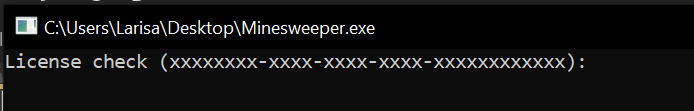
Project 0x02

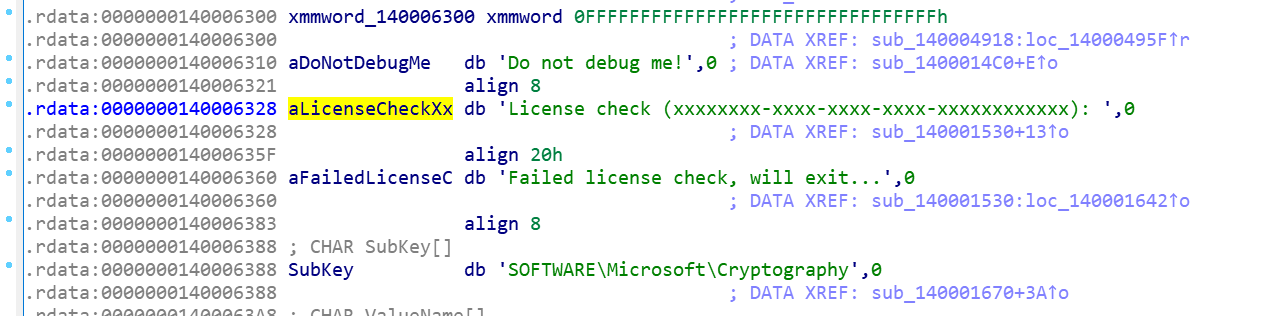
*Perform the following tasks:*

* *Access to the game is allowed only after entering a valid key. Understand how the validation system works and make your own software to generate random valid keys. (20p)*

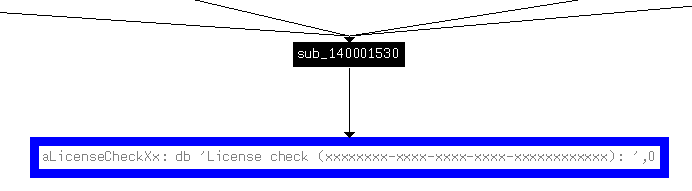
Clicking on the .exe, we get the following prompt:



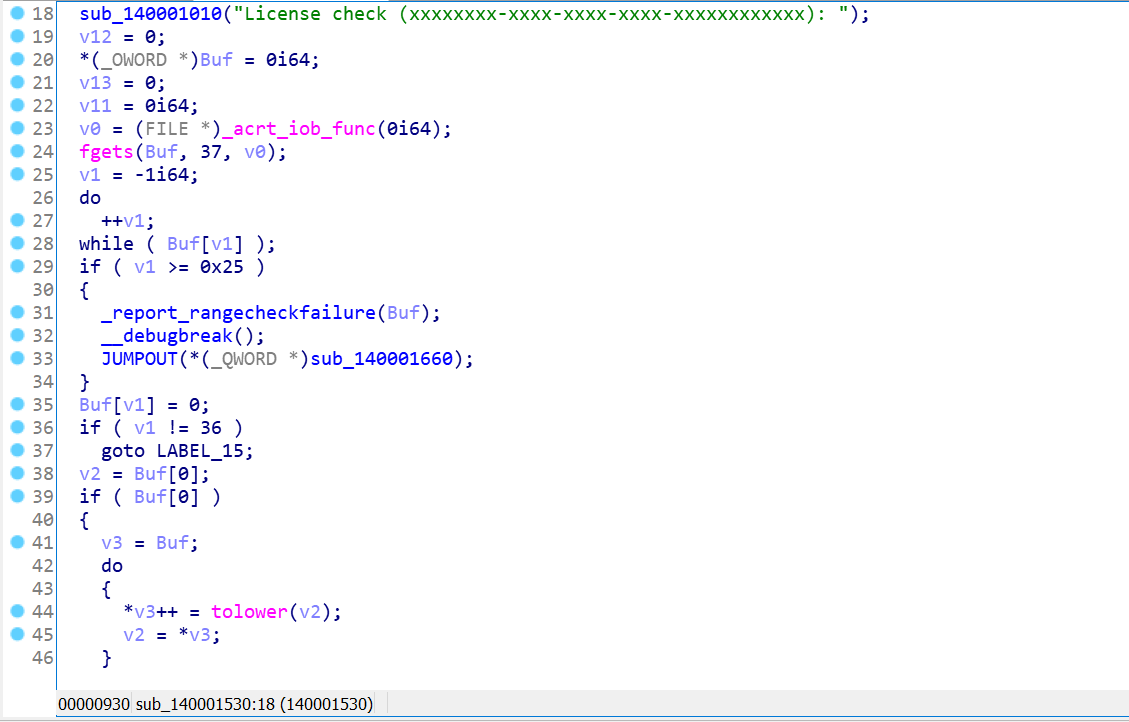
We enter the program in ***IDA Pro → CTRL + S → .rdata →*** and look after this string:



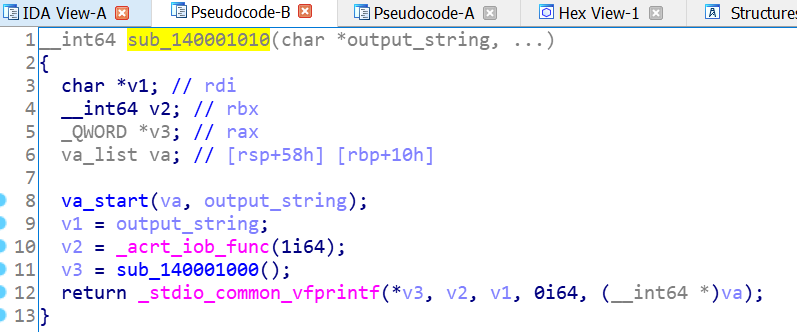
***Right click*** on the string and ***Xrefs Graph to***:



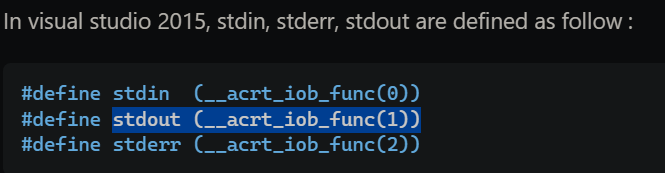
We found the function where it is used:

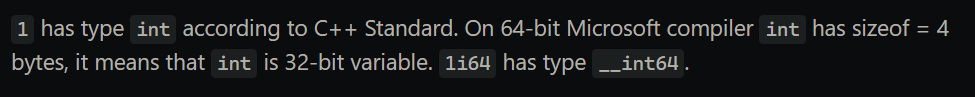


Let’s start with function ***sub\_140001010()*** – renamed as ***print\_to\_screen()***:

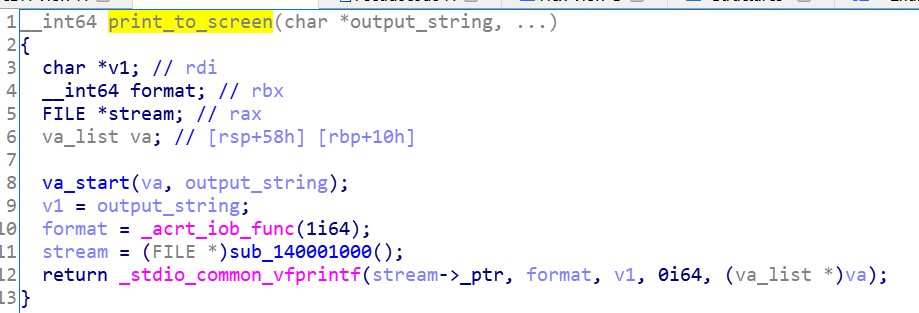


After looking through it, we can see it is a function that outputs the string to the screen. ([Source\_va\_start](https://cplusplus.com/reference/cstdarg/va_start/), [source\_acrt\_iob\_func](https://stackoverflow.com/questions/30412951/unresolved-external-symbol-imp-fprintf-and-imp-iob-func-sdl2), [source\_1i64](https://stackoverflow.com/questions/1264059/in-c-what-is-the-difference-between-1-and-1i64), [source\_stdio\_common\_vfprintf](https://cplusplus.com/reference/cstdio/vfprintf/))

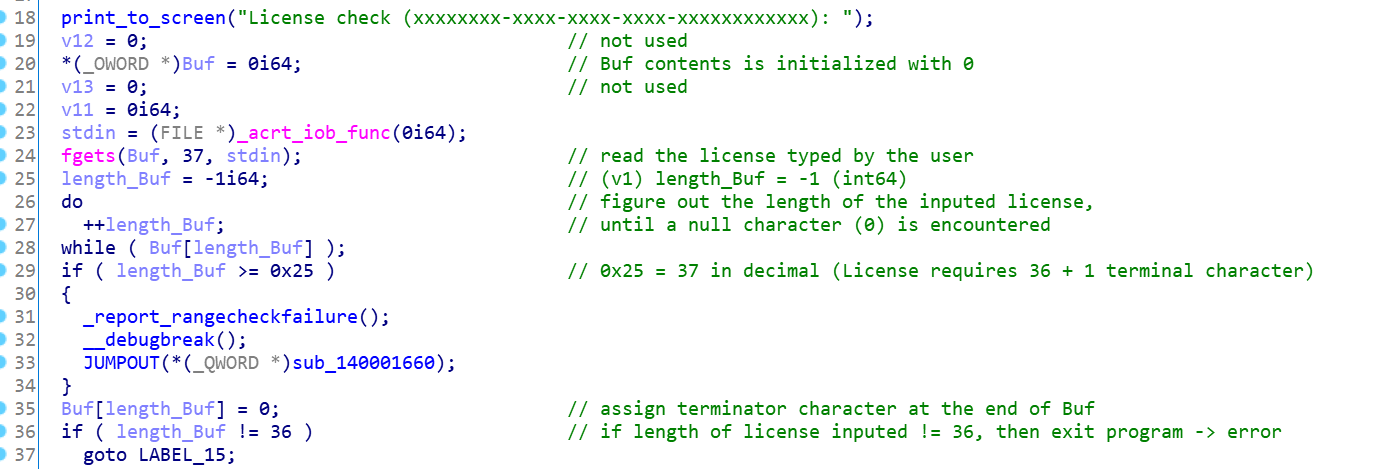




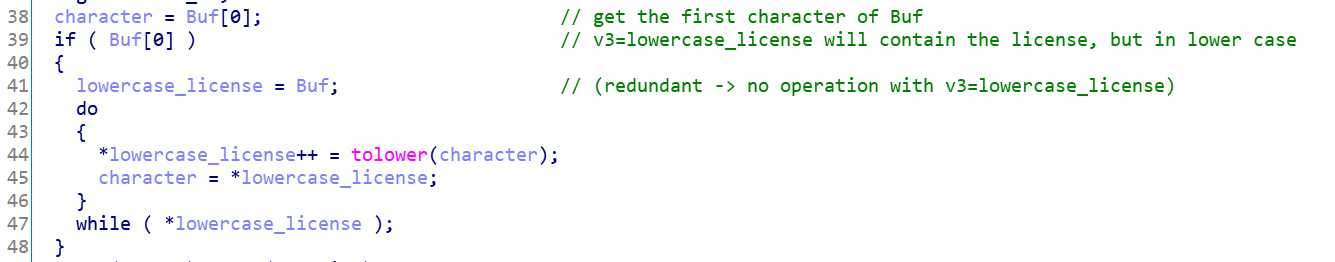
After renaming and retyping, we get:



Coming back to the original function (***sub\_140001530()***), we can see that firstly, the program gets the license from the user and performs different checks in order to ensure that the basic requirements are followed (length):



Next, we enter a loop, where the characters of the license are transformed to lower case and stored in v3 (renamed as ***lowercase\_license***):



