dump files in Visual Studio Enterprise
3. Compare the memory content

Note: this feature is only available in Enterprise Edition of Visual Studio

Dumping application memory

Even though it is possible to create a memory dump with Task Manager, it is recommended to use **procdump** from SysInternals.

1. Download **procdump** from https://docs.microsoft.com/en-us/sysinternals/downloads/procdump

2. Start a Prompt, create and move to a **dumps** folder in your hard drive

3. When the application is in a stable state, type the following command line:
   procdump -ma <process ID>
   Note: the fist time Sysinternals tools are started, it is required to accept the EULA form.

   *Note: by default, a .dmp file will be created with a name following the pattern **<process name>_<date>_<time>.dmp** but you can pass a filename as last parameter if you prefer.*

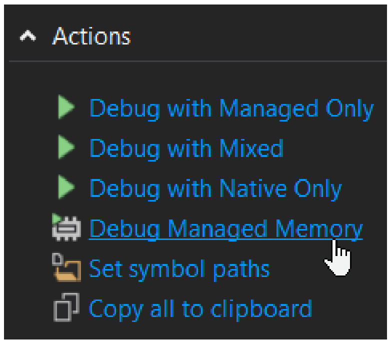
4. Do the same after creating and closing 3 Mirror windows.
   Note: don't forget to press the Full Collect button in order to only have referenced objects in the memory dump.

Opening a dump with Visual Studio

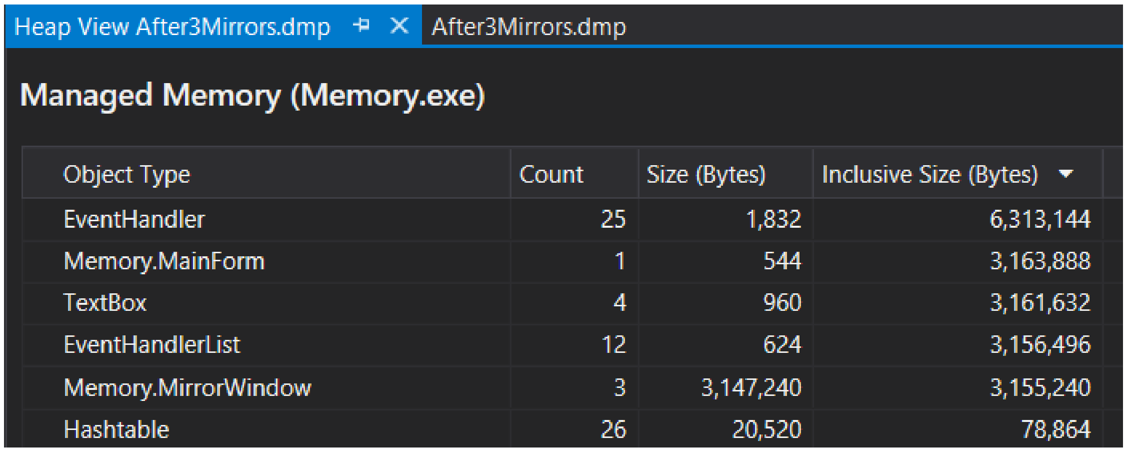
Visual Studio is able to open a memory dump and provides a few debugging services helping troubleshoot hang issues.

1. Select the .dmp file with 3 Mirror windows from **File | Open | File…** or drag and drop it on Visual Studio title bar from Windows Explorer

2. In the **Minidump File Summary** window, click the **Debug Managed Memory** link in the upper right **Actions** section



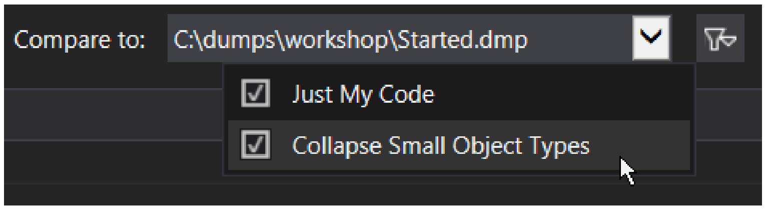
3. A new panel appears with the count and size of type instances



Managed Memory (Memory.exe)

Object Type	Count	Size (Bytes)	Inclusive Size (Bytes)
EventHandler	25	1,832	6,313,144
Memory.MainForm	1	544	3,163,888
TextBox	4	960	3,161,632
EventHandlerList	12	624	3,156,496
Memory.MirrorWindow	3	3,147,240	3,155,240
Hashtable	26	20,520	78,864

Note: don't forget to check these two options for a more easier to read list

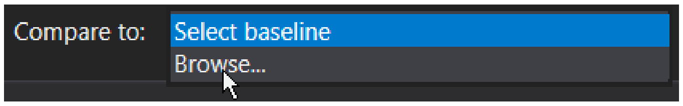


Compare to: C:\dumps\workshop\Started.dmp

☑ Just My Code

☑ Collapse Small Object Types

4. The next step is to compare with the baseline chosen via **Browse…** in



Look for the application types such as **Memory.MirrorWindow**: what do you see in the **Count Diff.** column?

5. Click the type with a positive count diff and look at the **Paths to Root** lower pane
   What could be the reason for the unexpected reference?
   How to fix the problem?

Post-mortem analysis with WinDBG

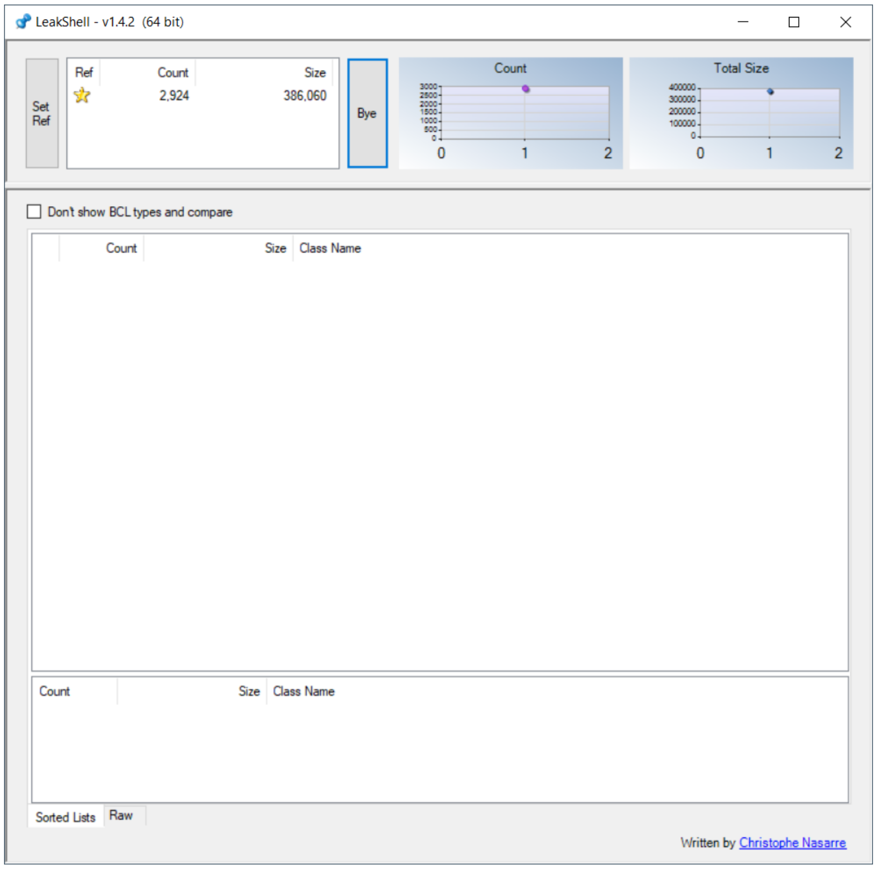
In case of large memory dumps, it is possible that Visual Studio fails to load. In that case, it is still possible to troubleshoot memory leaks in a more "manual" way with the following steps:

1. Download LeakShell from https://1drv.ms/u/s!Ahfhqceam4crke1VJNaiKtDkFG9sYA
2. Start LeakShell. This companion tool listens to Windows clipboard events and will compute diffs based on sos command results run in WinDBG
3. Save the application memory to dump files as a reference and after 3 Mirror windows have been created
4. Open the reference dump file in WinDBG
   a. Load sos extension: **.loadby sos clr**
   b. Clear the screen: **.cls**

c. List the type instances count: **!dumpheap -stat**

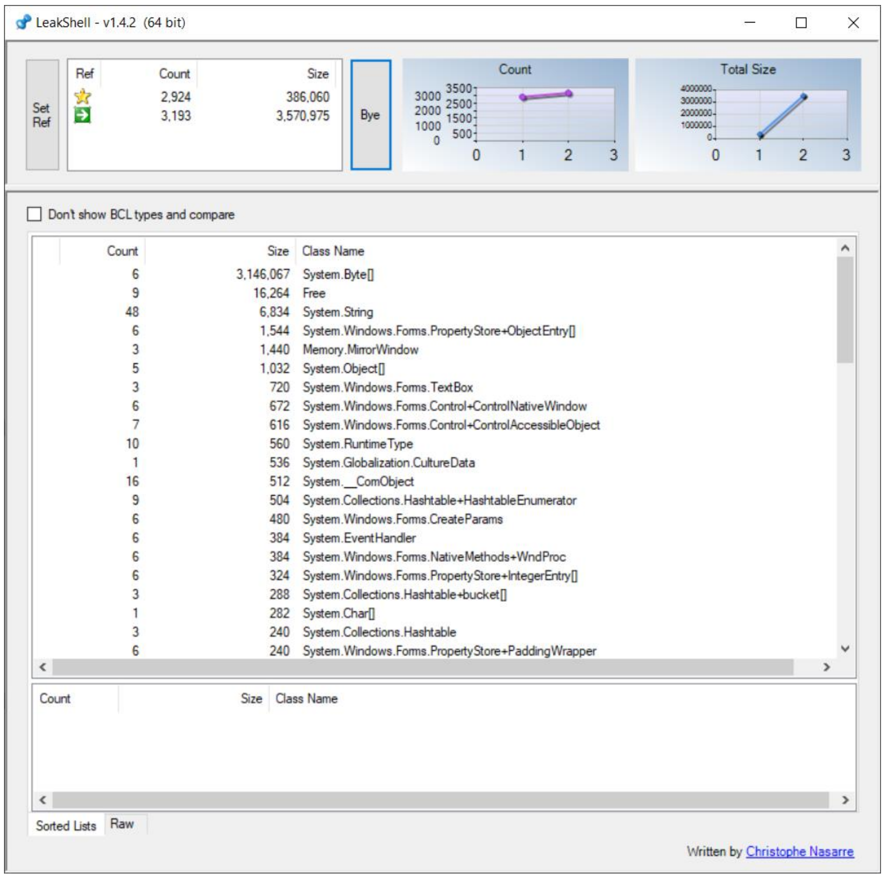
d. Select the whole output with **CTRL+A** and copy it to the clipboard with **CTRL+C**

e. You should see the UI of LeakShell change with a new reference snapshot


LeakShell - v1.4.2 (64 bit)
Set Ref
Bye
Count
Total Size
Don't show BCL types and compare
Sorted Lists
Raw
Written by Christophe Nasarre

Ref	Count	Size
☆	2,924	386,060

5. Open the 3 Mirror windows dump in another WinDBG
    a. Load sos extension: **.loadby sos clr**
    b. Clear the screen: **.cls**
    c. List the type instances count: **!dumpheap -stat**
    d. Select the whole output with **CTRL+A** and copy it to the clipboard with **CTRL+C**
    e. LeakShell should now show the 2 snapshots and the diff between them


LeakShell - v1.4.2 (64 bit)

Set Ref

Ref	Count	Size
☆	2,924	386,060
➔	3,193	3,570,975

Bye

☐ Don't show BCL types and compare

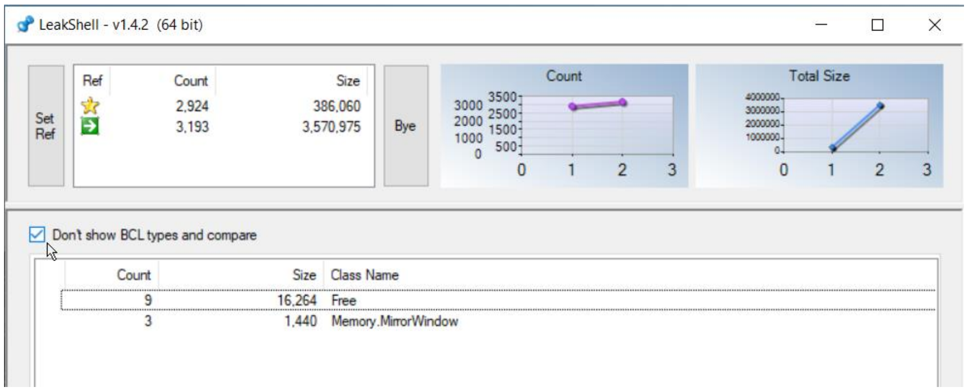
Count	Size	Class Name
6	3,146,067	System.Byte[]
9	16,264	Free
48	6,834	System.String
6	1,544	System.Windows.Forms.PropertyStore+ObjectEntry[]
3	1,440	Memory.MirrorWindow
5	1,032	System.Object[]
3	720	System.Windows.Forms.TextBox
6	672	System.Windows.Forms.Control+ControlNativeWindow
7	616	System.Windows.Forms.Control+ControlAccessibleObject
10	560	System.RuntimeType
1	536	System.Globalization.CultureData
16	512	System.__ComObject
9	504	System.Collections.Hashtable+HashtableEnumerator
6	480	System.Windows.Forms.CreateParams
6	384	System.EventHandler
6	384	System.Windows.Forms.NativeMethods+WndProc
6	324	System.Windows.Forms.PropertyStore+IntegerEntry[]
3	288	System.Collections.Hashtable+bucket[]
1	282	System.Char[]
3	240	System.Collections.Hashtable
6	240	System.Windows.Forms.PropertyStore+PaddingWrapper

