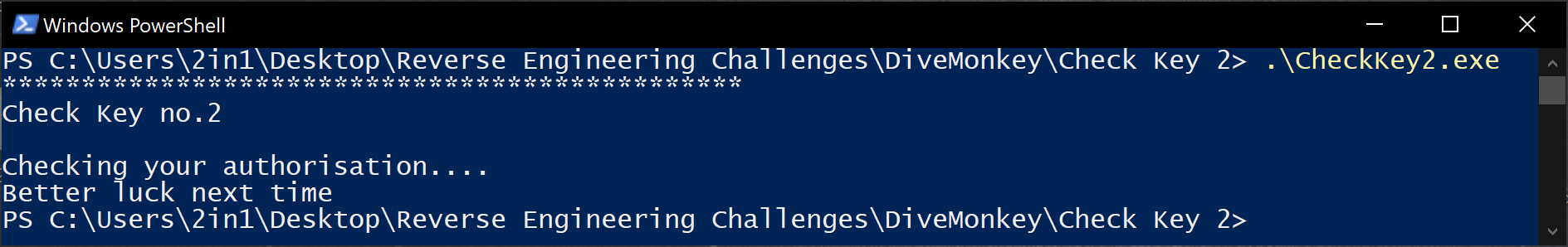
**DiveMonkey Challenge – CheckKey 2**

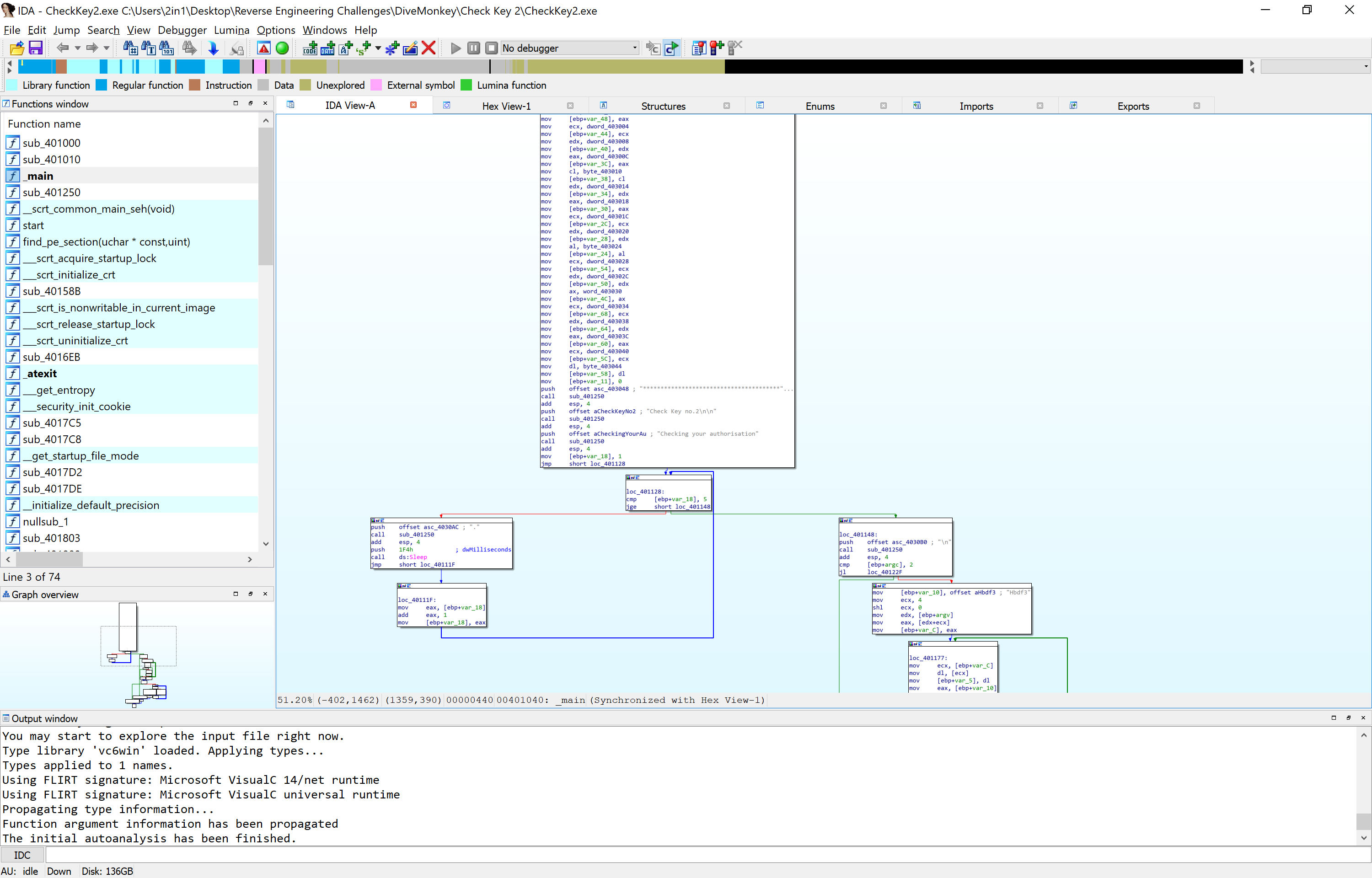
Firstly, I feel it right to admit that I initially needed a couple of hints to take the win with this, however once I got past the initial snag, I figured out the way in which we can bypass the string compare to automagically give us the flag! So, as before, we need to see what happens by default when we run this application.