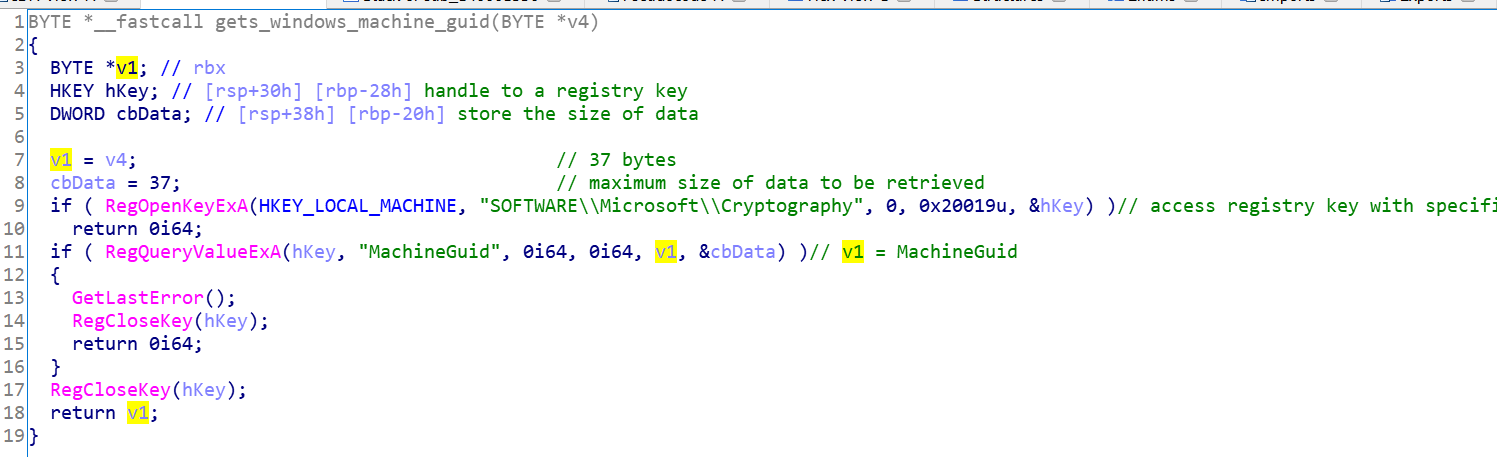
We dynamically allocate 37 bytes to ***v4***.



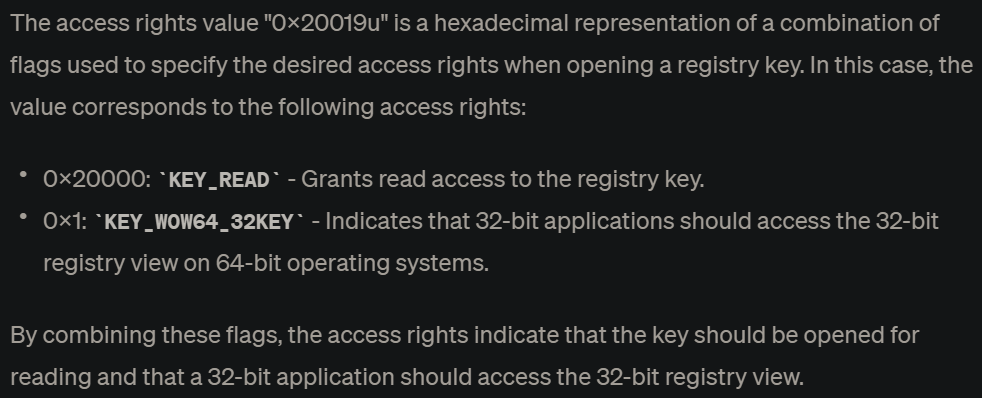
Let’s see what ***sub\_140001670()*** is doing with ***v4***:

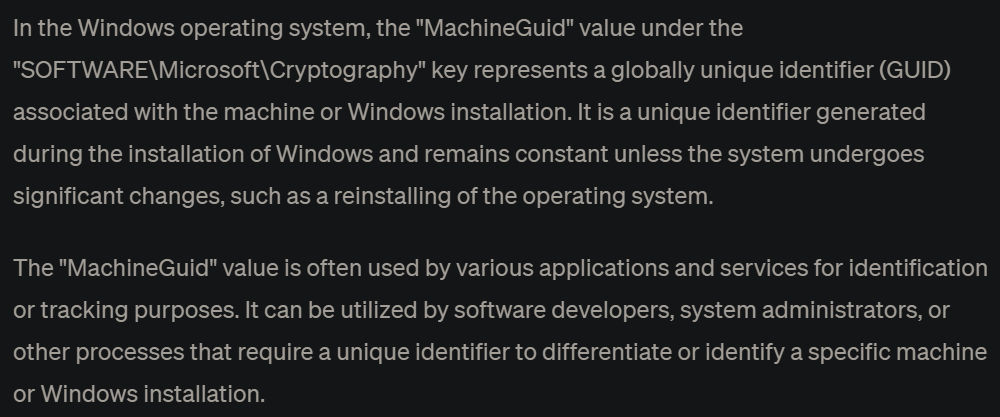


After renaming and retyping we get the following function.



Basically, this function, renamed as ***gets\_windows\_machine\_guid()***, tries to get the 37 byte value of the registry key „*MachineGuid*” into the buffer pointed by v1.

