A Motivating Example of WinDbg Scripting for .NET Developers

August 5, 2014

tags: Debugging, DEV, Tools 3 comments

WinDbg scripting gets a pretty bad name — its somewhat contrived syntax, weird limitations, and hard to decipher expressions being the common culprits. In some cases, however, WinDbg scripts can be a very effective and reliable tool for extracting information from memory and processing it a meaningful way.

This post offers a simple example that hopefully will be useful as you begin to explore WinDbg scripts. For a more thorough explanation and more complex scripts, make sure to check my past posts on traversing std::vector and std::map.

Let's set the stage with a simple console application that creates a number of heap objects and th waits for user input. This simulates an application server, perhaps, that processes requests and kee some of them in memory while they are pending.

```
namespace OrderProcessing
{
    class Order
    {
        public int CustomerId { get; set; }
        public string ProductName { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<Order> orders = new List<Order>();
            for (int i = 0; i < 100; ++i)
            {
                orders.Add(new Order
                {
                    CustomerId = i,
                    ProductName = "Product #" + i
                });
            }
            Console.ReadLine();
```

```
        }
    }
}
```

Now, suppose that we want to build a script that dumps out the product names for all important customers (whose customer id is higher than 50) that are currently in application memory. Doing with vanilla Visual Studio is very hard unless you know in advance where to find these objects. Furthermore, you want to do this in production — where you don't have Visual Studio installed.

Let's start by identifying **Order** objects in our heap. This is a fairly simple exercise in using the SOS **!dumpheap** command, which can take a type name:

```
0:003> .loadby sos clr
0:003> !dumpheap -type OrderProcessing.Order
 Address       MT     Size
028b2354 00c738b0       24
028b236c 73015738       16
028b237c 00c73860       16
028b392c 73015738       32
028b394c 00c73860       16
028b399c 00c73860       16
... truncated for brevity

Statistics:
      MT    Count    TotalSize Class Name
00c738b0        1           24
System.Collections.Generic.List`1[[OrderProcessing.Order, OrderProcessing]]
73015738        7         1120 System.Object[]
00c73860      100         1600 OrderProcessing.Order
Total 108 objects
```

Note that the statistics at the end indicate that we have more than just **Order** instances — we also have lists and arrays displayed. If we only want the **Order** objects, it's better to filter by their meth table pointer, like this:

```
0:003> !dumpheap -mt 00c73860
 Address       MT     Size
028b237c 00c73860       16
028b394c 00c73860       16
028b399c 00c73860       16
028b39ec 00c73860       16
028b3a3c 00c73860       16
... truncated for brevity
```

```
Statistics:
      MT    Count    TotalSize Class Name
00c73860      100         1600 OrderProcessing.Order
Total 100 objects
```

We already have a small challenge in front of us — in an automatic script, we really can't rely on t method table pointer staying the same between multiple runs of the application. Which means we need a way to find the method table pointer automatically, and that's where the **!Name2EE** command can help. The only problem is then parsing its output automatically, and that's a job for the **.foreach** command that can skip a number of tokens until it reaches the one w care about:

```
0:003> !Name2EE OrderProcessing!OrderProcessing.Order
Module:      00c72ed4
Assembly:    OrderProcessing.exe
Token:       02000002
MethodTable: 00c73860
EEClass:     00c71338
Name:        OrderProcessing.Order

0:003> .foreach /pS 7 /ps 100 (mt {!Name2EE
OrderProcessing!OrderProcessing.Order}) { .echo mt }
00c73860
```

Once we have that, it's time to explore an individual **Order** object so we can obtain the customer and product name from it automatically. First, let's look at the object's raw memory:

```
0:003> dd 028b5c30 L8
028b5c30  00c73860 028b5c60 0000005f 00000000
028b5c40  73053b04 0000005f 00000000 730521b4
```

The first 4 bytes, in bold above, are the method table pointer. They are followed by the object's fie — which seem to be in reverse declaration order. 5f looks like the customer id, and the preceding bytes look like a pointer (to a string, as we know). Now, suppose we want to print out just the customer id:

```
0:003> ? poi(028b5c30+8)
Evaluate expression: 95 = 0000005f
```

The **poi** operator performs a simple memory dereference of the specified address. Indeed, we get (95 in decimal), which is this customer's id. What about the product name, which is a string? We simply need a double dereference. Let's start with looking at the string in memory:

```
0:003> dc 028b5c60 L8
```

```
028b5c70  00630075 00200074 00390023 00000035  u.c.t. .#.9.5...
```

The string's characters are clearly visible, but they don't start where the string object begins. The f two double words are **String**'s method table pointer and its length (0b). We need to skip 8 bytes the actual characters then and use the **du** command, which prints out a Unicode string:

```
0:003> du 028b5c60+8
028b5c68  "Product #95"
```

We have all the moving pieces and it's time to snap them together to a single script. We want to obtain **Order**'s method table pointer, find all **Order** objects on the heap, and for each **Order** obje evaluate an expression that says: "If the customer id is greater than 50, print out the product nam for that order."

Here's the full script that does this. You can paste it into a text file and then execute it in the debugger using the **$$>< scriptfile.txt**command.

```
.foreach /pS 7 /ps 100 (mt {!Name2EE OrderProcessing!OrderProcessing.Order})
{
  r $t0 = mt
}
.foreach (order {!dumpheap -mt @$t0 -short})
{
  .if (poi( order +8)>50)
  {
    du poi( order +4)+8
  }
}
```

The first part sets the **$t0** pseudo-register (variable) to the method table of the **Order** class. The n part iterates over all the **Order**objects and evaluates the condition we specified. Note the spaces around the **order** iteration variable — they are required for the debugger to successfully perform string interpolation.

This example hopefully provides the motivation for using WinDbg scripts in the real world. They c automate processes that would otherwise take minutes or hours to complete, and can give you insight into what's going on in your application in a reusable fashion.

I am posting short links and updates on Twitter as well as on this blog. You can follow me: @goldsh