Better luck next time? Challenge, accepted!

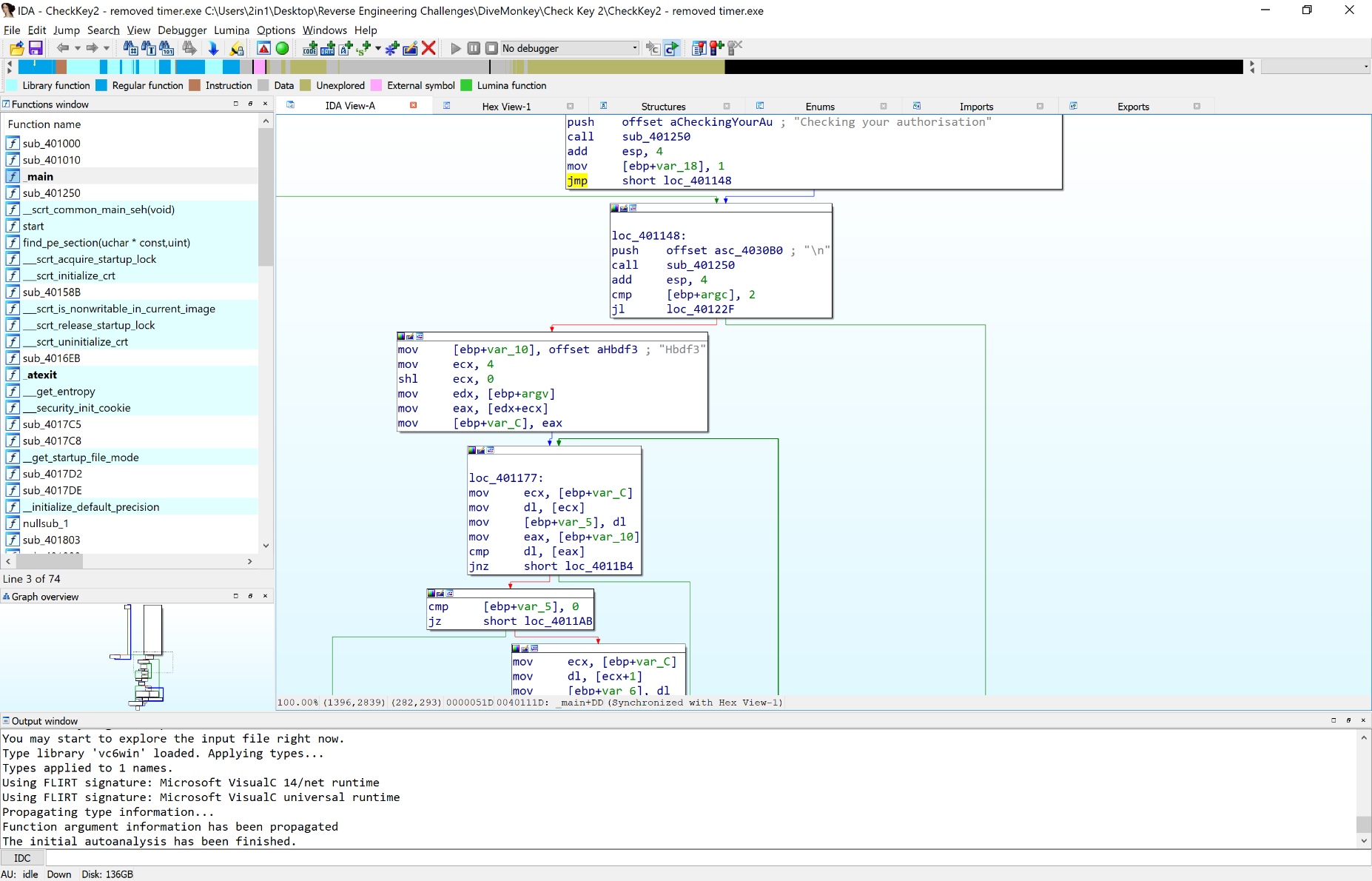
The same words of wisdom go out from the last challenge, but this time I’m going to show how you can use the power of each different debugger/disassembler to gain the insight into what is going on (although it can be solely achieved using just 1 of them)