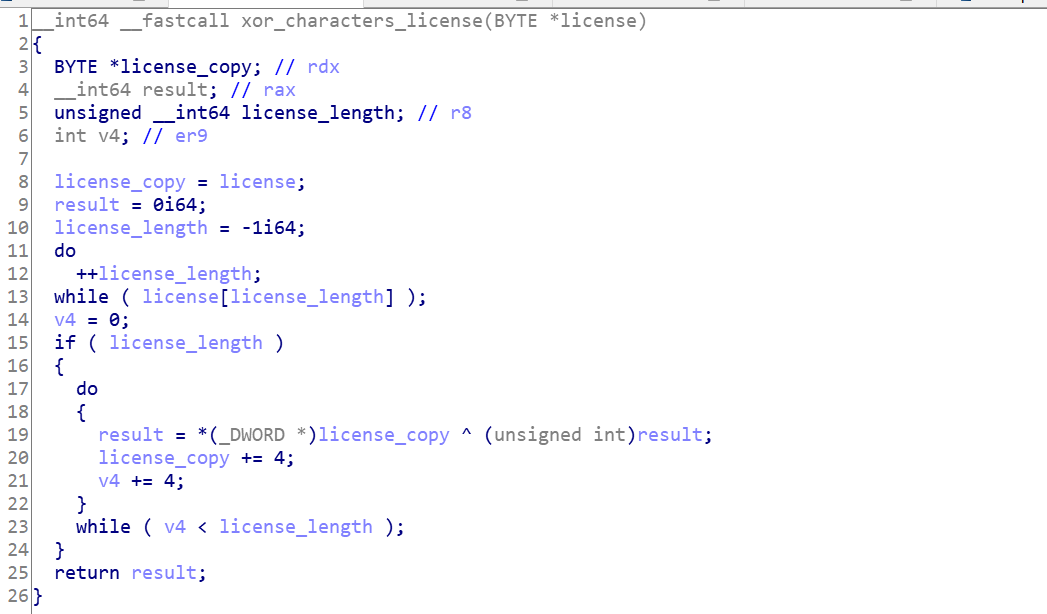




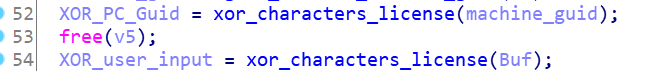
We move on to ***sub\_140001730()*** (renamed as ***xor\_characters\_license()***):



After renaming the variables, we can notice that this function performs an XOR operation between each block of 4 characters (1 byte == int; we also calculate the length of the license in order to know when to stop the xor operation):



We get the xor-ed value obtained from the MachineGuid of the computer (***v7 = XOR\_PC\_Guid***) and the xor-ed value obtained from the input (from the user; ***v8 = XOR\_user\_input***):

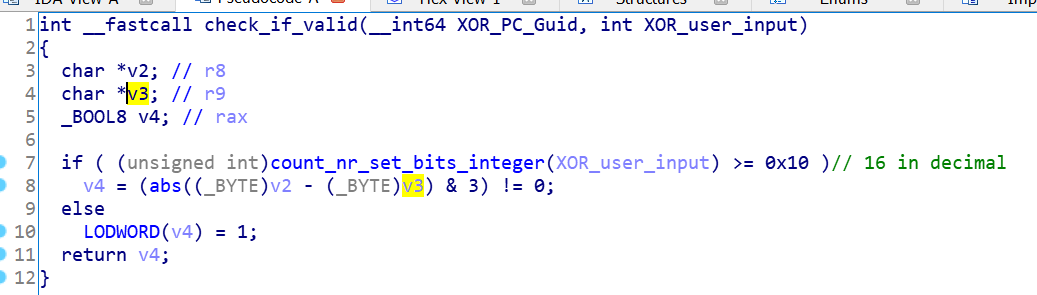


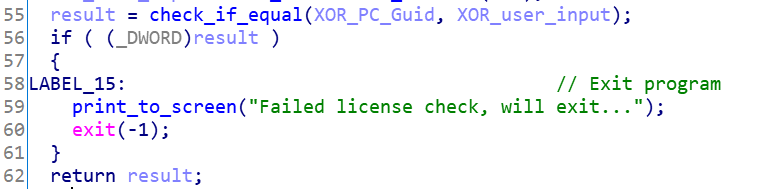
I wrote a C++ code that does that: ***xor\_characters.cpp***

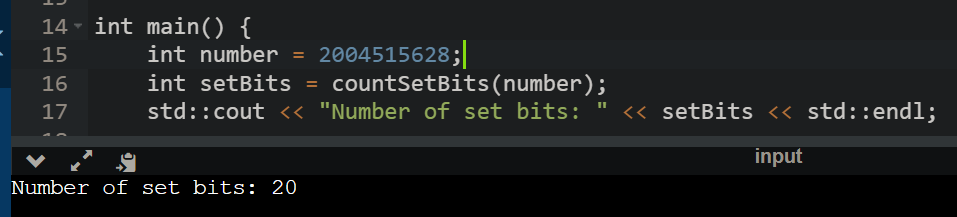


Moving on, ***sub\_1400014F0()*** (renamed as: ***check\_if\_valid()***) will check if the the license is valid: using the Brian Kernighan's algorithm to count the number of set bits in an integer (function renamed as ***count\_nr\_set\_bits\_integer()***) we test if the XOR-ed value from the user’s input has >= 16 set bits (I wrote a function: ***count\_set\_bits.cpp***). If false, then result = 1 and the check will fail; otherwise we check line 8 (v2 = XOR\_PC\_Guid, v3 = XOR\_user\_input → ***final\_validation.cpp***).

