# [Reversing and Exploiting with Free Tools: Part](https://knowledgezone.helpsystems.com/display/PL/Reversing+and+Exploiting+with+Free+Tools%3A+Part+4) 18

It is good to pause and before continuing with deeper exercises, we will see the topic of writing scripts with free tools, in this case, how to write scripts for IDA FREE.

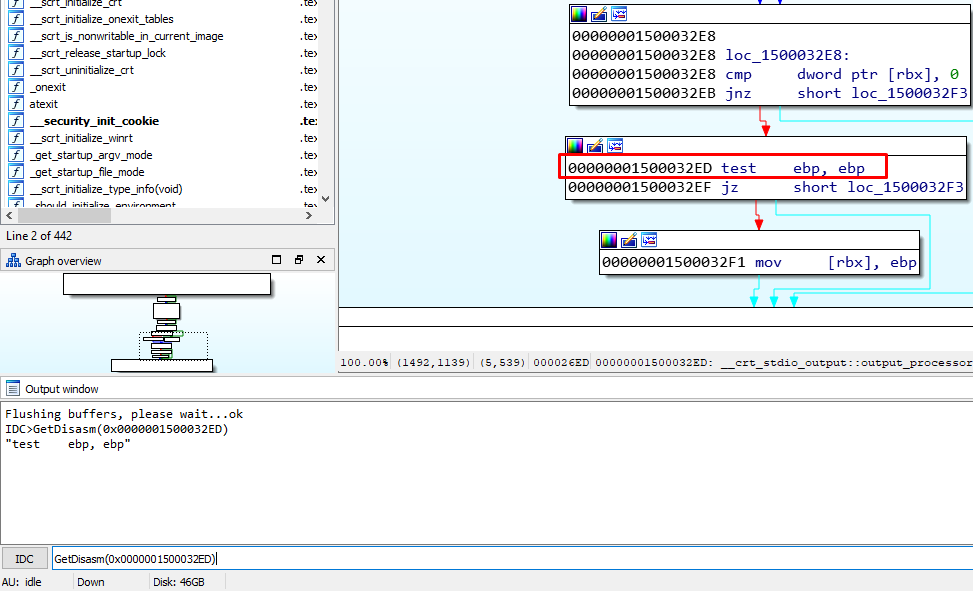
We all know that IDA PRO includes IDA PYTHON to be able to write and execute scripts, only that using free tools we do not have that possibility, but taking advantage of what we have, which is quite a lot. IDA FREE provides support to create and execute scripts with IDC, which is a sort of C, but it works to be able to create scripts for IDA FREE and PRO

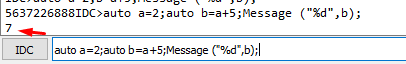
.

IDC SCRIPTING

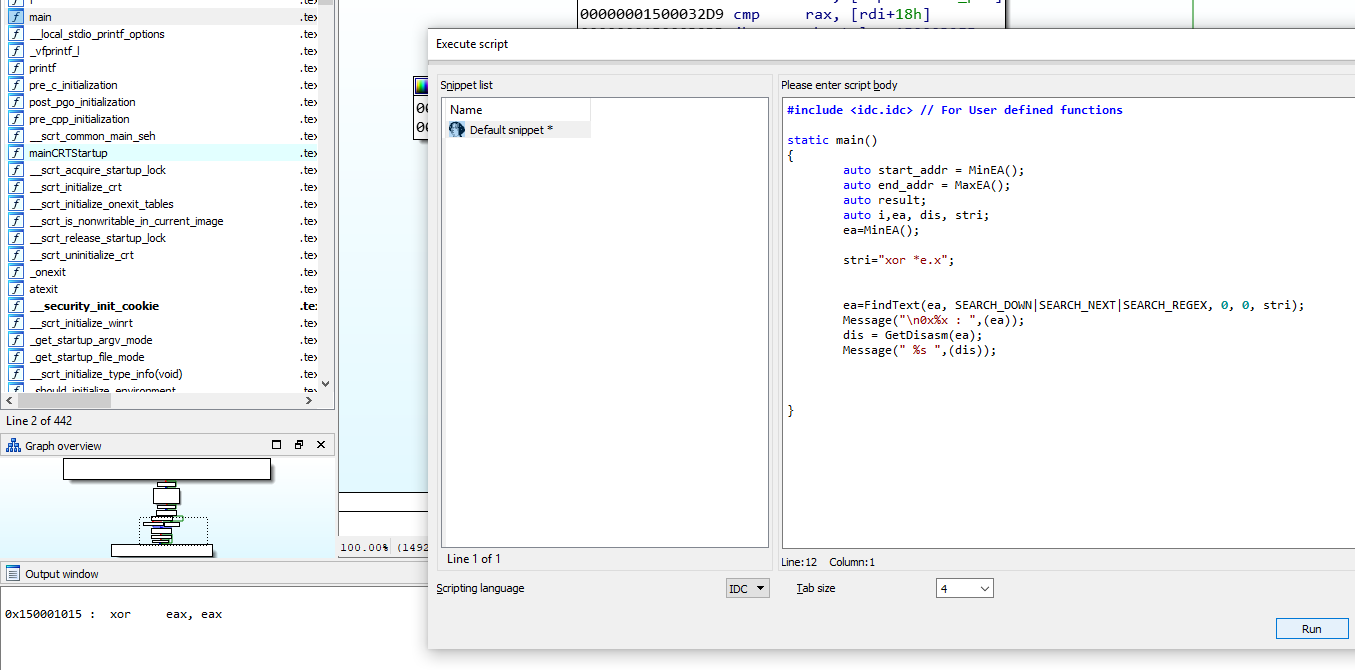
We can execute scripts that we write in idc files, with the menu option FILE-SCRIPT FILE.

We also have the IDC bar below, where you can run single commands or multiple commands, separated by semicolons.