To begin, I like to open IDA and gain a wider understanding of the entire flow diagram. I find that IDA displays all of the tree, opposed to x32 which will only show code from the current pointer (though you can select different code, re-graph and see what you want)

IDA automagically has found \_main (or the real entry point of the program) and we can see loads of niff naff before seeing the clear text strings we saw in the normal run, as well as our friendly timer loop before jumping off to more code.



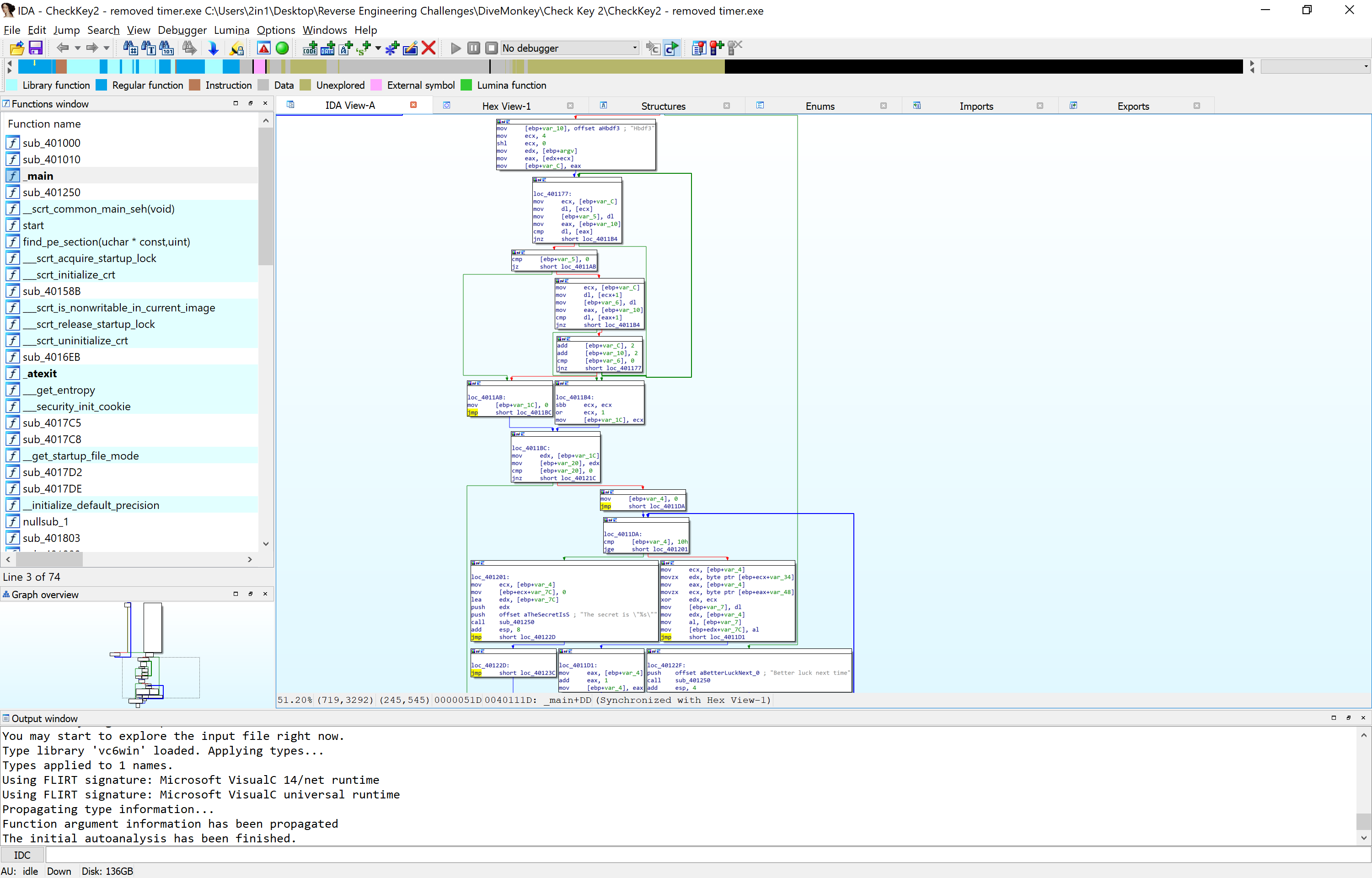
Personally the first thing I did was remove the timer loop. This removed the 5 second loop that, if the program was ran once in a blue moon isn’t too much of an annoyance, but when you begin to get more complicated programs, each loop that is only calling ‘sleep’ for a duration and incrementing a timer, just adds annoyance. However, certain programs may actually require this loop to run, in a form of anti debugging/tamper measures.