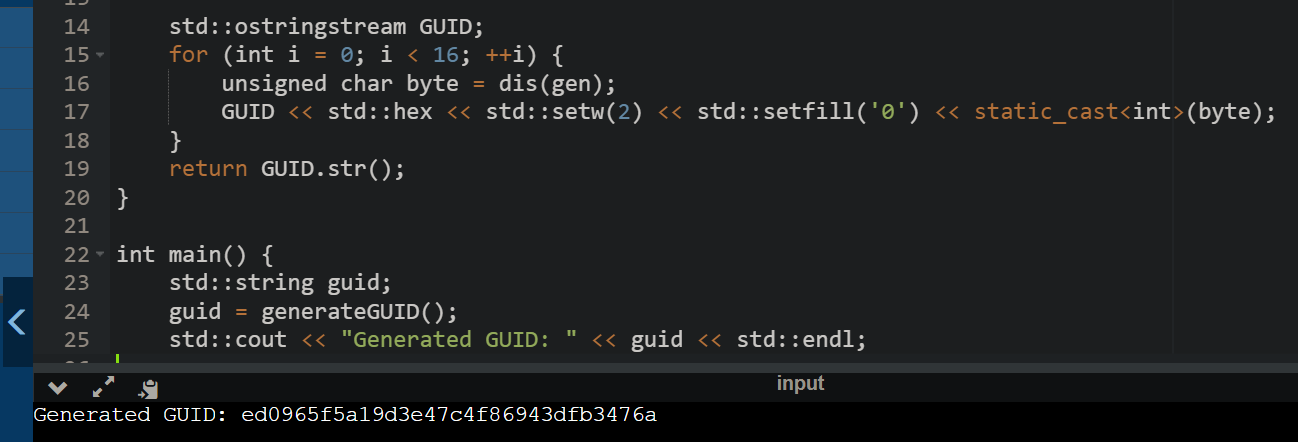




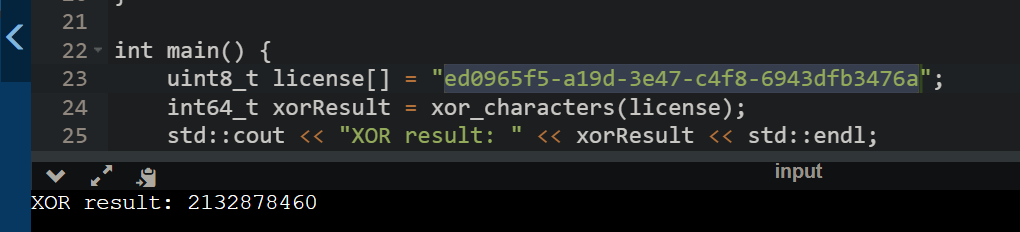


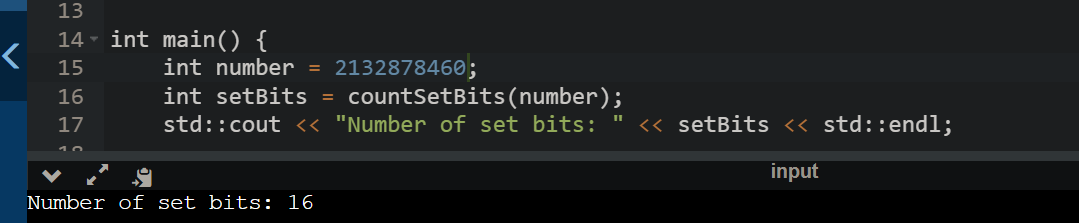


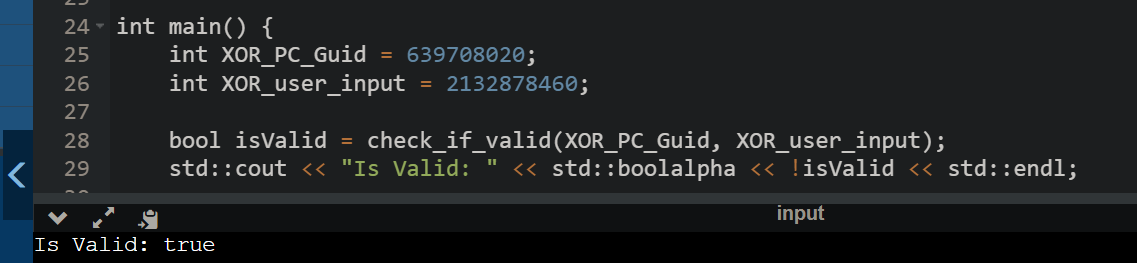
Now we know what this function is doing. We rename it ***check\_license()***. I wrote a C++ code that generates valid license: ***generate\_license.cpp*** (after a license is generated, then check with ***xor\_characters.cpp, count\_set\_bits.cpp*** and ***final\_validation.cpp***)



Let’s test it:





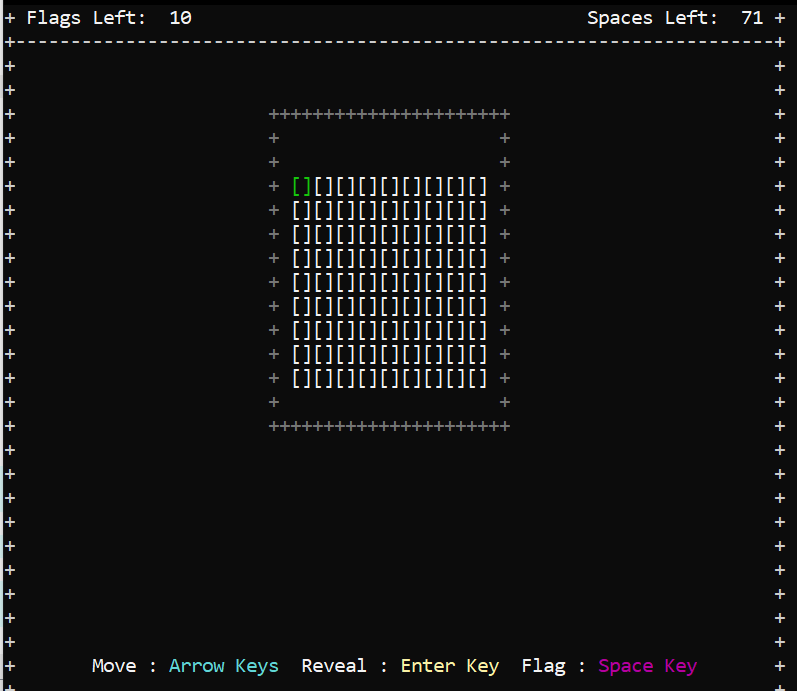


Working just fine.

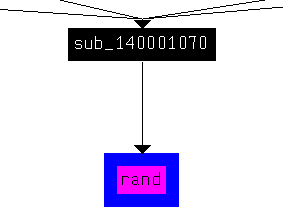
Another valid license: *6119326f-ac1e-4bb2-af55-0e0714d6dbf6*

* *Reveal all the bombs in the game, for all 3 levels of difficulty. (40p)*

This is the intro screen:

We can assume that the positions of the bombs are random. Click in ***Imports tab, double click on rand, right click on rand, Xrefs to***:

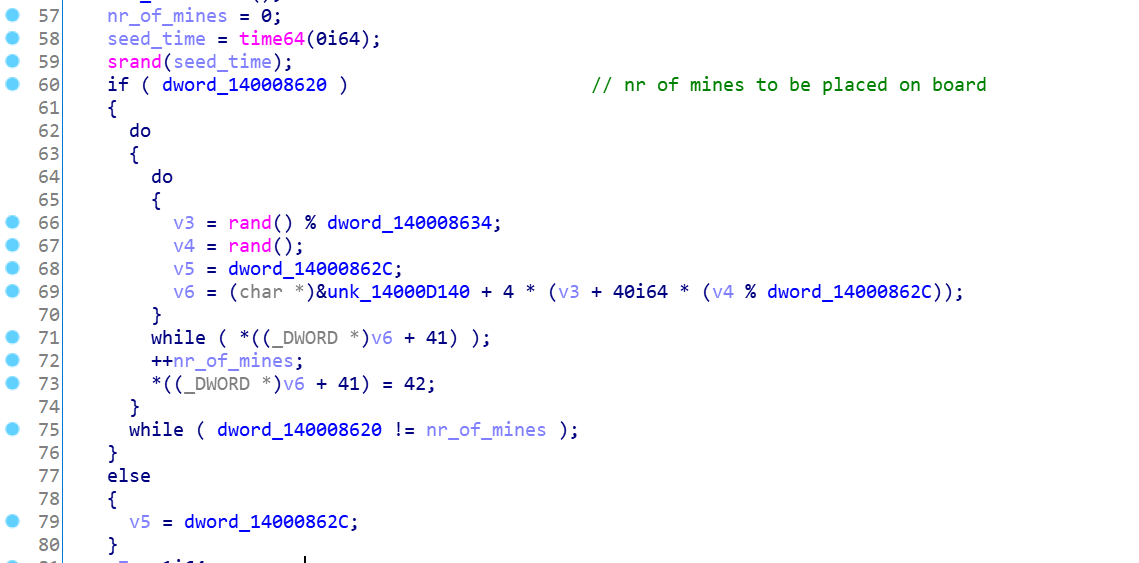
 We can note that ***srand*** also xrefs to the same function.

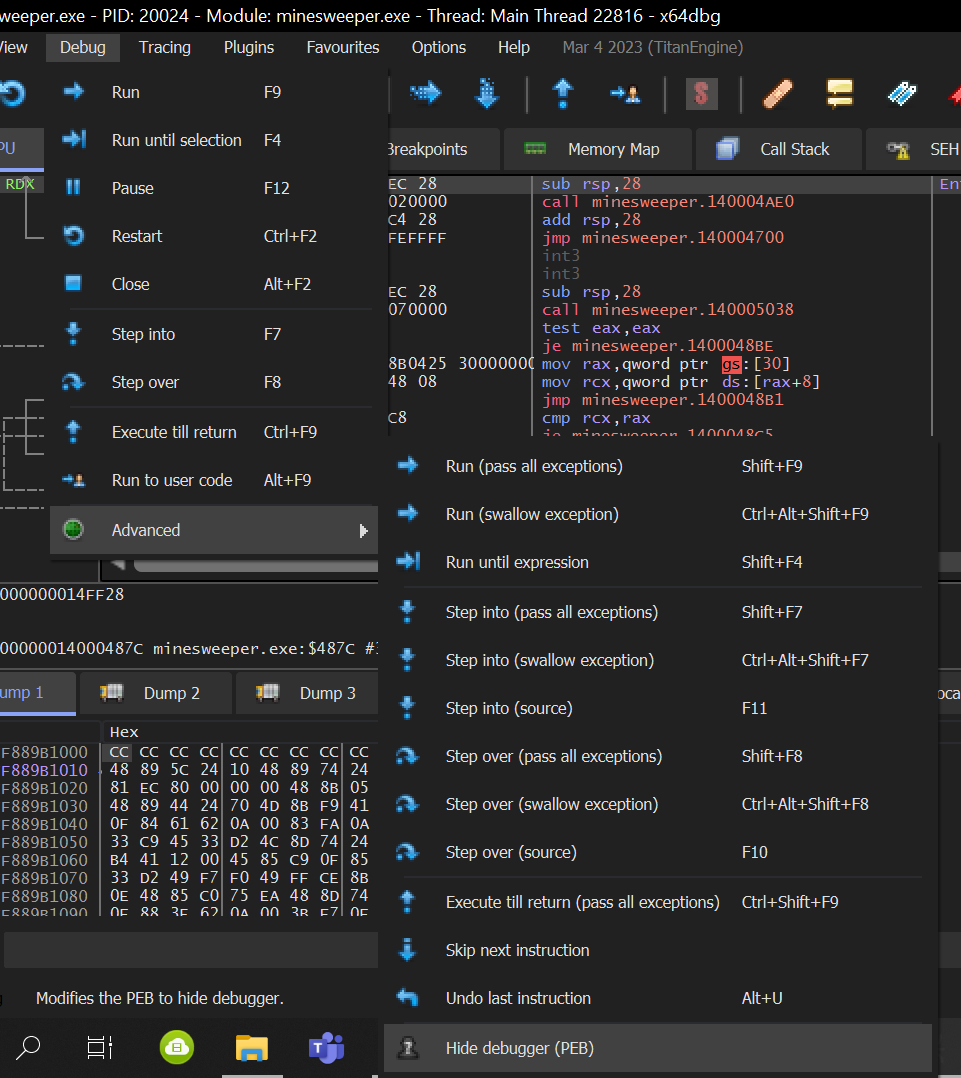
***sub\_1400014C0()*** → ***antiDebugger()***;

***sub\_140001980() → draw\_interface()***;

***sub\_140002400() → make\_interface\_pretty\_and\_ set\_initial\_position\_cursor()*** (clear the console screen by setting the cursor position to different coordinates and outputting space characters. The final call seems to be setting the cursor position to a particular location on the console screen.); ***sub\_140001C20() → implement\_difficulty\_level\_menu()*** (simple interactive menu for choosing a difficulty level, with visual feedback for the selected option)