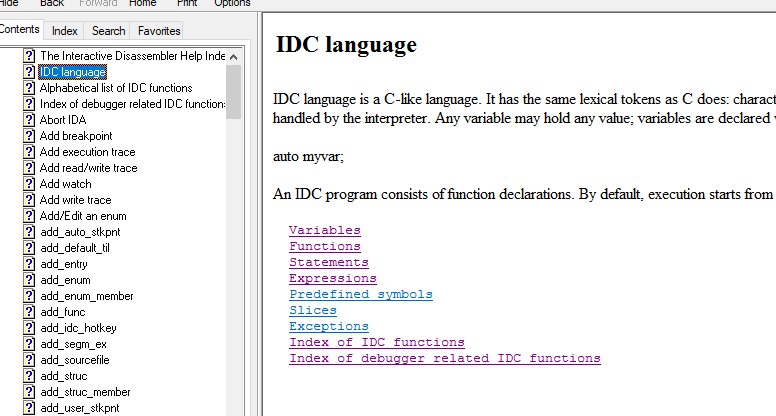


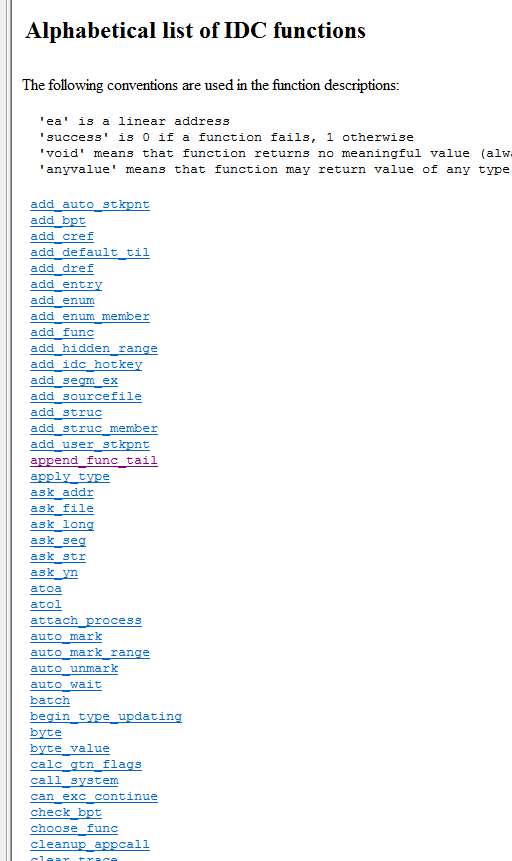


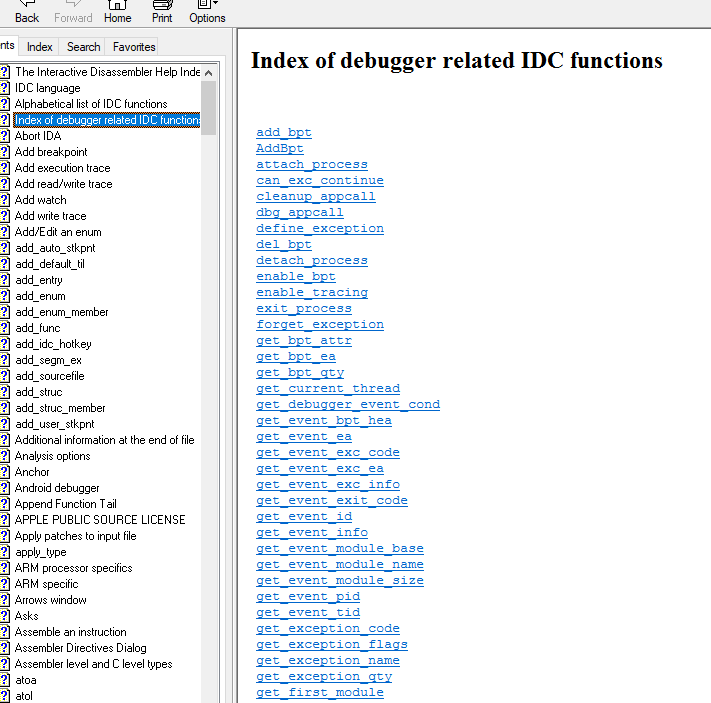
Obviously, the bar is not very comfortable, but we have a third option to test small code snippets or scripts.



Let us begin. A list of functions that can be used in IDC can be found in IDA HELP.



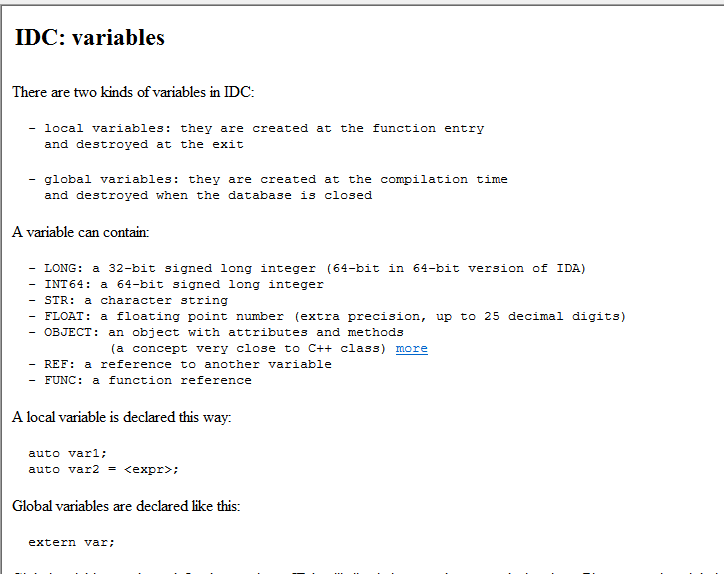




We see that it has many functions, some to use when debugging, others can be used without debugging, it has many possibilities.

Well, let's see what "almost C" is like, which is this IDC language.

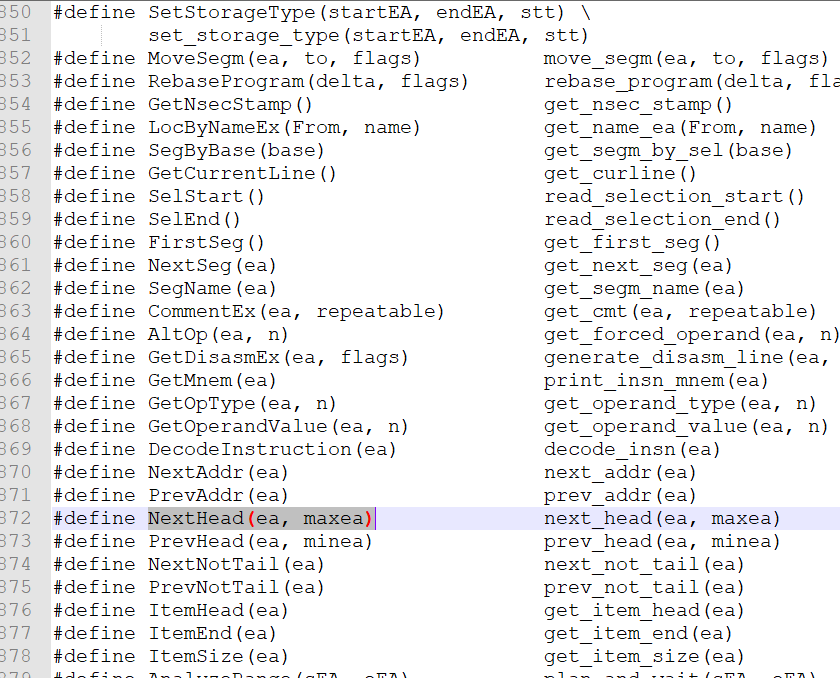
One of the main differences is that it is not a language where you must declare different types. The generic type is "auto" and is used for all local script variables.



