**How to reverse engineer (RE) the NMS exe.**

Credit to @monkeyman192#7076 for this guide. All I (@Gmr\_Leon#1477) did was rewrite the start and a few other spots to adjust for x64dbg’s differences from IDA and supply x64dbg screenshots to (mostly) match for Monkeyman’s screenshots.

**Basic program setup and initial run:**

In this tutorial, we will look at how to use [x64dbg](http://x64dbg.com/) to RE the NMS.exe to discover the properties of the many structs used in game to be used to decompile and recompile mbin files.

Download the latest snapshot from the linked site and unzip it using 7zip or whichever program you prefer to wherever you wish. Right-click and run x96dbg.exe as admin in the release folder to configure it as you wish.

Adding shell commands adds it to your right-click context menu, allowing you to right click pretty much anything on your PC and open it with x64dbg to check out. The second dialog box simply has you pop in some desktop shortcuts (which I’d recommend, as the .exes for the main program are buried away in a roundabout place).

First, open the exe with x64dbg and open NMS.exe through the File Menu (making sure you have the 64-bit version of x64dbg open), or go to where your NMS.exe file is and right click it to debug with x64dbg if you said yes to the shell prompt mentioned above.

You will need to leave the program to run for a little while as it needs to parse the entire exe which may take a little while.

**Searching for Structs:**

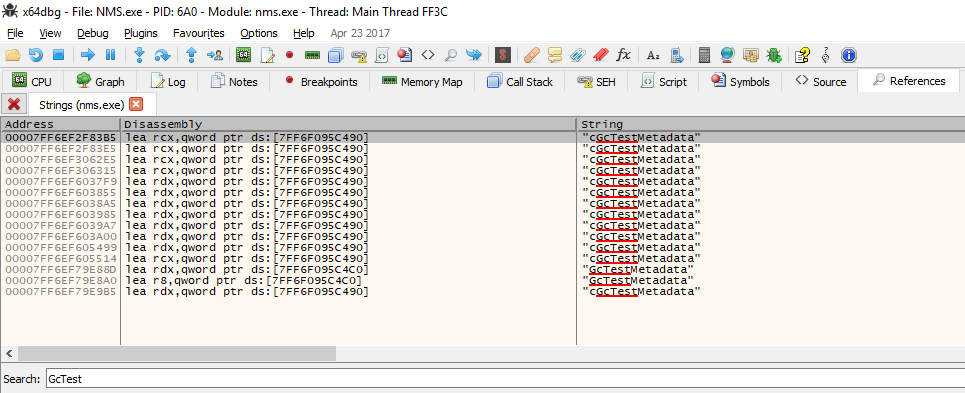
The first struct we will look for in the exe is the *GcTestMetadata* struct. We do this so that we can name several data types early on to assist us in the mapping of structs later.

To search for a struct, click the Symbols tab in the third row at the top of the window and in the left column double-click the nms.exe module, which will jump you back to the CPU tab. This limits the disassembly view (the upper left, largest frame shown) in the CPU tab to what we’re interested in. Once back in the CPU tab, right click in the disassembly view and go down the menu that appears to the last two options, Search For and Find References To.

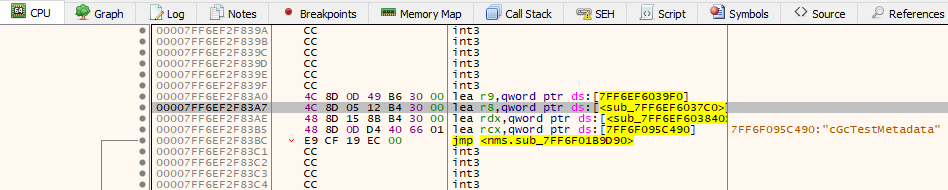
You’re interested in the first of those options, so hover your cursor over it and another menu appears, hover over Current Module and more options appear, with String References being what you’re after. Click this and it will take you to the References tab. Here x64dbg will begin searching the nms.exe module for all string references, which can take a moment. Wait a few seconds for this to do its thing.

Once done, you can begin entering in the search bar at the bottom for what you’re after, which in this case is *GcTestMetadata.* It’s not case sensitive though, so you can type it as you want (just remember it’s not meatdata!).