Count	Size	Class Name

Sorted Lists | Raw

Written by Christophe Nasarre

6. For a first high level view of non Microsoft types, check the **Don't show BCL types and compare** box


LeakShell - v1.4.2 (64 bit)

Set Ref

Ref	Count	Size
☆	2,924	386,060
➔	3,193	3,570,975

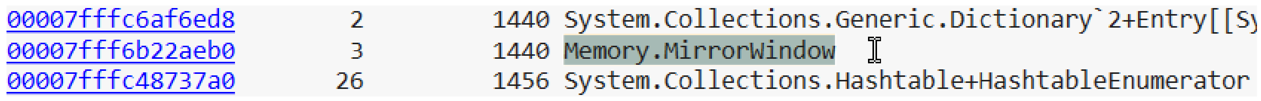
Bye

☑ Don't show BCL types and compare

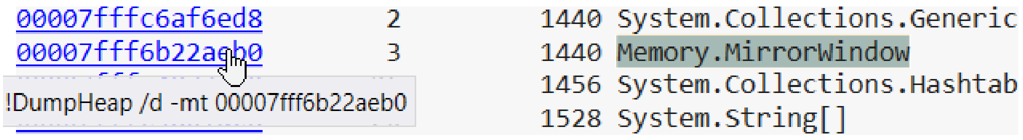
Count	Size	Class Name
9	16,264	Free
3	1,440	Memory.MirrorWindow

The next step is to find what is referencing these instances of Memory.MirrorWindow

1. Go back to the WinDBG with the 3 MirrorWindow dump
2. Look for the instances of Memory.MirrorWindow with a simple CTRL+F

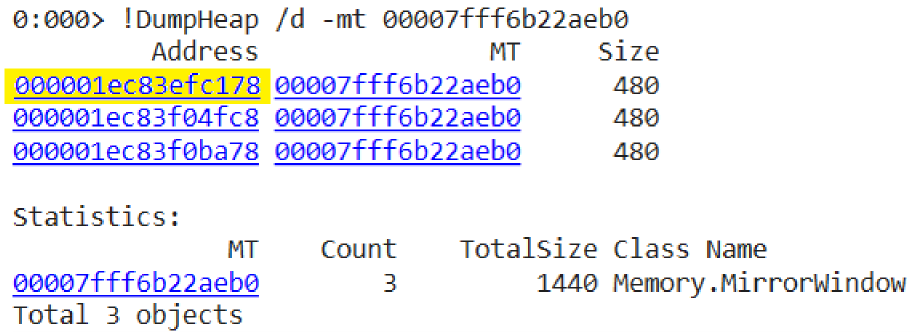

```
00007fffc6af6ed8        2          1440 System.Collections.Generic.Dictionary`2+Entry[[Sy
00007fff6b22aeb0        3          1440 Memory.MirrorWindow
00007fffc48737a0       26          1456 System.Collections.Hashtable+HashtableEnumerator
```

3. Click the link on the left to list all instances



```
00007fffc6af6ed8          2          1440 System.Collections.Generic
00007fff6b22aeb0          3          1440 Memory.MirrorWindow
                                     1456 System.Collections.Hashtab
!DumpHeap /d -mt 00007fff6b22aeb0    1528 System.String[]
```

4. The question to answer now is how to find what is the root of these instances



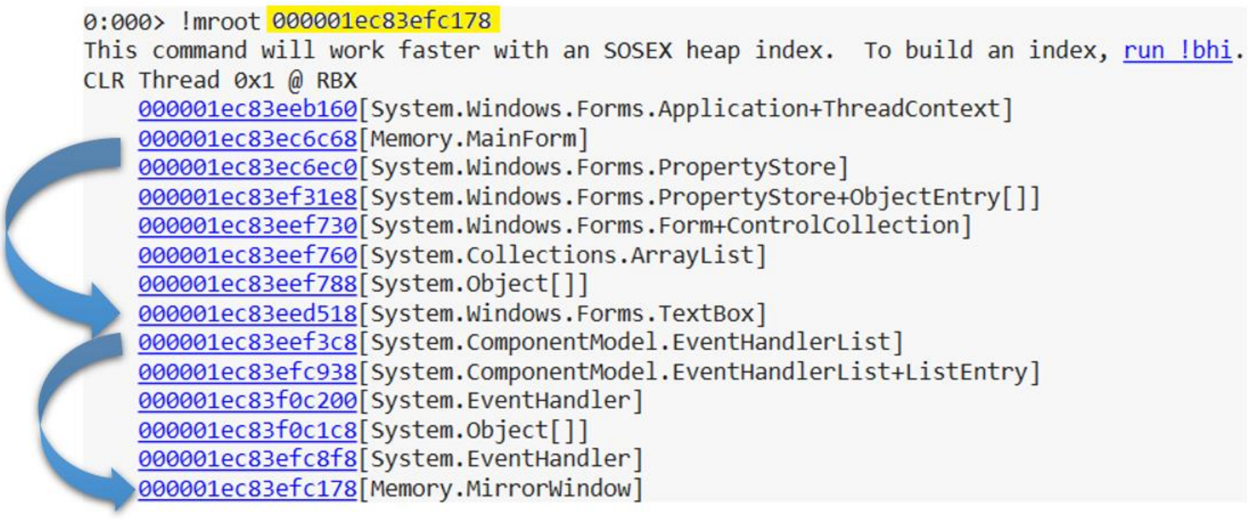
```
0:000> !DumpHeap /d -mt 00007fff6b22aeb0
         Address               MT     Size
000001ec83efc178 00007fff6b22aeb0      480
000001ec83f04fc8 00007fff6b22aeb0      480
000001ec83f0ba78 00007fff6b22aeb0      480

Statistics:
              MT    Count    TotalSize Class Name
00007fff6b22aeb0        3         1440 Memory.MirrorWindow
Total 3 objects
```