Changing from JGE – Jump if Condition is met, to JMP, forces the program to skip that 5 second, 4 …. Printing piece. After saving and running, we find that it doesn’t break anything, and saves us 5 seconds each run time. It all adds up when testing repeatedly!

The next stage however is what caught me out. This program is checking for any arguments, compares against the required amount (In this case 2. First being a check, checker, and the second is required input)

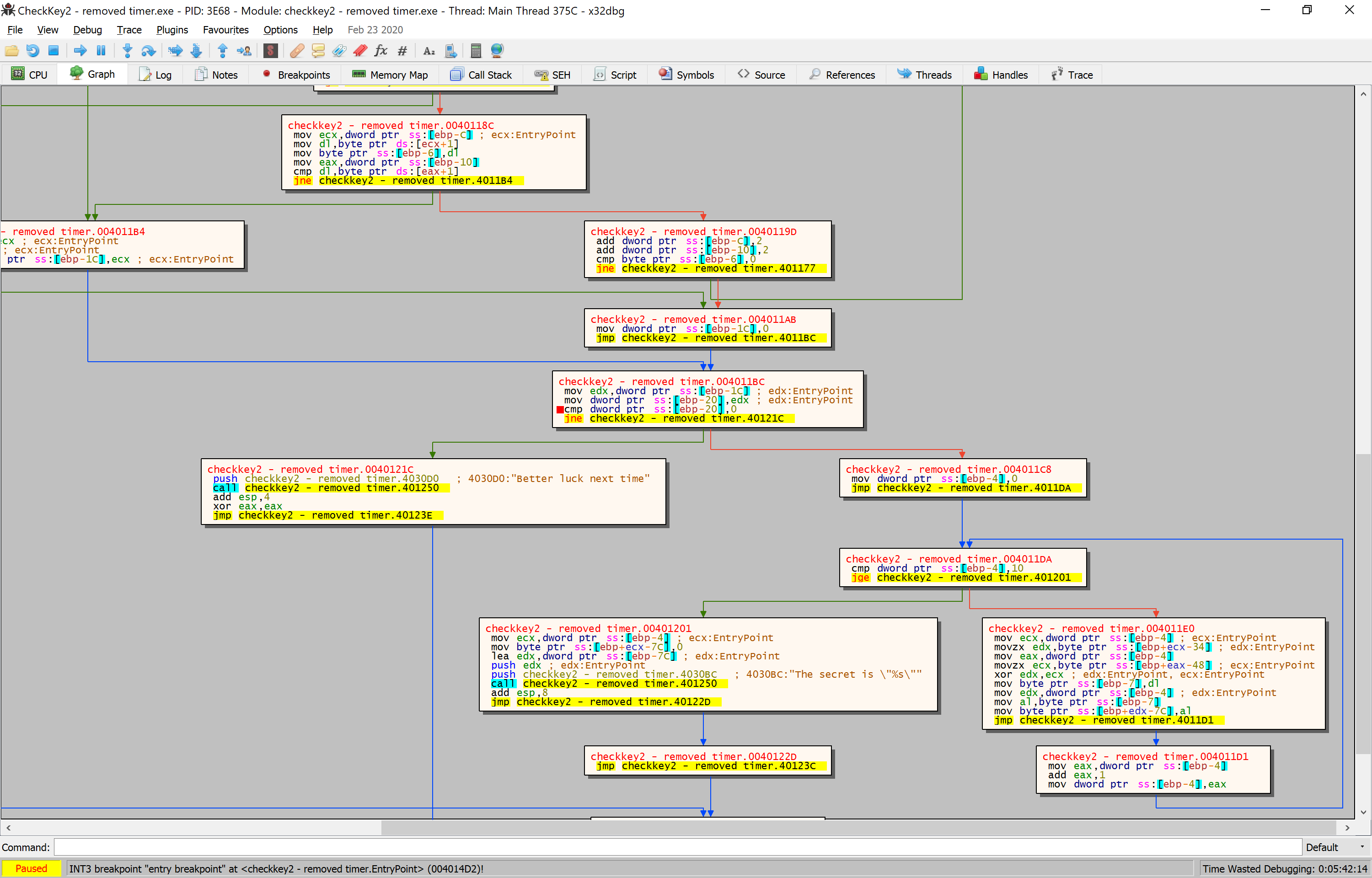
Upon checking arguments, if none have been found, it jumps to our friendly better luck next time comment. If it finds more than required amount, it also fails. On inputting one argument, it conducts a JL – Jump if condition met. The picture below showcases the part of the program in which it conducts it’s check on the arguments received, whether it’s the correct argument, then either gives us the secret passphrase or throws it in our face to have more luck once again.