We can see the type of content of each variable. Global variables are declared with "extern" and not much else, obviously it is a very simple scripting language only.

In addition to what the documentation says, it is a good idea to have the idc.idc file open in a text editor.

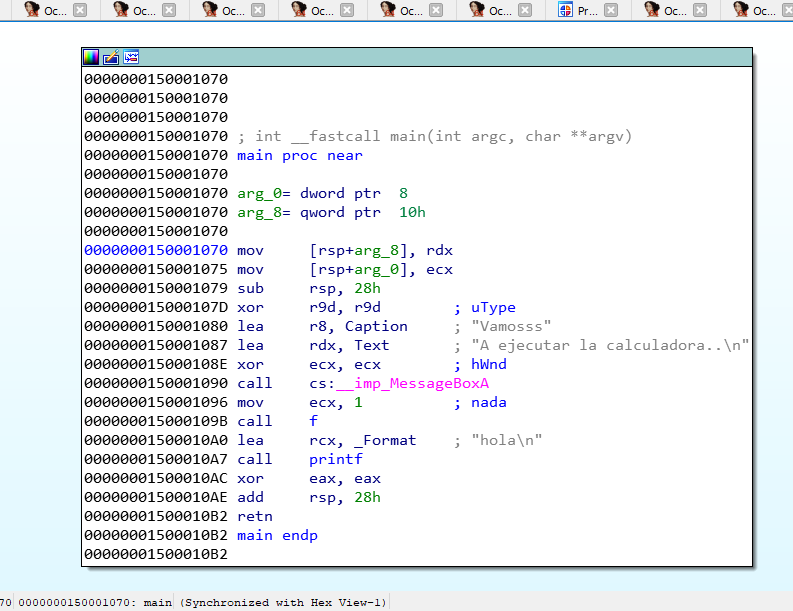
C: \ Program Files \ IDA Freeware 7.0 \ idc

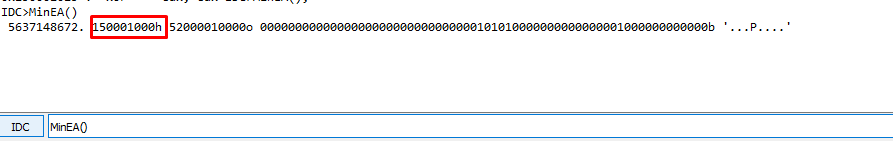


It has functions that are not in the list, and can help us a lot.

We are going to know and execute some functions, to get used step by step to the IDC.

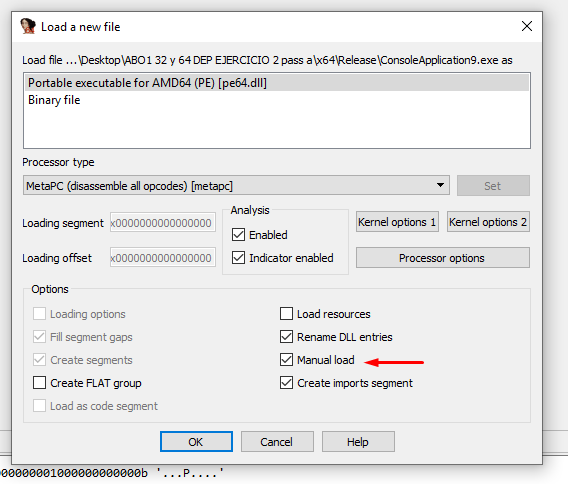
To test and work all together with the same executable, we will use the one from the last 64-bit exercise.

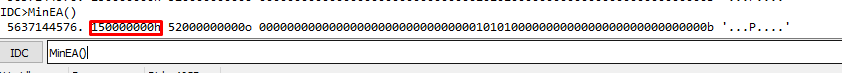




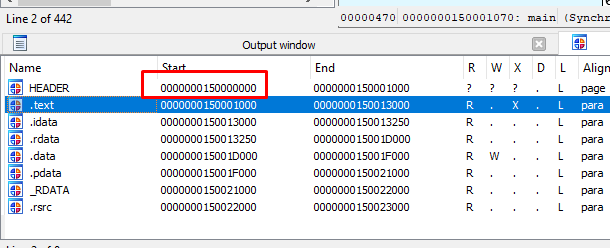
We see that the MinEa() function returns the address corresponding to the beginning of the first section we have loaded. In this case it corresponds to the code section, since the default header is not loaded.

If I reload the executable with MANUAL LOAD, overwriting the old database and agreeing to load all sections, including the header, then MinEa () will return the imagebase to me.





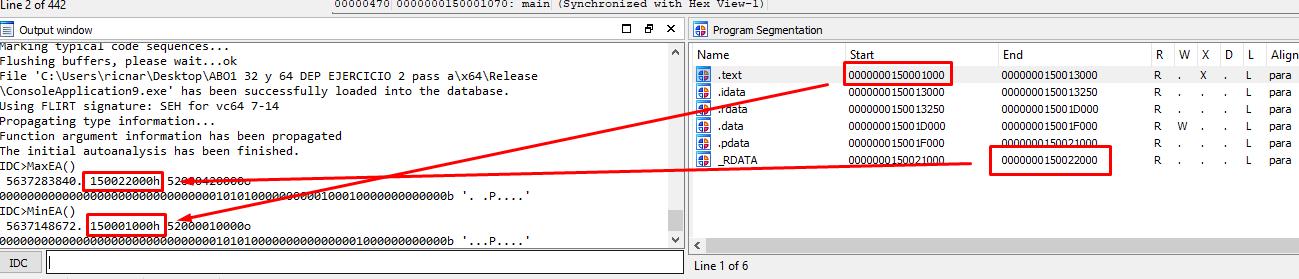
With VIEW-OPEN SUBVIEWS-SEGMENTS we can see all the loaded sections.