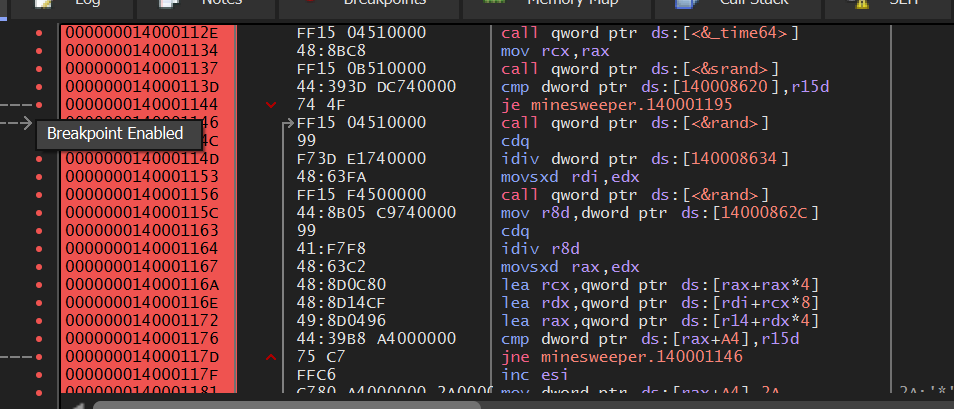
This part of code seems to be the place where the bombs are put on the board. It initializes the random number generator, checks the number of mines to place, and loops until the desired number of mines is reached.

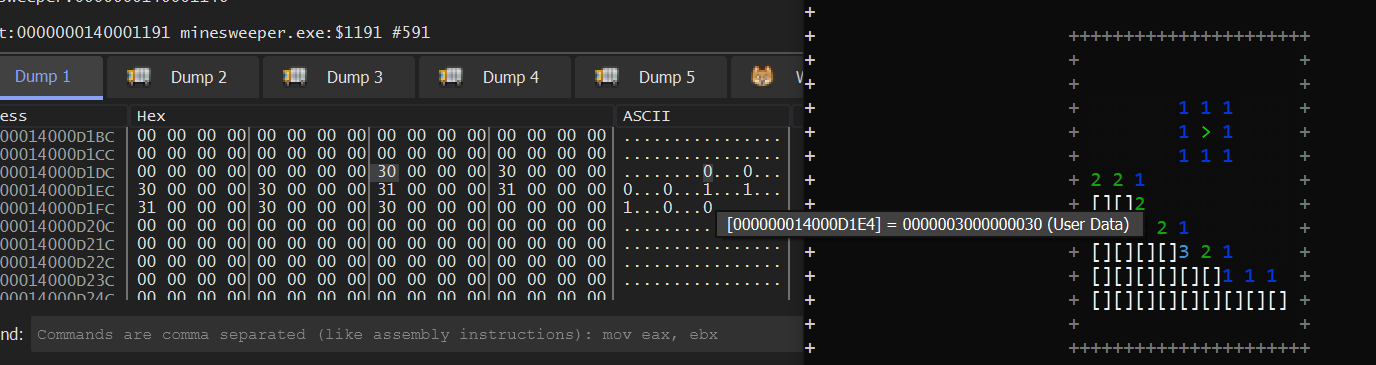


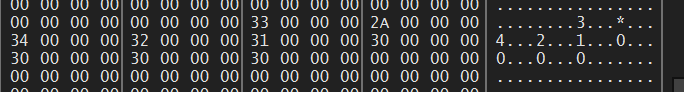


Before each execution, in order to bypass the antiDebugger.

After setting some breakpoints (follow in the dump) and running in order for all the bombs to be place, we can see the matrix:







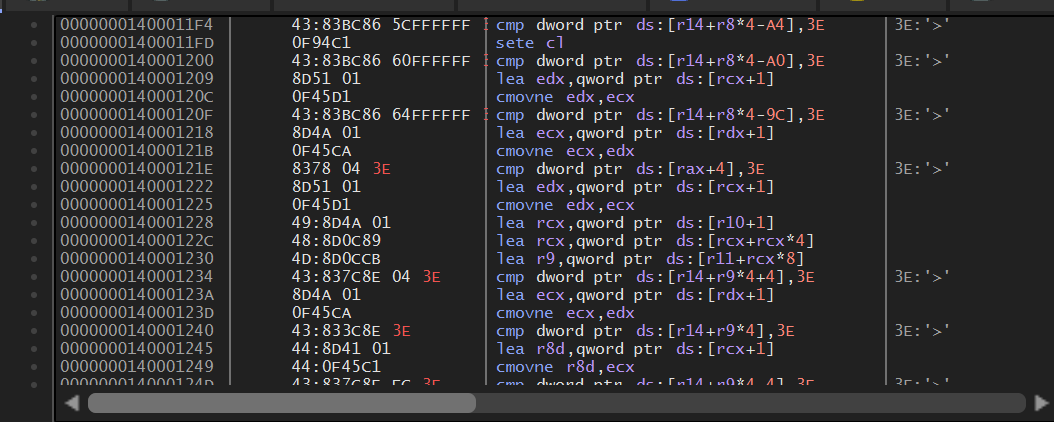
***0xD1E4*** is the starting address of the matrix:

* 0 (empty space): 30
* 1: 31(2: 32; 3: 33; 4: 34.....)
* \* (bomb): 2A
* > (flag): 3E

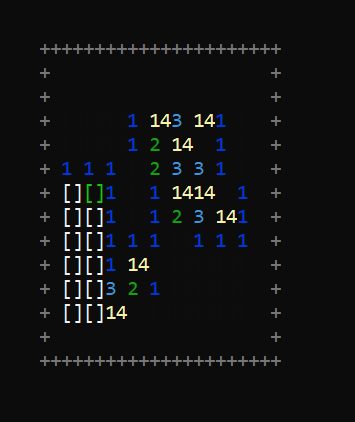
***0x1181*** places the bombs. (Right click, Binary, Edit, ...)



If we scroll a little bit down, we will see a lot of 2A. So we change all of them with 3E:



This is what we get:

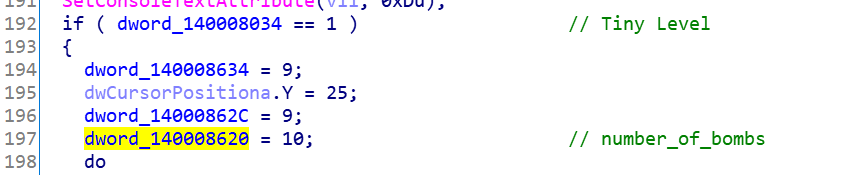
 

It is not exactly what we want. The game will never end, thou, even if you click on a bomb, instead, the value 14 will be displayed (***bombs\_not\_working.exe***).