The answer is provided by the **mroot** command of the **sosex** extension

5. Load the sosex extension: **.load <path to the sosex.dll>\sosex**

6. Pick a reference to a MirrorWindow instance to call: **!mroot <address>**



```
0:000> !mroot 000001ec83efc178
This command will work faster with an SOSEX heap index.  To build an index, run !bhi.
CLR Thread 0x1 @ RBX
    000001ec83eeb160[System.Windows.Forms.Application+ThreadContext]
    000001ec83ec6c68[Memory.MainForm]
    000001ec83ec6ec0[System.Windows.Forms.PropertyStore]
    000001ec83ef31e8[System.Windows.Forms.PropertyStore+ObjectEntry[]]
    000001ec83eef730[System.Windows.Forms.Form+ControlCollection]
    000001ec83eef760[System.Collections.ArrayList]
    000001ec83eef788[System.Object[]]
    000001ec83eed518[System.Windows.Forms.TextBox]
    000001ec83eef3c8[System.ComponentModel.EventHandlerList]
    000001ec83efc938[System.ComponentModel.EventHandlerList+ListEntry]
    000001ec83f0c200[System.EventHandler]
    000001ec83f0c1c8[System.Object[]]
    000001ec83efc8f8[System.EventHandler]
    000001ec83efc178[Memory.MirrorWindow]
```

7. The same relationship between the root MainForm and the MirrorWindow via an event handler listening to a TextBox

*Note: things might be more complicated in term of references and you might want to see ALL reference paths ending up to an instance. In that case, add **-all** to the **mroot** command.*

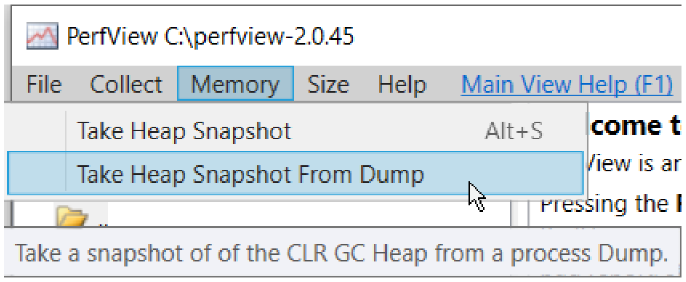
Memory analysis with Perfview

A last but not least option to analyze memory issues is provided by Perfview. This free tool is a Microsoft open source project from https://github.com/microsoft/perfview and allows you, among much more other things, to search for memory leaks.

1. Save the application memory to dump files as memory consumption grows over time
2. In Perfview, create heap snapshots from the dump files
3. Compare snapshots and find references root

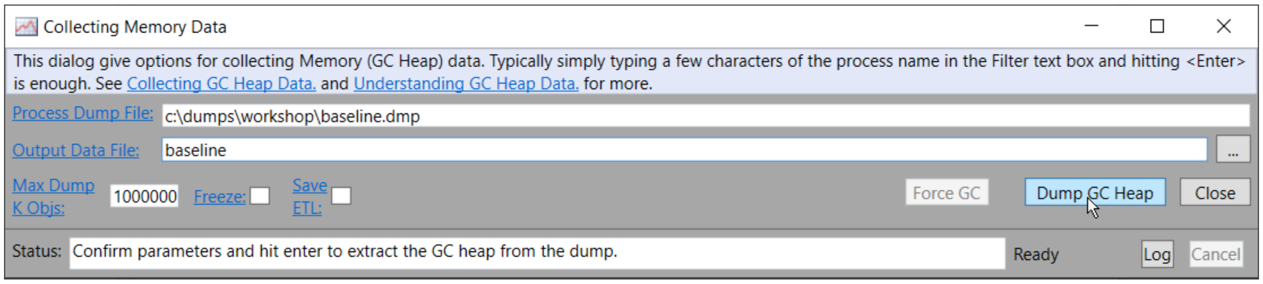
Note: this walkthrough will be using memory dumps (see previous section for dump file generation) but you could do the same with a live .NET process on production machine.

1. In Perfview, select **Memory** | **Take Heap Snapshot from Dump**



Take a snapshot of of the CLR GC Heap from a process Dump.

2. Enter the path to the baseline dump file and use **baseline.gcheap** as output data file name. In addition, in order to avoid object sampling, enter 1000000 in the **Max Dump Obj K Objs** textbox



This dialog give options for collecting Memory (GC Heap) data. Typically simply typing a few characters of the process name in the Filter text box and hitting <Enter> is enough. See Collecting GC Heap Data. and Understanding GC Heap Data. for more.

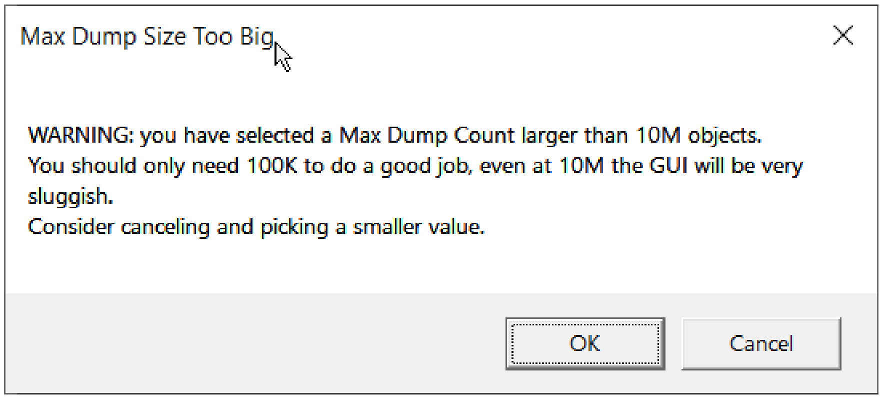
Process Dump File: c:\dumps\workshop\baseline.dmp

Output Data File: baseline

Max Dump K Objs: 1000000 Freeze: ☐ Save ETL: ☐

Status: Confirm parameters and hit enter to extract the GC heap from the dump. Ready