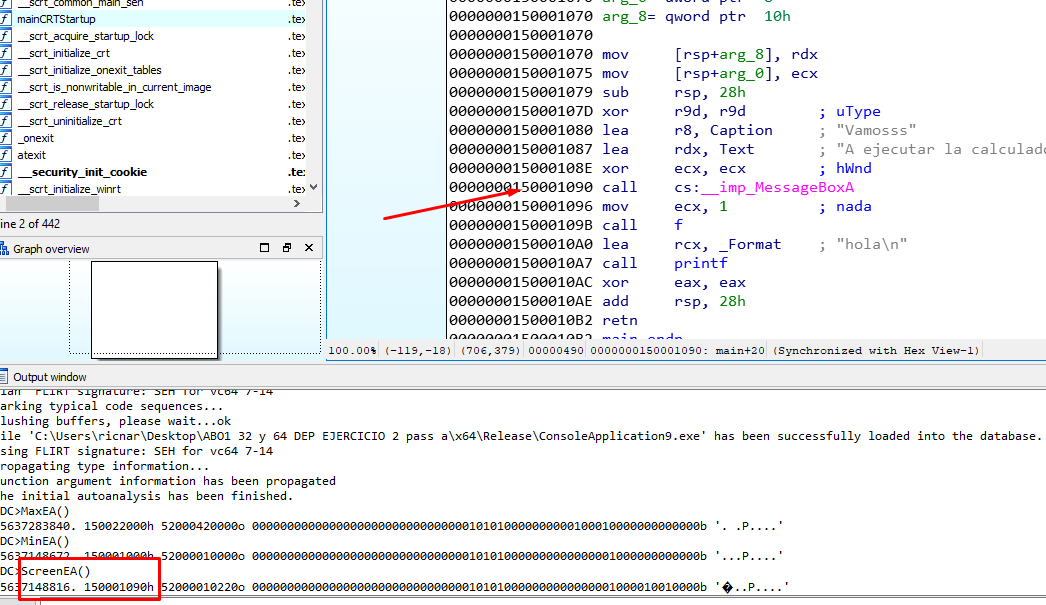
Of course with MaxEA () we will see the end of the last loaded section.



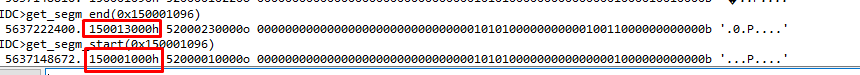
I will reload it normally without loading the header.



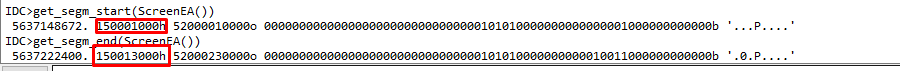
We see that it always shows the beginning of the lowest loaded section and the end of the last loaded section.



ScreenEa () shows us the address where the cursor is.



These functions are quite similar, but we must pass them an address, and they return the beginning and the end of the section to which the provided address corresponds. This can be concatenated by looking for the beginning and end of the section where the cursor is.

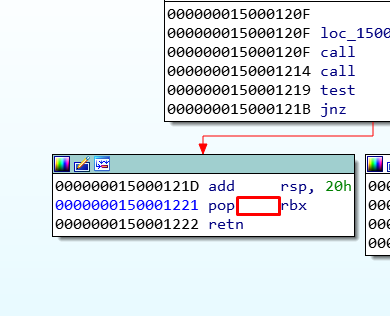


SCRIPT TO SEARCH GADGETS

Well, slowly we will create a homemade script that looks for some gadgets or instructions in the code section.

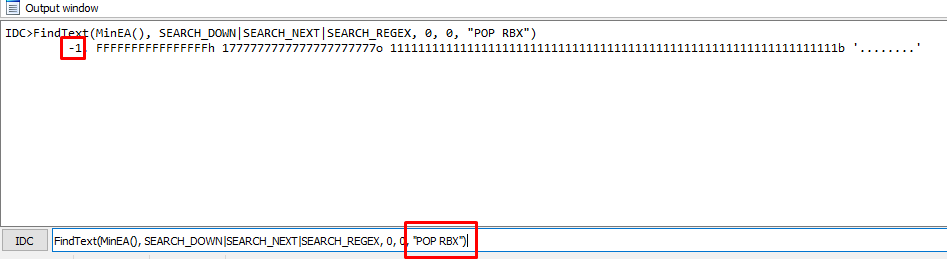
Of course, since we only search for text, we will only find the results of the instructions aligned and detected as such by IDA. We will not be able to find misaligned instructions or look at areas that are not detected as code, but for practice and a quick search it will help us.

We will use FindText () which can search for instructions using REGEX.

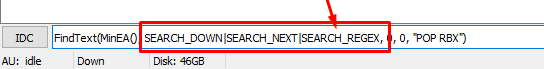


Since we search for text, if we search for example “POP RBX”, I would not find the result that I show above because it has more than one blank space and should write it exactly the same with the same amount of blank space.

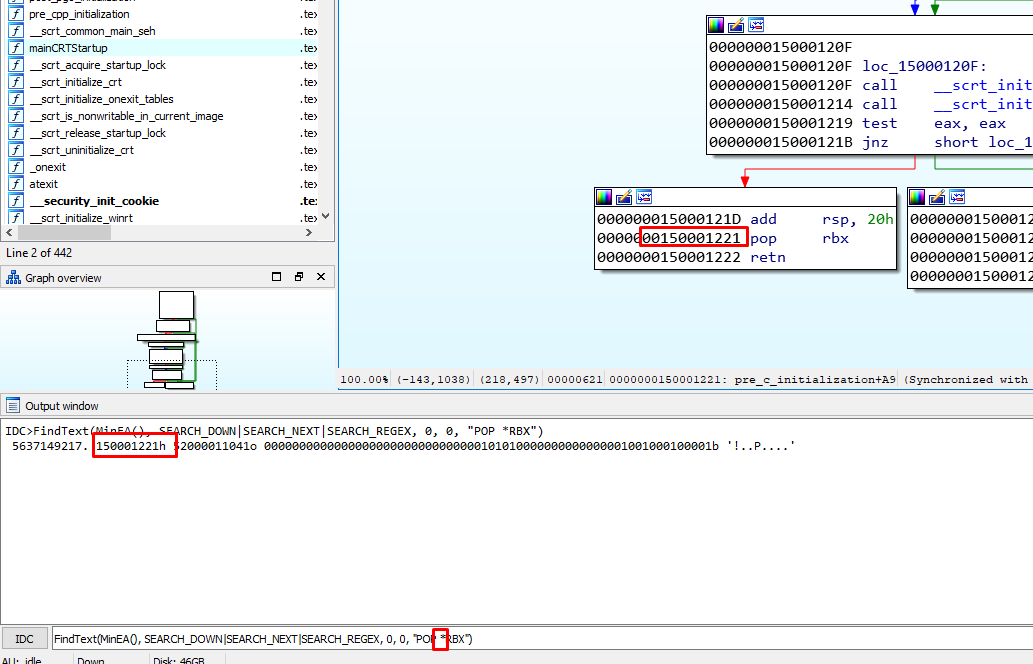
I see you can't find it.(-1 is fail)



I will add the option to search using REGEX.