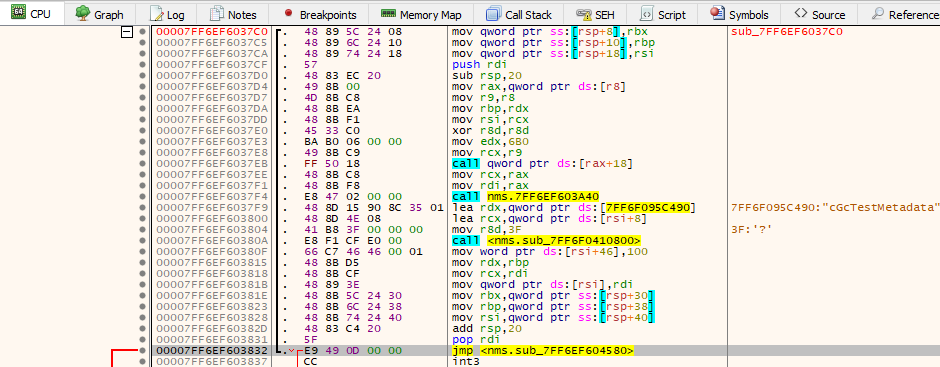
The results that appear should look something like this:



The locations will however vary depending on the version of the exe you are using. From this view, right click the first result and click Follow in Disassembler, which will bring you back to the CPU tab. You should then see this (after you scroll up thrice):



In *most* cases, the steps you will want to do are as follows:

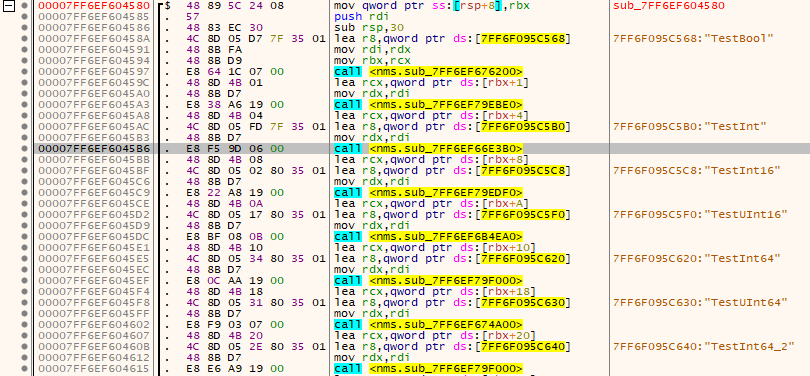
* Left click on the line with the *r8* string (usually 2 above the *rcx* string). Then right click anywhere along the selected line (it will highlight gray).  
  As above, you will again choose Follow in Disassembler, but this time there will be two options, *r8* or the *Address: [7FF6EF6037C0]* as in the above screenshot’s highlighted line. You want to select the *Address* option.
* On this next view, right before the *int3* tag, there is a line with *jmp nms.sub<hex number>*. Left click then right click this line, Follow in Disassembler, select Address again.

* This will take you to the actual data for the struct.

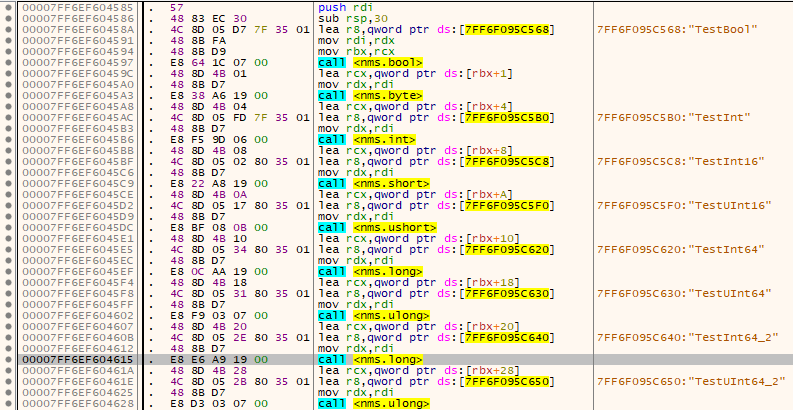
It should be noted that sometimes the *r8* sub location will take you straight to the struct, but normally you will need to follow the above simple process.

This sub you are at you should rename the sub\_<~> to be whatever the actual struct name is. This can help later on when something in a different struct uses this struct, you will already know what the data type is!

Now that we have the struct, we can begin naming things. First, open up the [GcTestMetadata.cs](https://github.com/monkeyman192/MBINCompiler/blob/master/MBINCompiler/Models/Structs/GcTestMetadata.cs) (MBINCompiler>Models>Structs) file either from a local clone of the MBINCompiler git repository, or open the file directly online. We will be comparing the structure of this file to the structure of the struct in the exe.