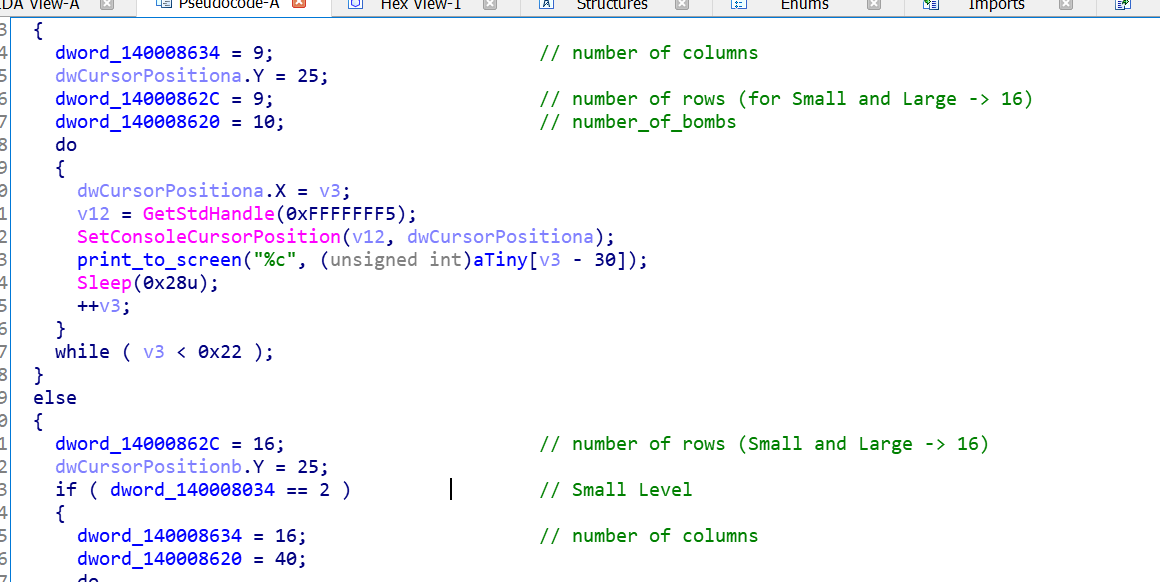
The issue is most likely coming from *sub\_140003100()*: the interface prints ***[]*** as the matrix (there are actually 2 matrix: one in memory, where the bombs are, one in the console) (photo above).

* *Change the game such that all positions have a bomb. (20p)*

In ***sub\_140001C20()*** (renamed: ***implement\_difficulty\_level\_menu()***), we can observe the variable ***dword\_140008034***. Depending on the choice of the user, this variable will become 1 (Tiny), 2 (Small) or 3 (Large), depending on the user’s choice for difficulty.



For each level of difficulty, ***dword\_140008620*** (which is the number of bombs) gets a new value: 10 (Tiny), 40 (Small), 99 (Large):



So: Tiny → 81 cells (9x9) (from 0xA to 0x51); Small → 256 cells (16x16) (from 0x28 to 0x100); Large → 480 cells (16x30) (from 0x63 to 0x1E0). These are the new number of bombs.

**Tiny:**





**Small:**





**Large:**

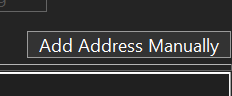


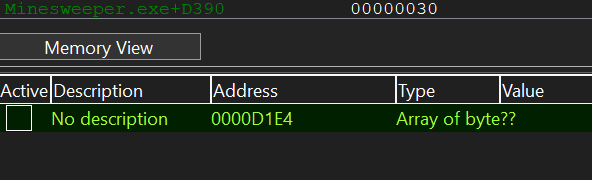


Right click, Patches, Patch File, ***all\_bombs.exe***

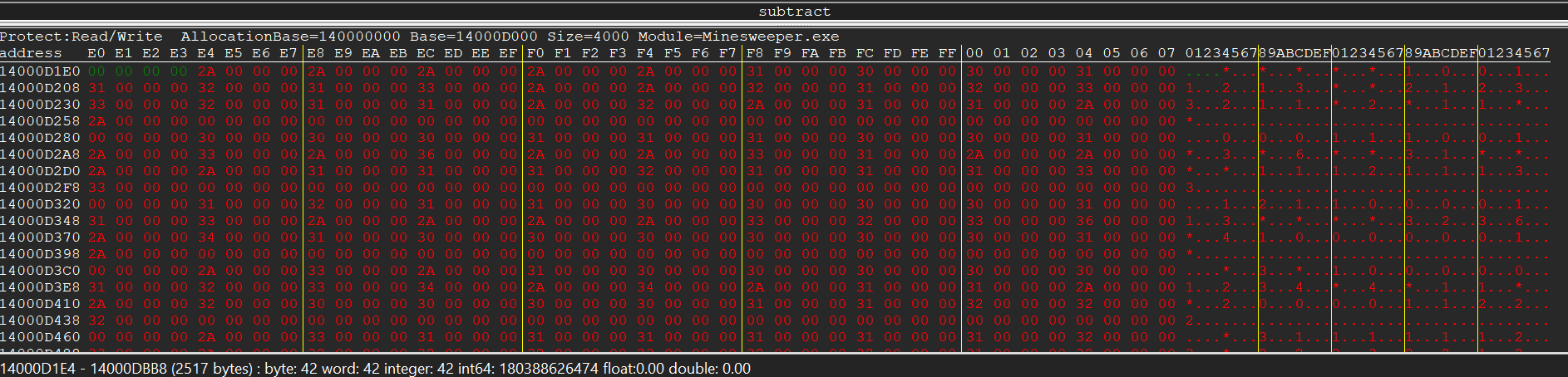
* *Write “RE” on the board using bombs, only on the highest level of difficulty. (20p)*

After some reasearch, I stumble upon Cheat Engine. We select a PROCESS to open (Minesweeper.exe). Board is between address: 0xD1E4 – 0xDBB8





Click on Memory View:



Copy the value and we need to modify it manually. (... dar nu mai am timp =))))