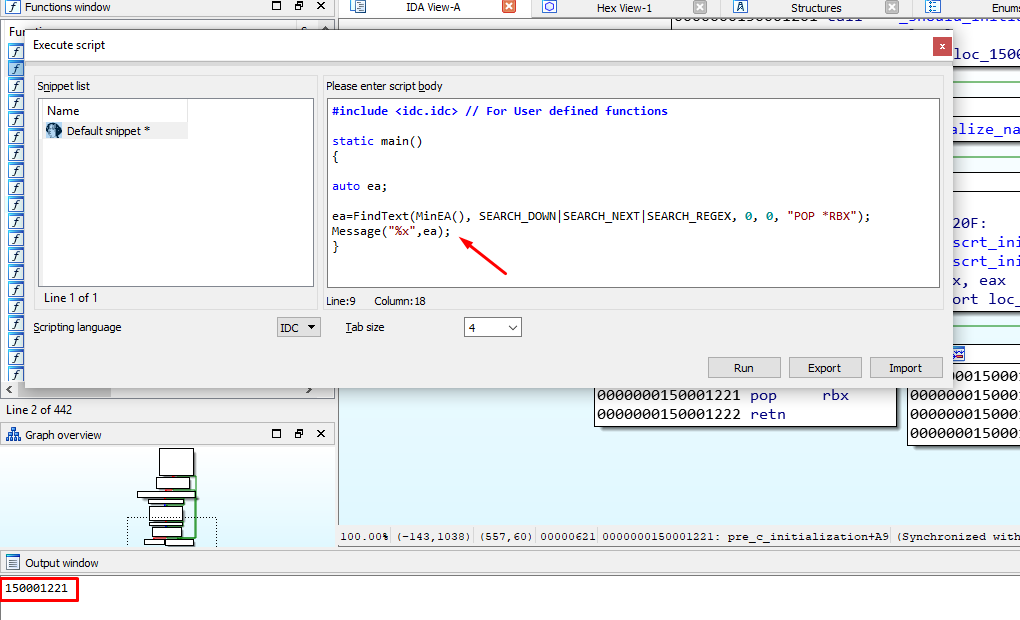


So I can use the asterisk which allows me to match with a string that has repeated the character that is just before the asterisk.



This search will find “**POP EBX”, “POP EBX”, “POP EBX”,** and will match no matter how many blank spaces there are between POP and EBX.

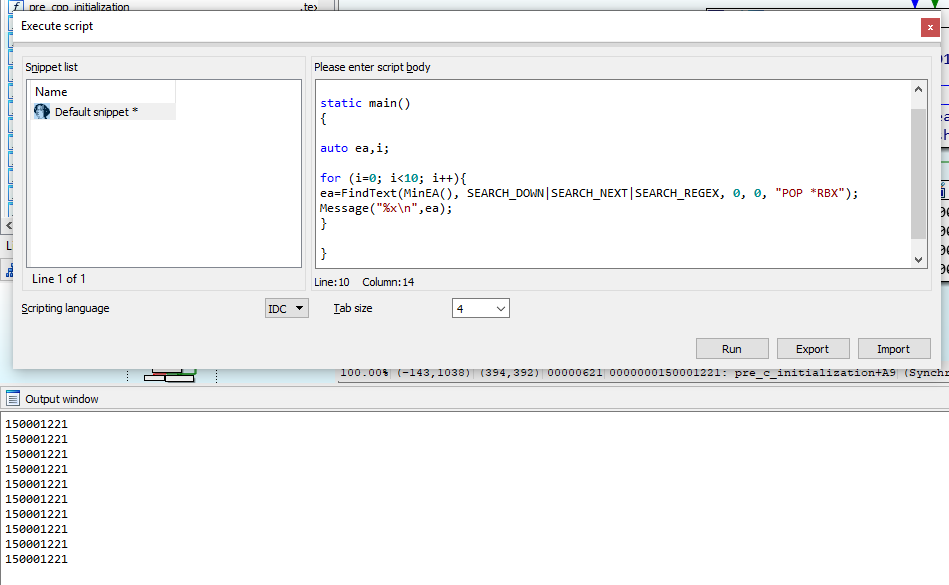
Since I would like to find up to 10 results, I will do it in the command window.



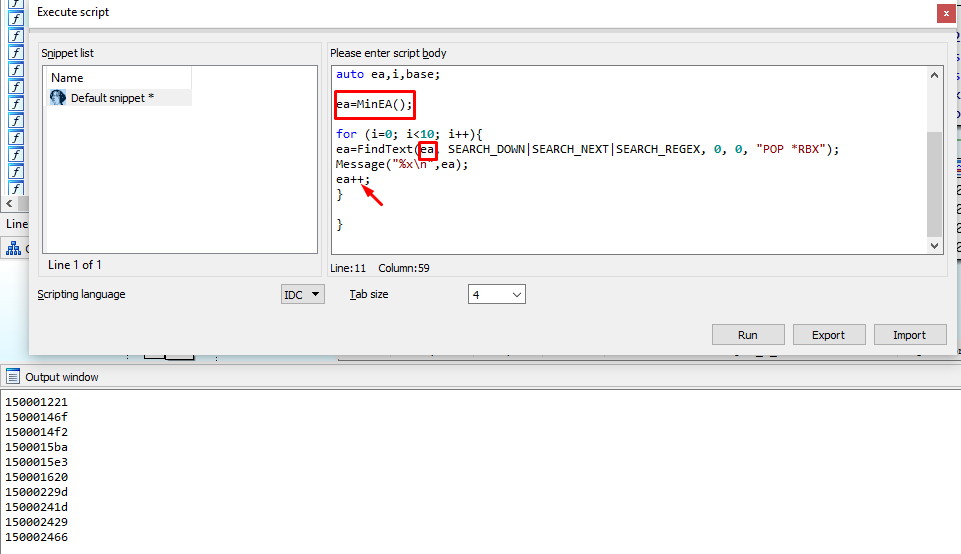
With that code, I find a single address and print it with the Message function, which is similar to a printf.

I declare the local variable "ea" as of type "auto" (don't forget the semicolons at the end of each declaration).

Now, since I want to print 10 results, I will write a loop.



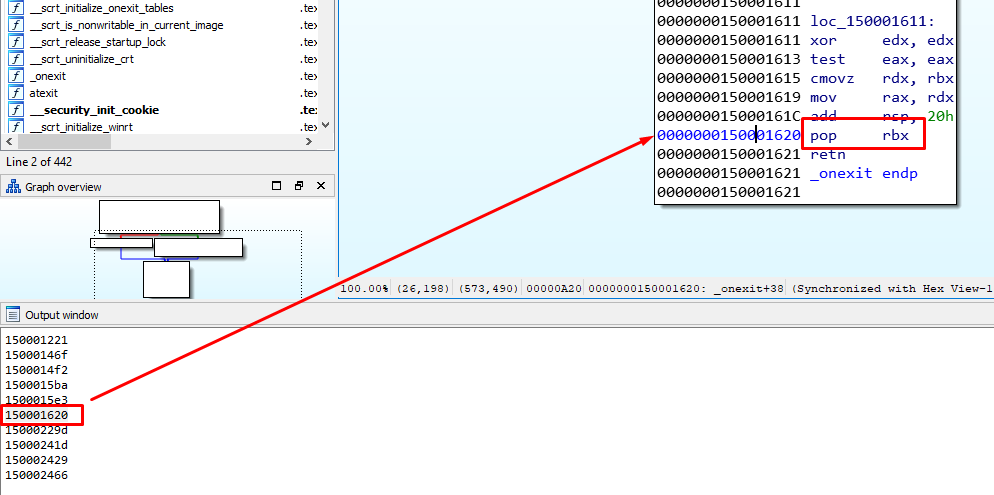
I always find the same result. To avoid this, I must increase the search address after finding a result, so I can search for other different results.