Thankfully, the argument in question is not saved in any encrypted format and can be quite clearly seen at the top of this picture. Originally I thought this was a red herring!

If you attempt to edit the compare function to continue through, you end up with a memory allocation error. In short, because no argument was found, it tries to address memory that has not been allocated in any way. So we need to find a way in which to bypass that check, whilst ensuring that our conditions are met, opposed to the intended ones.

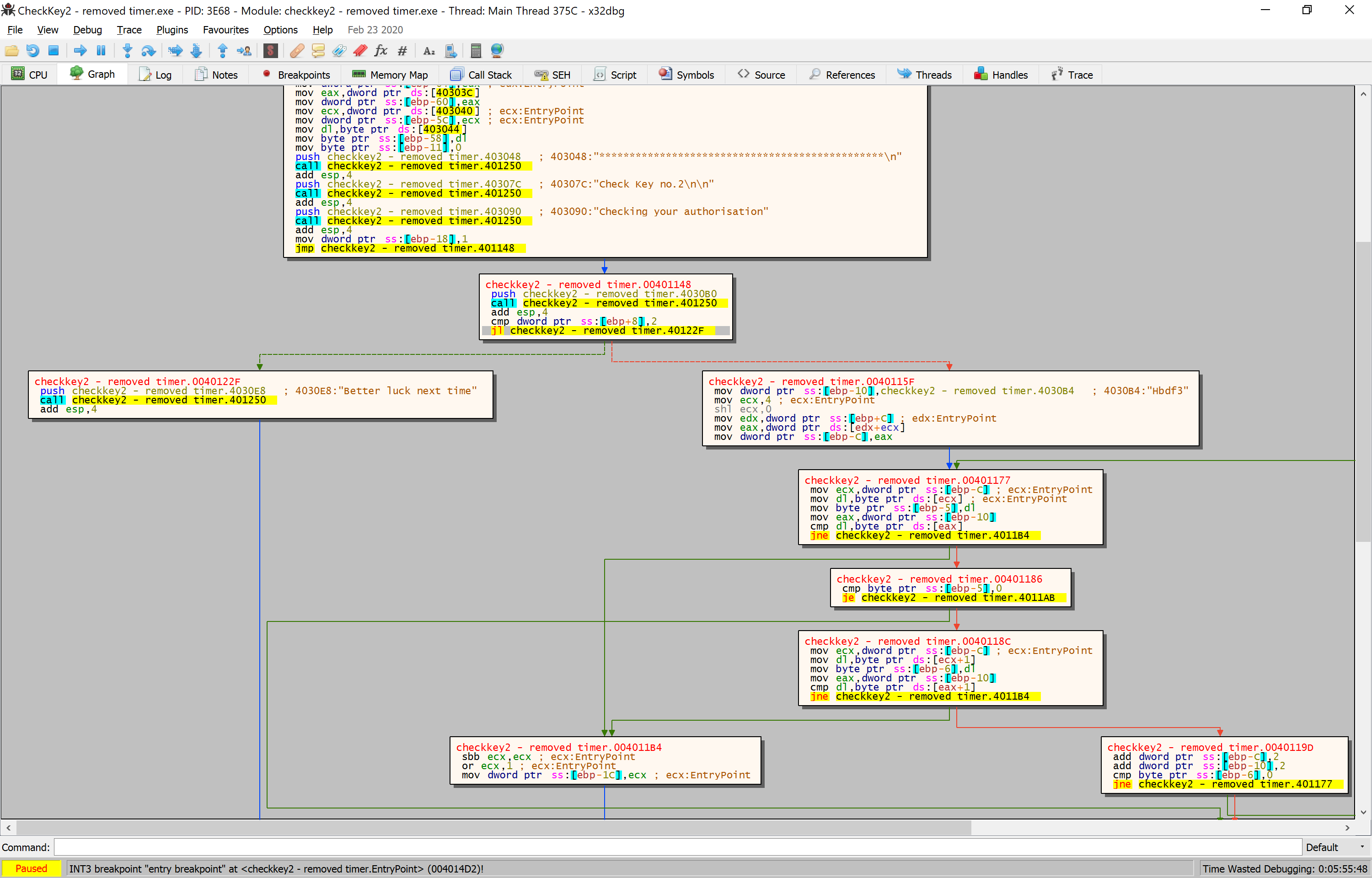
With this, we now load up x32DBG for easier address relocation patching (again, personal preference!) but more importantly, a better understanding of the diagram flow, as IDA likes putting everything close together, not always in a logical order.