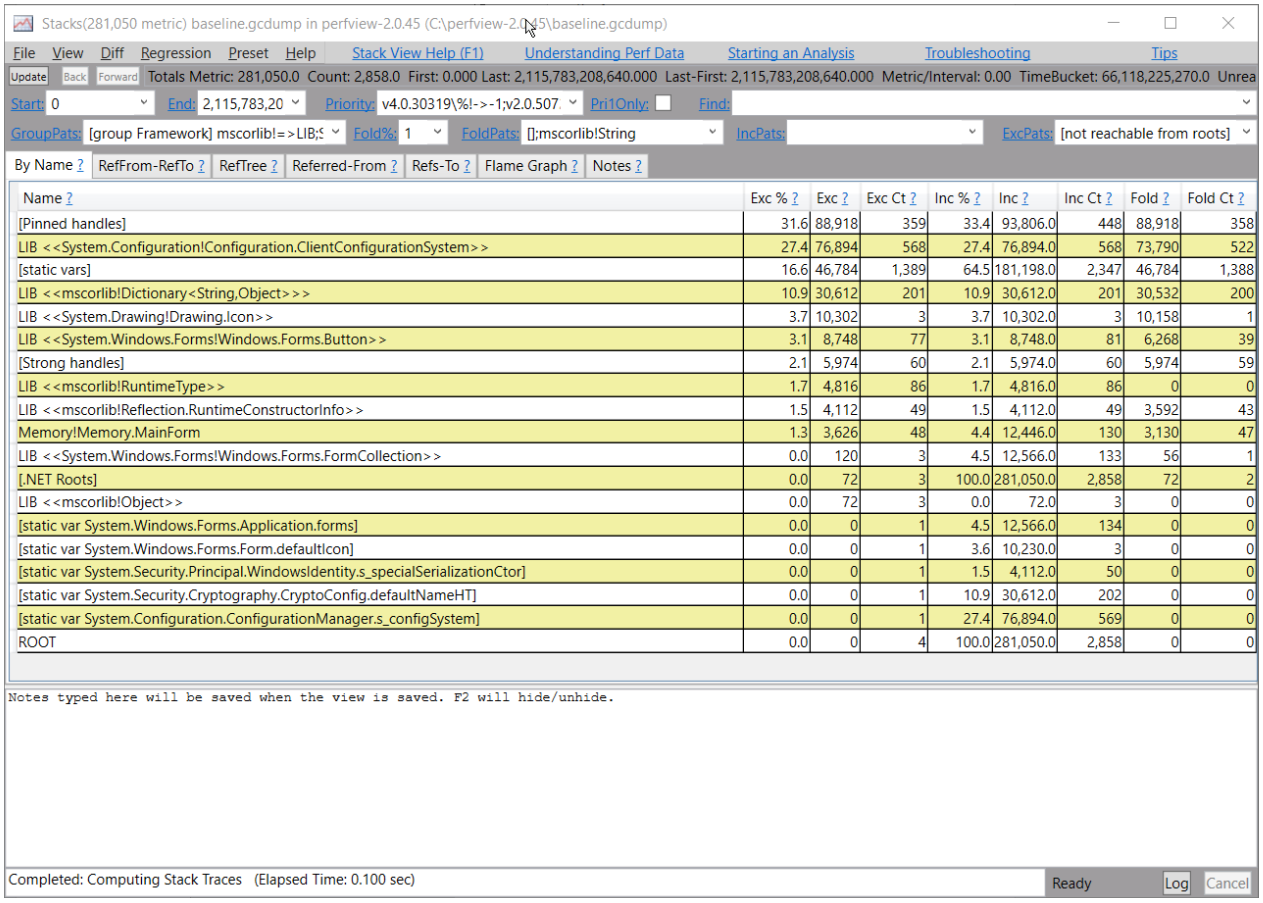
and press the **Dump GC Heap** button.
Dismiss the warning dialog box related to sampling



WARNING: you have selected a Max Dump Count larger than 10M objects.
You should only need 100K to do a good job, even at 10M the GUI will be very sluggish.
Consider canceling and picking a smaller value.

Note: you might need sampling in case of very large dump heaps but it could hide objects with very few instances.

3. Now a new **Stacks** window should have popped up



Stacks(281,050 metric) baseline.gcdump in perfview-2.0.45 (C:\perfview-2.0.45\baseline.gcdump)

Totals Metric: 281,050.0 Count: 2,858.0 First: 0.000 Last: 2,115,783,208,640.000 Last-First: 2,115,783,208,640.000 Metric/Interval: 0.00 TimeBucket: 66,118,225,270.0 Unrea

Start: 0 End: 2,115,783,20 Priority: v4.0.30319\%!->-1;v2.0.507 Pri1Only: Find:

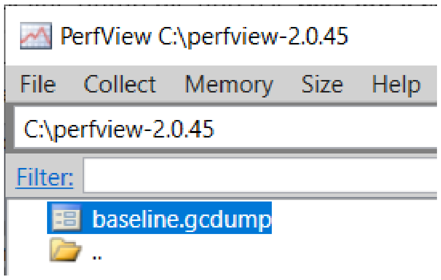
GroupPats: [group Framework] mscorlib!=>LIB;S Fold%: 1 FoldPats: [];mscorlib!String IncPats: ExcPats: [not reachable from roots]

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?	Fold ?	Fold Ct ?
[Pinned handles]	31.6	88,918	359	33.4	93,806.0	448	88,918	358
LIB <<System.Configuration!Configuration.ClientConfigurationSystem>>	27.4	76,894	568	27.4	76,894.0	568	73,790	522
[static vars]	16.6	46,784	1,389	64.5	181,198.0	2,347	46,784	1,388
LIB <<mscorlib!Dictionary<String,Object>>>	10.9	30,612	201	10.9	30,612.0	201	30,532	200
LIB <<System.Drawing!Drawing.Icon>>	3.7	10,302	3	3.7	10,302.0	3	10,158	1
LIB <<System.Windows.Forms!Windows.Forms.Button>>	3.1	8,748	77	3.1	8,748.0	81	6,268	39
[Strong handles]	2.1	5,974	60	2.1	5,974.0	60	5,974	59
LIB <<mscorlib!RuntimeType>>	1.7	4,816	86	1.7	4,816.0	86	0	0
LIB <<mscorlib!Reflection.RuntimeConstructorInfo>>	1.5	4,112	49	1.5	4,112.0	49	3,592	43
Memory!Memory.MainForm	1.3	3,626	48	4.4	12,446.0	130	3,130	47
LIB <<System.Windows.Forms!Windows.Forms.FormCollection>>	0.0	120	3	4.5	12,566.0	133	56	1
[.NET Roots]	0.0	72	3	100.0	281,050.0	2,858	72	2
LIB <<mscorlib!Object>>	0.0	72	3	0.0	72.0	3	0	0
[static var System.Windows.Forms.Application.forms]	0.0	0	1	4.5	12,566.0	134	0	0
[static var System.Windows.Forms.Form.defaultIcon]	0.0	0	1	3.6	10,230.0	3	0	0
[static var System.Security.Principal.WindowsIdentity.s_specialSerializationCtor]	0.0	0	1	1.5	4,112.0	50	0	0
[static var System.Security.Cryptography.CryptoConfig.defaultNameHT]	0.0	0	1	10.9	30,612.0	202	0	0
[static var System.Configuration.ConfigurationManager.s_configSystem]	0.0	0	1	27.4	76,894.0	569	0	0
ROOT	0.0	0	4	100.0	281,050.0	2,858	0	0

```
Notes typed here will be saved when the view is saved. F2 will hide/unhide.
```