Comparing the highlighted sub in the above picture to the data type of the TestInt variable in the c# struct, we can see that it should be an int, thus we can relabel (by pressing *Alt* and *;* [semicolon]or right clicking and selecting Label then the .sub\_address option that appears, *not* current address) the sub.

Once we have relabeled the subs it should look something like this:

Note that I have added some extra things compared to the struct, namely the byte data type as it is missing (at time of writing) from the struct.

You can go through and name all the different data types.

Something to note is the *[rbx+20]* (for example) above each of the names. This is the location of the variable within the struct.

It is important to look at this value and think of whether it is what you expect it to be, because you can use it to both determine the size of the previous chunk of data, and use this to determine whether or not padding is potentially needed.

As you RE more and more of the exe you will get used to the sizes of different data types (eg, float is 0x4 (all numbers are generally in hex), Colour is 0x10, etc).

There are two more special types of data types that occur in the exe. One I will call a list, and the other an array (for lack of better names…)

**Lists:**

The list involves (unsurprisingly) a list of values of some certain length. We can see this in the exe by looking for example at the *TestStaticArray* variable.



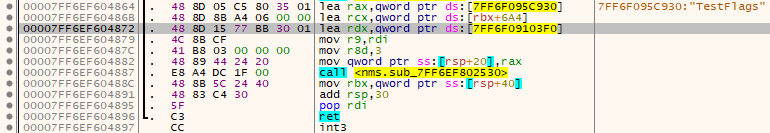
We see that this variable is located at 0x5FC, and is a list of floats (as seen in the c# struct). What is important to notice is that the exe tells us how long the list is. I have highlighted the length in the image (which is usually next to an *edx*) and you can compare this to the code in the struct.

You might ask, how do we know what the data type of the list is?! And this question isn’t always that easy to answer. If you scroll down past the struct there is another version of it (there are multiple versions of every struct in the exe, so sometimes it is required to check out these other versions to discern the data types of things) which can sometimes have more information of the problem. It also helps to have a look at the mbin and see what the data looks like at the location and see if it looks like anything familiar. Ultimately it sometimes just comes down to looking at the size of the struct that we are having a list of, and comparing that to other structs and just guessing by thinking about what data type it could possibly have. So a new struct to do with missions migh have a list of mission requirements, and there might be a mission requirements struct, so often you can work things out that way. In the end, lots of this comes down to practice and noticing patterns and understanding in a way how the structs work.

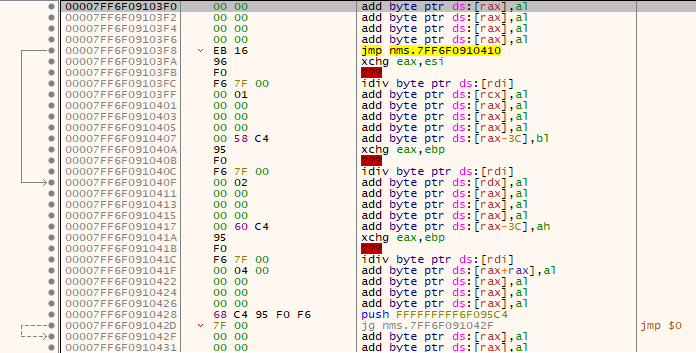
It should also be mentioned that there are dynamic length lists (as seen with the *TestDynamicArray* variable). The space taken up by each of these is 0x10, however in the mbin this will point to another location in the struct where the actual data is.

**Arrays:**

The next interesting data type is the array. The array is a list of values that can be chosen by the game, we will look at the *TestFlags* variable for this one.



If we left click the third line, right click and follow in disassembler the address in the *unk\_<hex number>* function, the view will change to a list of the possible options (or...whatever this is):



These might be the values in the list of strings that can be chosen and you can compare the code for these to the struct.

Once you have named at least most of the data types in this struct, you can start searching for other known structs and renaming any sub\_<~>’s.

Overall, this really is something that just take a bit of time to get used to doing. There are lots of little things to look out for, and there are plenty of little tricks. Things usch as pressing F5 when clicked on a sub will open up a window with a pseudocode view of the function, which can show you the data in a different way that is sometimes easier to analyse (I prefer the other view, but some prefer this pseudocode view).

Finally, a word on globals: Globals are *hard*. They do not obey the same rules as all the other structs (other than the debug globals for some odd reason…).

Finding the data for the globals in the exe is reasonably hard, but do-able when there is string data, but even once you have the location of the struct in the exe, it is still a fair bit of work to write the c# code for the struct from this data. It is only recommended that you look at these once you already have a very firm grasp on how everything works.