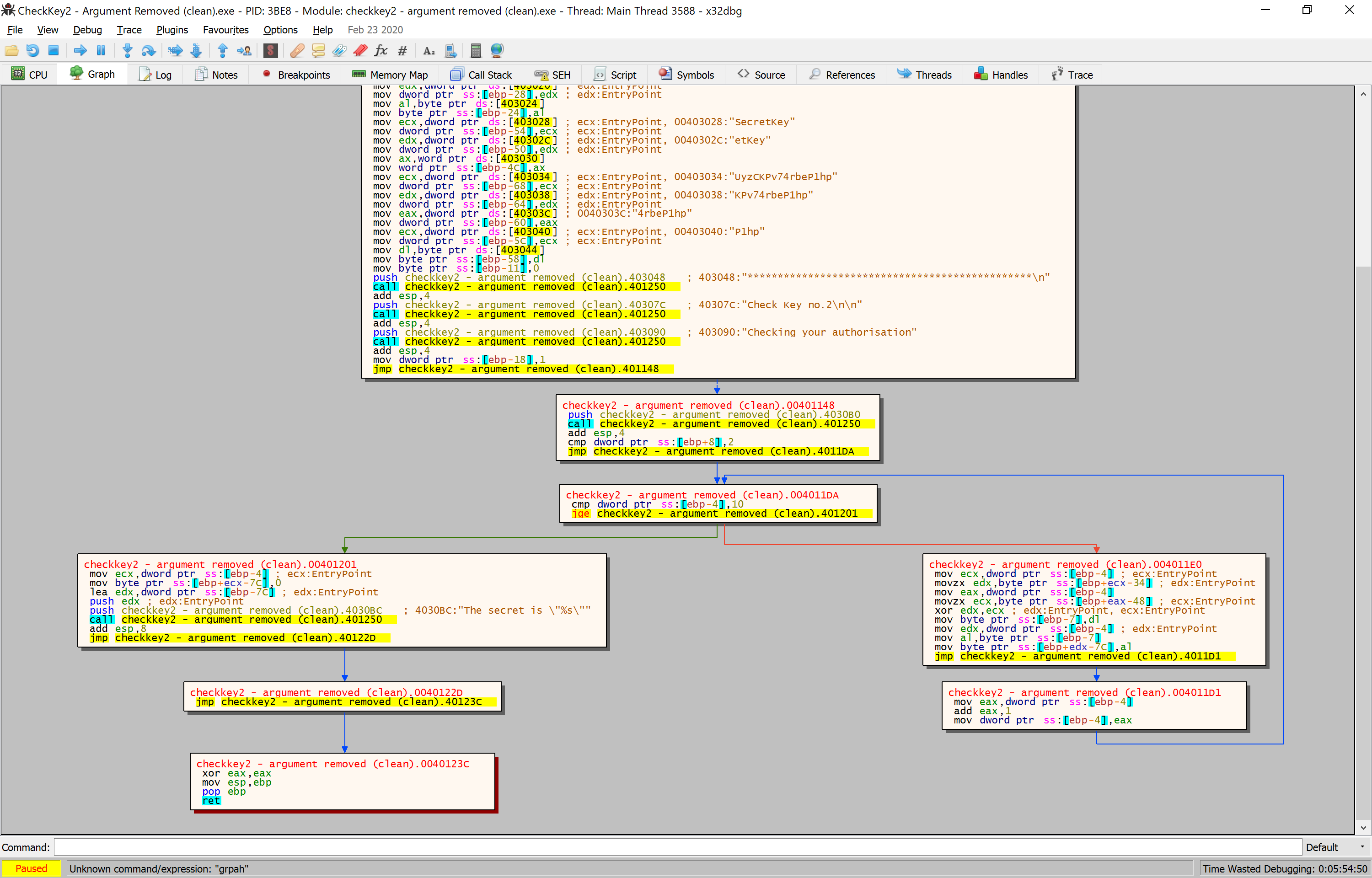
Upon loading, we can use the same addresses that IDA has given us, or using the same technique in Key 1, find \_main and initialise our graph. Following the graph a little further down, looking for clear breaks from ‘happening’ code and failure code, we get to this address region and find some compare loop, a XOR command, and plain text of the secret is…