Completed: Computing Stack Traces (Elapsed Time: 0.100 sec)

and in Perfview, you should see the **baseline.gcdump** file

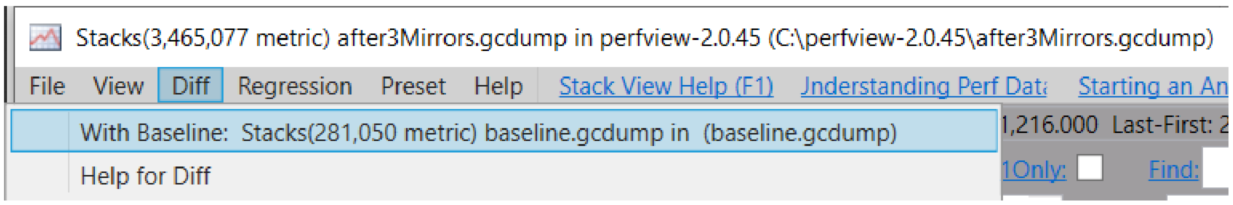


PerfView C:\perfview-2.0.45

C:\perfview-2.0.45

4. Do the same .gcdump file generation process but with the dump that contains unexpected memory.

5. In the last **Stacks** window, click **Diff** | **With Baseline:<name of baseline.gcdump>**



Stacks(3,465,077 metric) after3Mirrors.gcdump in perfview-2.0.45 (C:\perfview-2.0.45\after3Mirrors.gcdump)

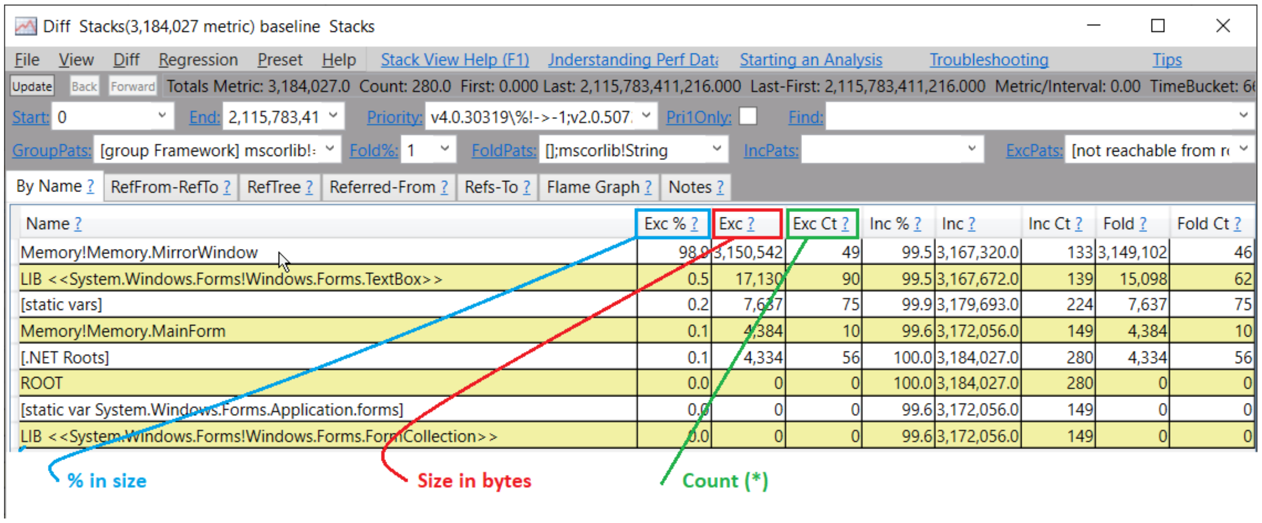
File View Diff Regression Preset Help Stack View Help (F1) Jnderstanding Perf Dat Starting an An

With Baseline: Stacks(281,050 metric) baseline.gcdump in (baseline.gcdump)

Help for Diff

*Note: Perfview lists all open **Stacks** windows (even those unrelated to memory such as CPU) so always check you select the one you want.*

6. The resulting **Diff Stacks** window shows up with the first By Name tab selected that contains the following columns:



Diff Stacks(3,184,027 metric) baseline Stacks

File View Diff Regression Preset Help Stack View Help (F1) Jnderstanding Perf Dat Starting an Analysis Troubleshooting Tips

Update Back Forward Totals Metric: 3,184,027.0 Count: 280.0 First: 0.000 Last: 2,115,783,411,216.000 Last-First: 2,115,783,411,216.000 Metric/Interval: 0.00 TimeBucket: 6

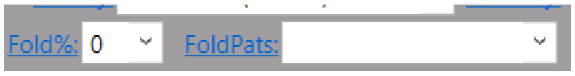
Start: 0 End: 2,115,783,41 Priority: v4.0.30319\%!->-1;v2.0.507 Pri1Only: Find:

GroupPats: [group Framework] mscorlib!: Fold%: 1 FoldPats: [];mscorlib!String IncPats: ExcPats: [not reachable from r

By Name ? | RefFrom-RefTo ? | RefTree ? | Referred-From ? | Refs-To ? | Flame Graph ? | Notes ?

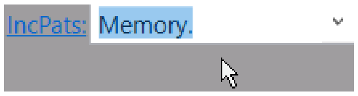
Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?	Fold ?	Fold Ct ?
Memory!Memory.MirrorWindow	98.9	3,150,542	49	99.5	3,167,320.0	133	3,149,102	46
LIB <<System.Windows.Forms!Windows.Forms.TextBox>>	0.5	17,130	90	99.5	3,167,672.0	139	15,098	62
[static vars]	0.2	7,637	75	99.9	3,179,693.0	224	7,637	75
Memory!Memory.MainForm	0.1	4,384	10	99.6	3,172,056.0	149	4,384	10
[.NET Roots]	0.1	4,334	56	100.0	3,184,027.0	280	4,334	56
ROOT	0.0	0	0	100.0	3,184,027.0	280	0	0
[static var System.Windows.Forms.Application.forms]	0.0	0	0	99.6	3,172,056.0	149	0	0
LIB <<System.Windows.Forms!Windows.Forms.FormCollection>>	0.0	0	0	99.6	3,172,056.0	149	0	0

The **Count** column might have invalid values due to folding default settings: empty the corresponding textbox at the top to ensure consistent numbers



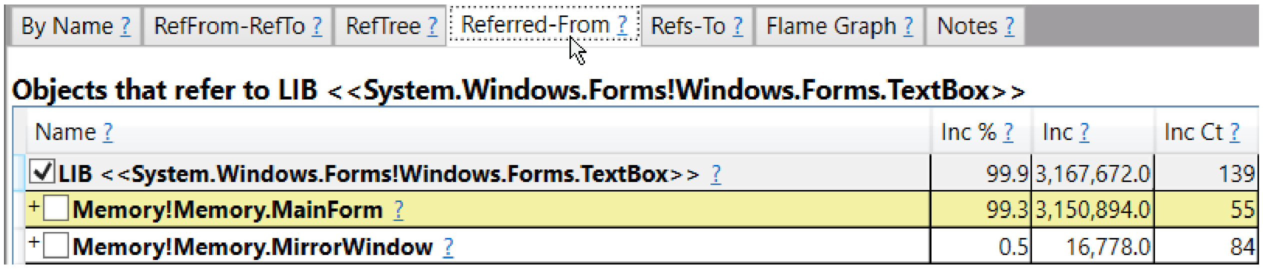
Fold%: 0 FoldPats:

7. Start by looking for types of your application: i.e. with **Memory** namespace in the **IncPats** textbox:



IncPats: Memory.