Of course I have to remove MInEA () from the loop, otherwise it will always search from there. I use the variable “ea” that will increase by one every time you search for a new result.

I found 10 results, I will verify some of the new results.

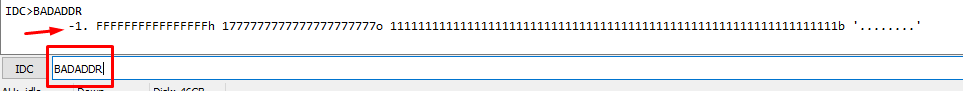
Well, we're going well.



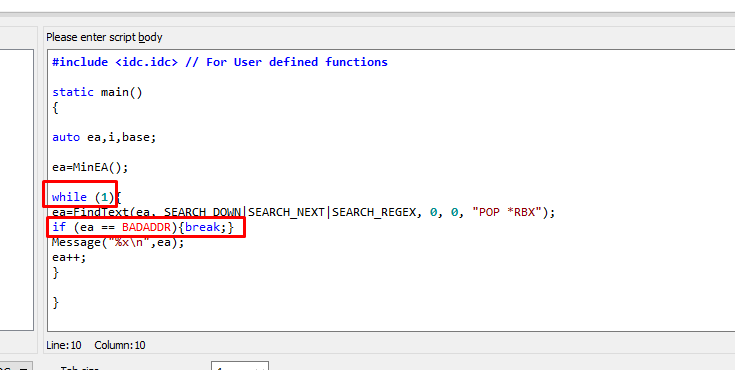
Next thing is to print the instruction I found, but first I will add this check:

**if (ea == BADADDR) {break;}**

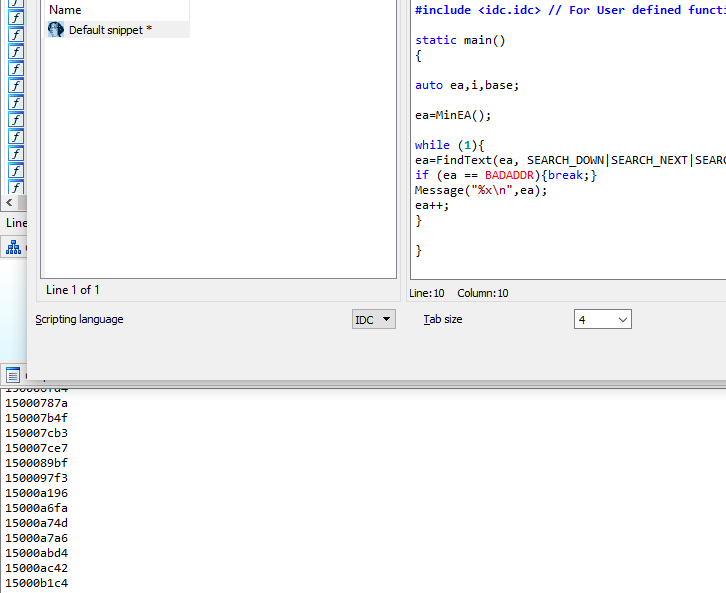
So if **ea = -1** it means that it does not find anything, and break.



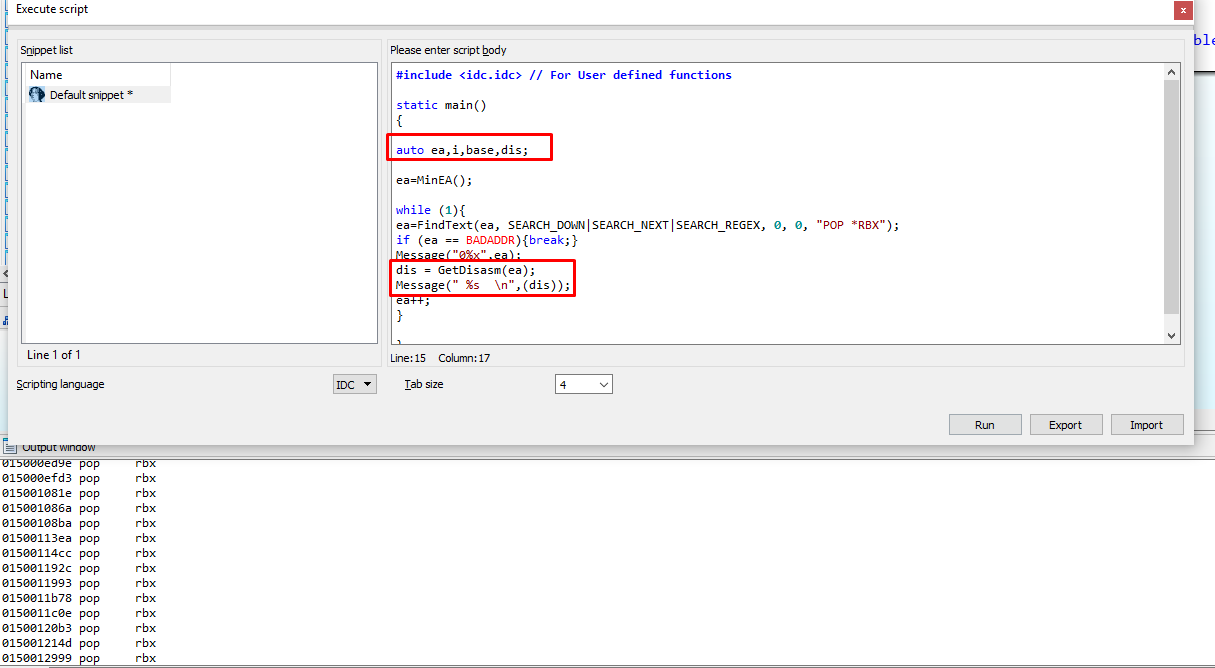
With this code, we could search not only 10 results but all of them.