With this find, we know that the secret key is XOR encrypted, but XOR encrypted means it also can be XOR decrypted. Time to make a big jump and see what happens if we skip the argument checker entirely (boxed in RED) and jump to the decryption loop (boxed in GREEN)

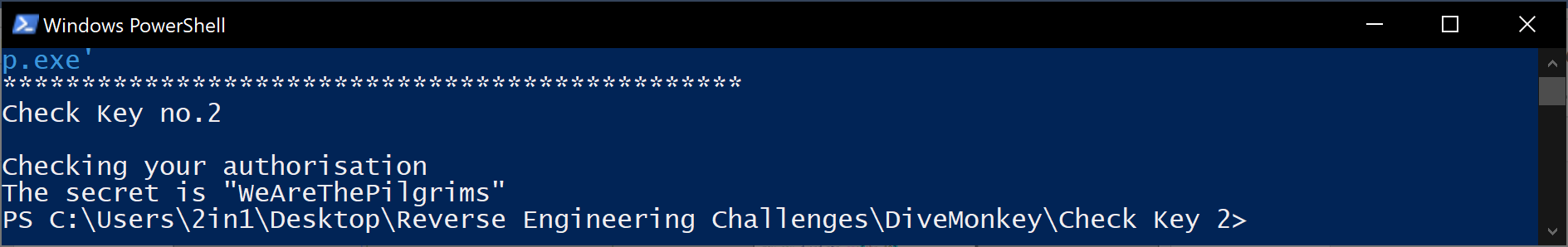
Using x32DBG, we can easily change addresses in real time, redraw the graph and see where the code goes to. By changing the function JL – XXX.40122F (found just after the big box containing the initial screen print of ‘checking authorisation’) into JMP XXX.4011DA, we are telling the program to load, then begin decrypting the key after 10 loops (cmp dword ptr ss:[ebp-4], 10) and continue with the program after 10 loops, into displaying the secret key…. We hope… (XXX is whatever you have the file named as)





Once we have patched this, we run the application in our command prompt and look what happens!

We have found the secret key beginning with W.



The reason this happens, is because we have jumped out a large proportion of code that checks the memory location of the argument and ensures that it is the one that is stored within the program itself. Should you attempt to bypass the check but not the validating code, you end up with a memory addressing error, since no argument was provided, the memory address hasn’t been used.

This however, is not the only way in which you can patch the program. You could patch it to pass checks with any argument, or how I originally made the hack ugly by pushing the valid argument onto the screen by calling XXX.4030B4 then calling XXX.401250 (which is called after every string, clearly a print text function)

By using 2 different programs in conjunction with each other, opposed to relying on just the one, we can see how different data on screen can actually aide our end result. While you could easily use just IDA, x32DBG or Ghidra (Ghidra wasn’t used in this tut or my attempt this time around)