Don’t be surprised if the Diff Stacks window lists types without Memory in their name: it is due to the fact that Perfview sees this memory diff as a call stacks diff! However, instead of a real thread call stack, it is more related to a references tree. For example, if double-click the line corresponding to **System.Windows.Forms!Windows.Forms.TextBox**, the current tab switches to **Referred to** :

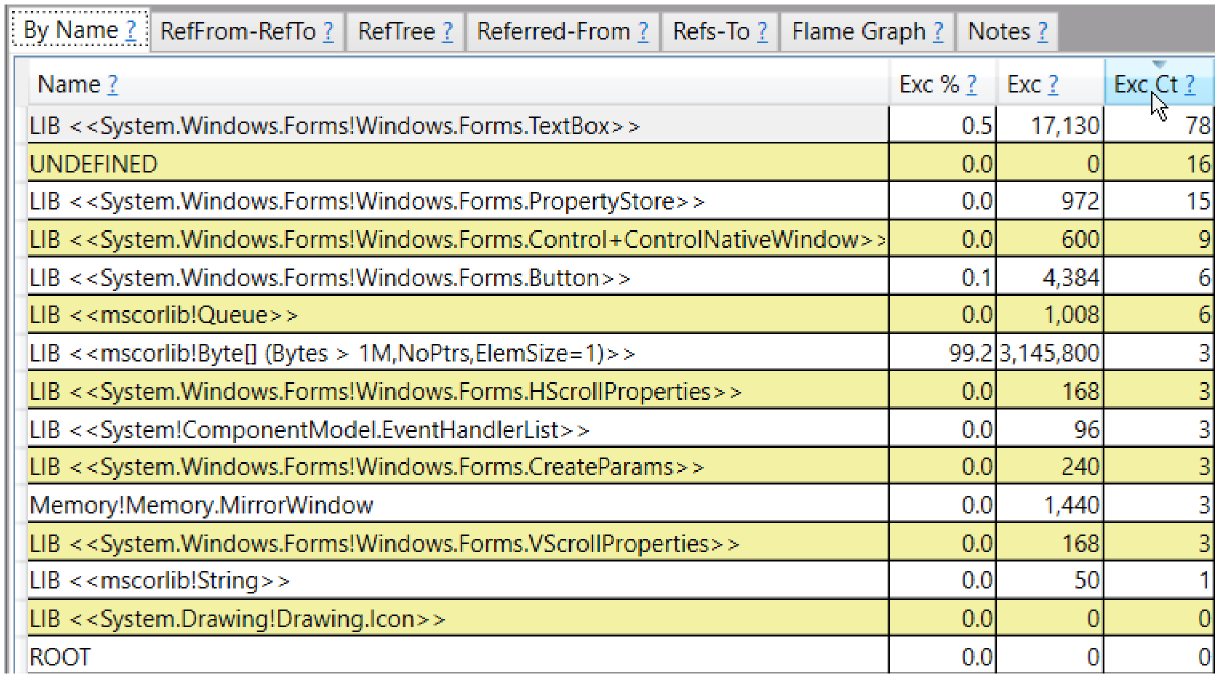

By Name ? | RefFrom-RefTo ? | RefTree ? | Referred-From ? | Refs-To ? | Flame Graph ? | Notes ?

Objects that refer to LIB <<System.Windows.Forms!Windows.Forms.TextBox>>

Name ?	Inc % ?	Inc ?	Inc Ct ?
✓ **LIB <<System.Windows.Forms!Windows.Forms.TextBox>>** ?	99.9	3,167,672.0	139
+ **Memory!Memory.MainForm** ?	99.3	3,150,894.0	55
+ **Memory!Memory.MirrorWindow** ?	0.5	16,778.0	84

And you can see that **Memory.MainForm** and **Memory.MirrorWindow** are both referencing **TextBox**. This is expected since each window contains such a TextBox.

8. Go back to the **By Name** tab and click the **Exc Ct column** to sort by diff count:

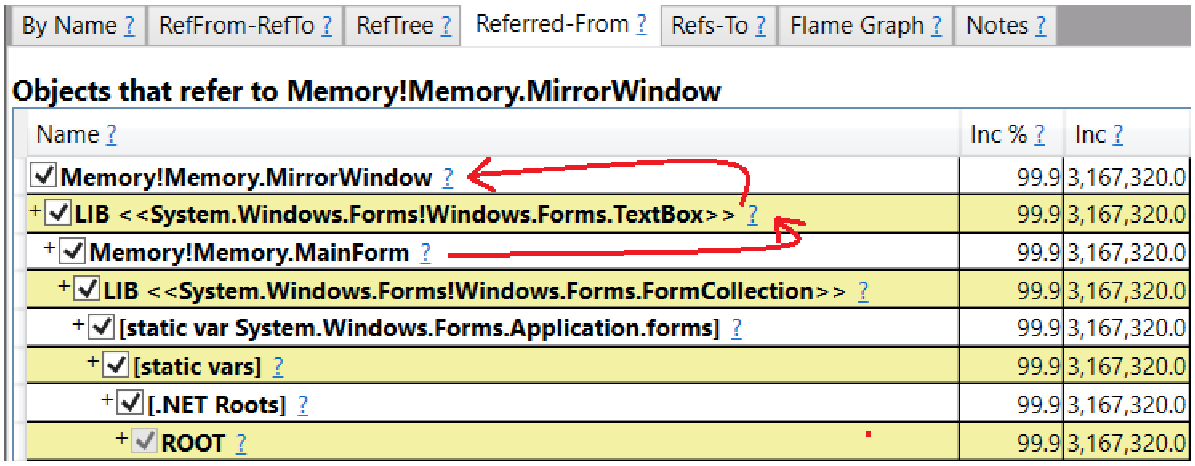

By Name ? | RefFrom-RefTo ? | RefTree ? | Referred-From ? | Refs-To ? | Flame Graph ? | Notes ?