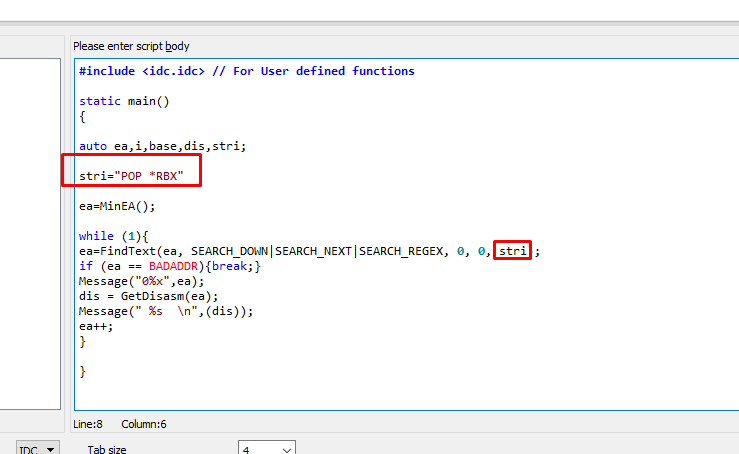
It searches until it returns -1 as many as it can find.



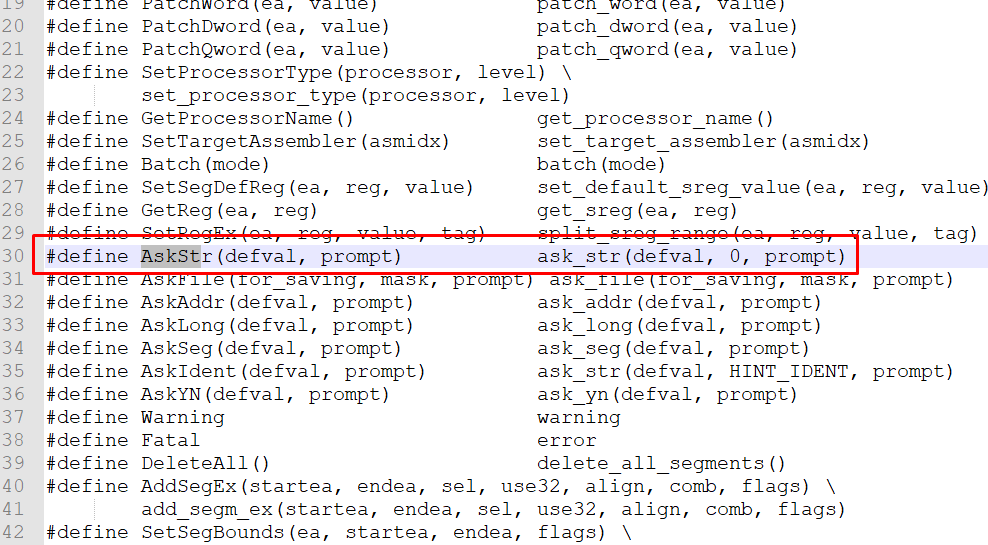
I need to print the statement, I use Get Disasm () to do it as it returns it to me as a string. I make a format string using Message to print it.

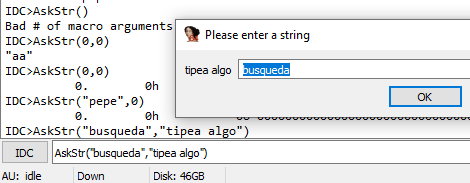


We are improving.



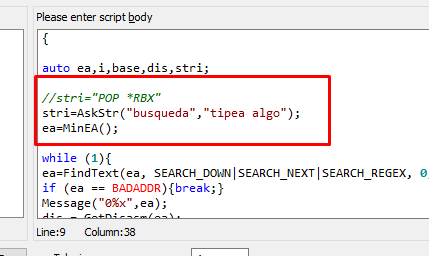
It is convenient to put the string to search at the beginning. You can even ask the user to type what they want to search for.

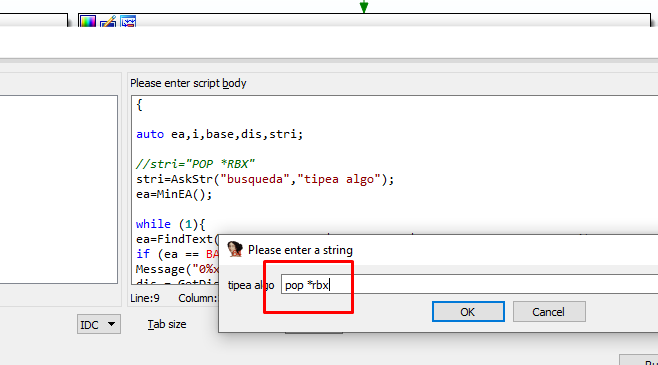




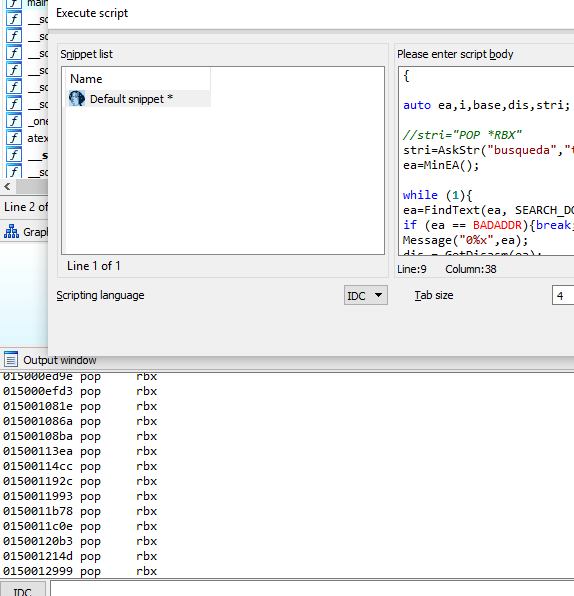
I won't add that for now, let's just let the strings be put in a variable.

Whoever likes it can do it like this.

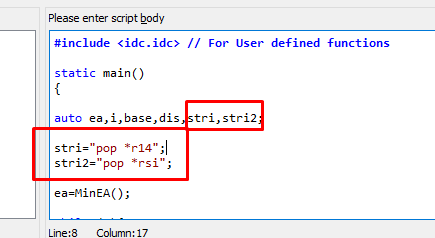




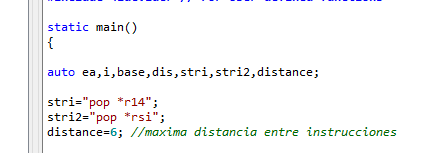
We can see that it works the same way.