Name ?	Exc % ?	Exc ?	Exc Ct ?
LIB <<System.Windows.Forms!Windows.Forms.TextBox>>	0.5	17,130	78
UNDEFINED	0.0	0	16
LIB <<System.Windows.Forms!Windows.Forms.PropertyStore>>	0.0	972	15
LIB <<System.Windows.Forms!Windows.Forms.Control+ControlNativeWindow>>	0.0	600	9
LIB <<System.Windows.Forms!Windows.Forms.Button>>	0.1	4,384	6
LIB <<mscorlib!Queue>>	0.0	1,008	6
LIB <<mscorlib!Byte[] (Bytes > 1M,NoPtrs,ElemSize=1)>>	99.2	3,145,800	3
LIB <<System.Windows.Forms!Windows.Forms.HScrollProperties>>	0.0	168	3
LIB <<System!ComponentModel.EventHandlerList>>	0.0	96	3
LIB <<System.Windows.Forms!Windows.Forms.CreateParams>>	0.0	240	3
Memory!Memory.MirrorWindow	0.0	1,440	3
LIB <<System.Windows.Forms!Windows.Forms.VScrollProperties>>	0.0	168	3
LIB <<mscorlib!String>>	0.0	50	1
LIB <<System.Drawing!Drawing.Icon>>	0.0	0	0
ROOT	0.0	0	0

Note: if you see negative value, it simply means that less instances of the types were present in this heap compared to the baseline.

The idea now is to look for counts that are a multiple of 3 (hence the interest to reproduce the leaky workflow an easy to spot number of times) and especially types from our own code. Unfortunately, it is not possible to sort by the **Name** column to more easily spot them…

9. Since **MirrorWindow** instances are not supposed to be in the heap dump, let's double-click the corresponding line to see which other types are referencing them

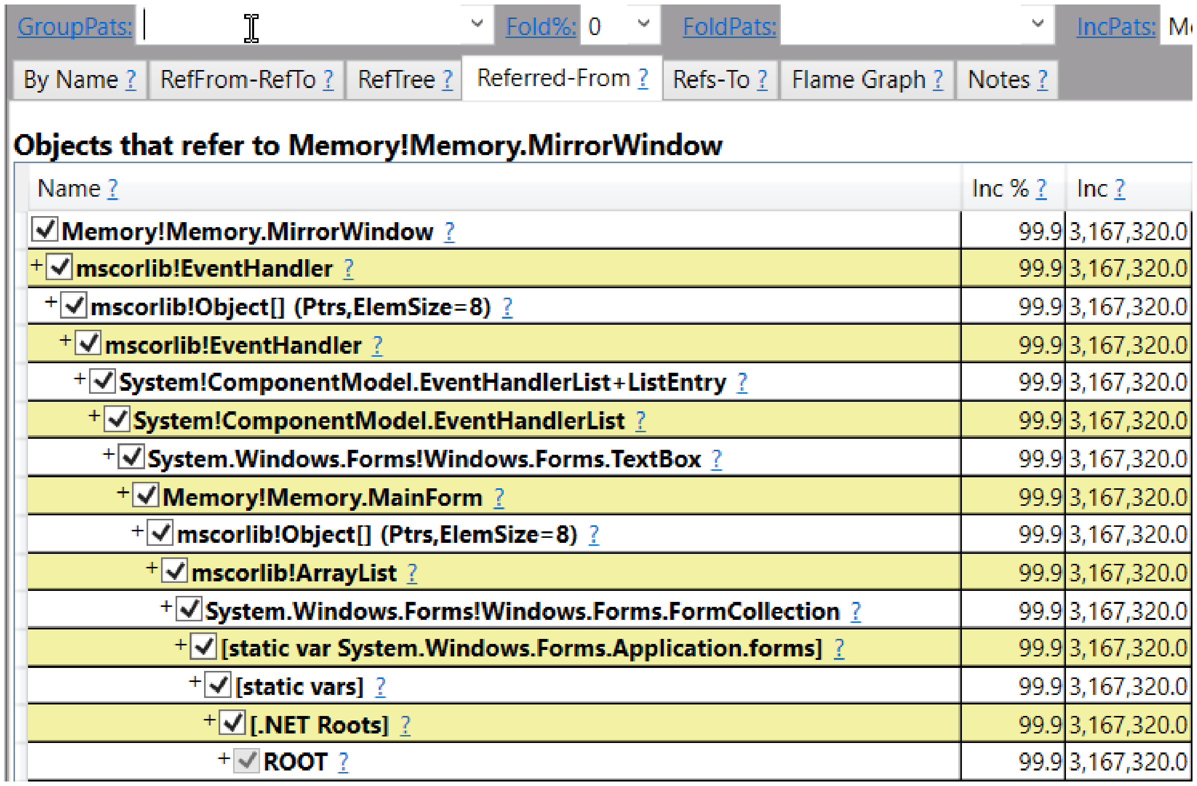


Objects that refer to Memory!Memory.MirrorWindow

Name ?	Inc % ?	Inc ?
✓ Memory!Memory.MirrorWindow ?	99.9	3,167,320.0
+ ✓ LIB <<System.Windows.Forms!Windows.Forms.TextBox>> ?	99.9	3,167,320.0
+ ✓ Memory!Memory.MainForm ?	99.9	3,167,320.0
+ ✓ LIB <<System.Windows.Forms!Windows.Forms.FormCollection>> ?	99.9	3,167,320.0
+ ✓ [static var System.Windows.Forms.Application.forms] ?	99.9	3,167,320.0
+ ✓ [static vars] ?	99.9	3,167,320.0
+ ✓ [.NET Roots] ?	99.9	3,167,320.0
+ ✓ ROOT ?	99.9	3,167,320.0

We see that the **MainForm** is referencing a **TextBox** that references the **MirrorWindow**: this does not really make sense and some pieces must be missing…

This is due to the grouping "feature" of Perfview: empty the **GroupPats** textbox on the top left to get a complete view of the references:



Objects that refer to Memory!Memory.MirrorWindow

Name ?	Inc % ?	Inc ?
✓ Memory!Memory.MirrorWindow ?	99.9	3,167,320.0
+ ✓ mscorlib!EventHandler ?	99.9	3,167,320.0
+ ✓ mscorlib!Object[] (Ptrs,ElemSize=8) ?	99.9	3,167,320.0
+ ✓ mscorlib!EventHandler ?	99.9	3,167,320.0
+ ✓ System!ComponentModel.EventHandlerList+ListEntry ?	99.9	3,167,320.0
+ ✓ System!ComponentModel.EventHandlerList ?	99.9	3,167,320.0
+ ✓ System.Windows.Forms!Windows.Forms.TextBox ?	99.9	3,167,320.0
+ ✓ Memory!Memory.MainForm ?	99.9	3,167,320.0
+ ✓ mscorlib!Object[] (Ptrs,ElemSize=8) ?	99.9	3,167,320.0
+ ✓ mscorlib!ArrayList ?	99.9	3,167,320.0
+ ✓ System.Windows.Forms!Windows.Forms.FormCollection ?	99.9	3,167,320.0
+ ✓ [static var System.Windows.Forms.Application.forms] ?	99.9	3,167,320.0
+ ✓ [static vars] ?	99.9	3,167,320.0
+ ✓ [.NET Roots] ?	99.9	3,167,320.0
+ ✓ ROOT ?	99.9	3,167,320.0

Now the role of the event handler becomes visible.

So… what is the cause of the memory leak and how to fix it?