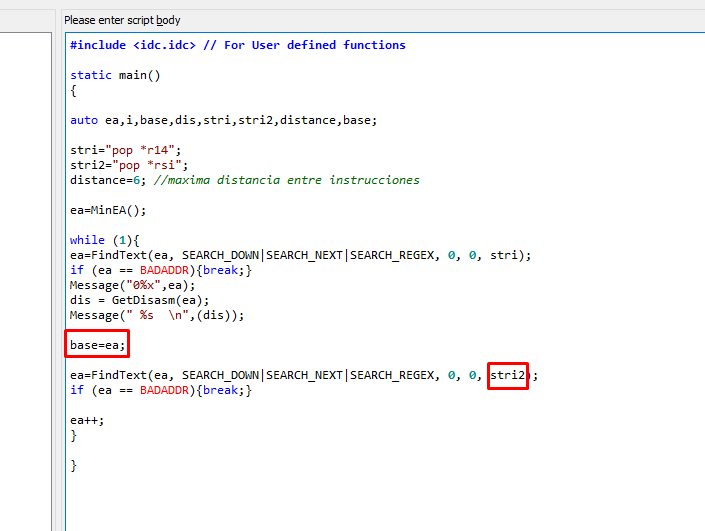


I will add another string.



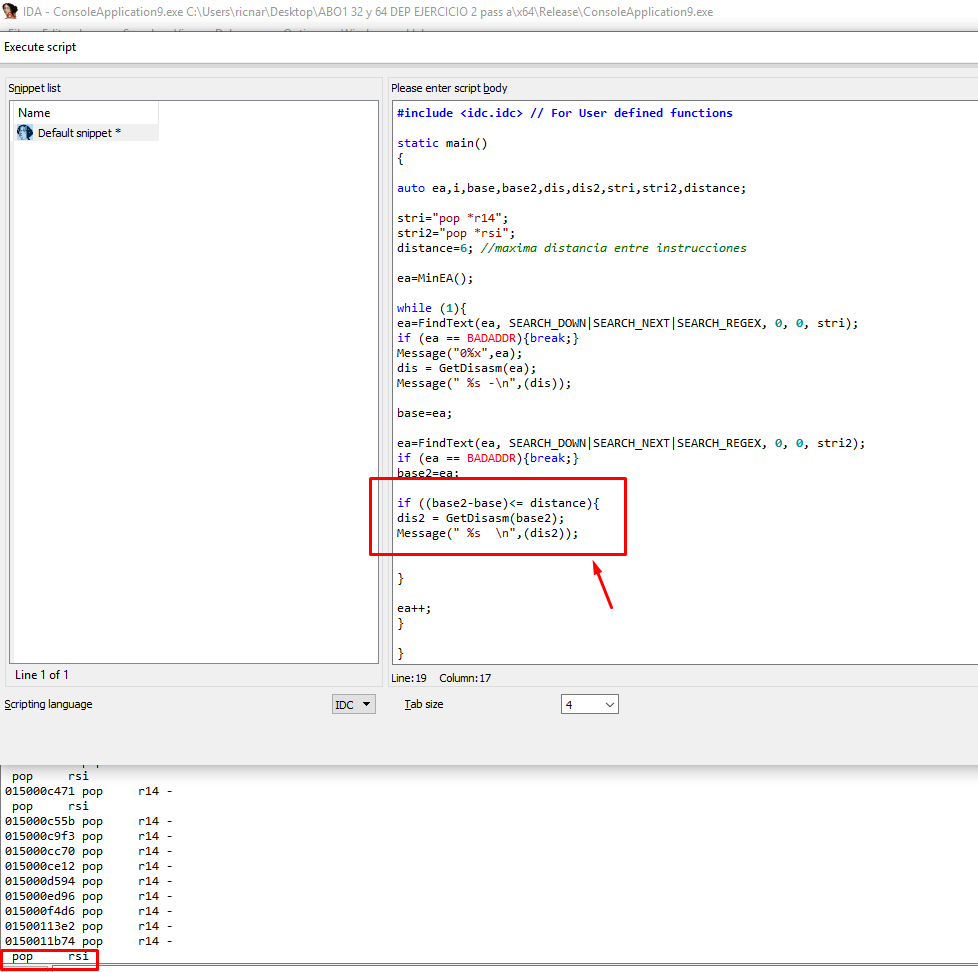
From the direction the first result is in, I will search for the second and there will be a maximum distance between them. If the distance is less than that maximum distance, I accept it.



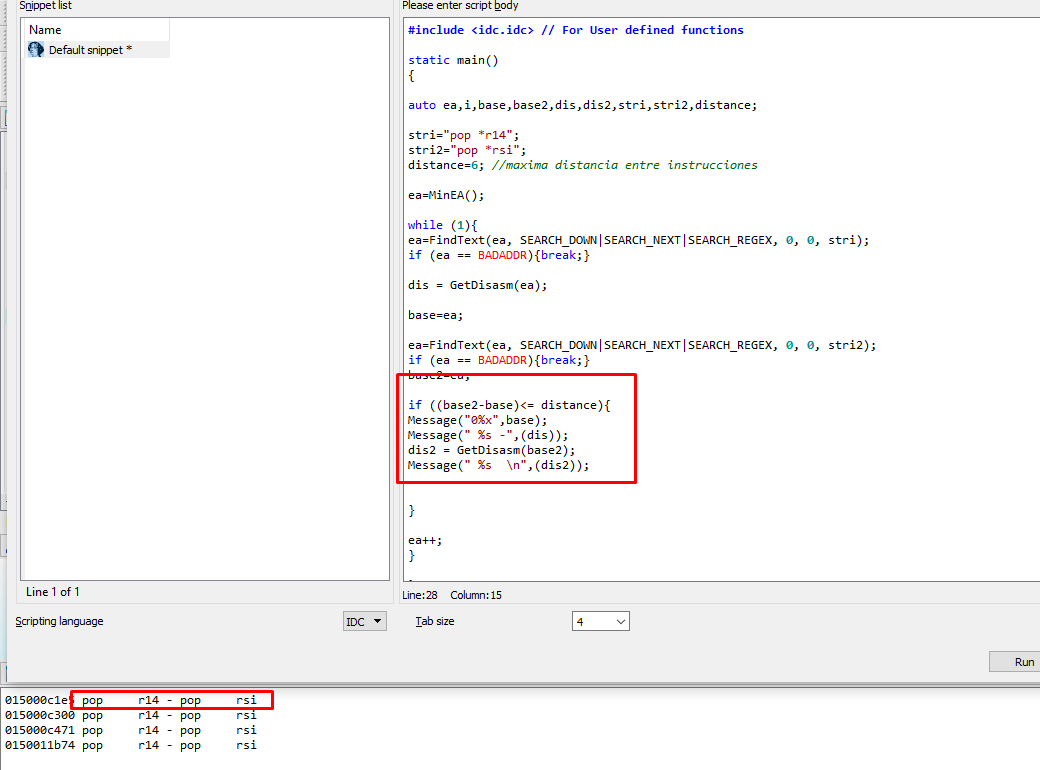


I keep the address of the first match stored in the base variable, and look for the second result by searching from the address of the first one.

If I find a second result, I should check the distance between them.



In some results I only found one POP r14 and cannot find POP RSI. In other results I find both. I'll move the print so that it only prints when I find both.

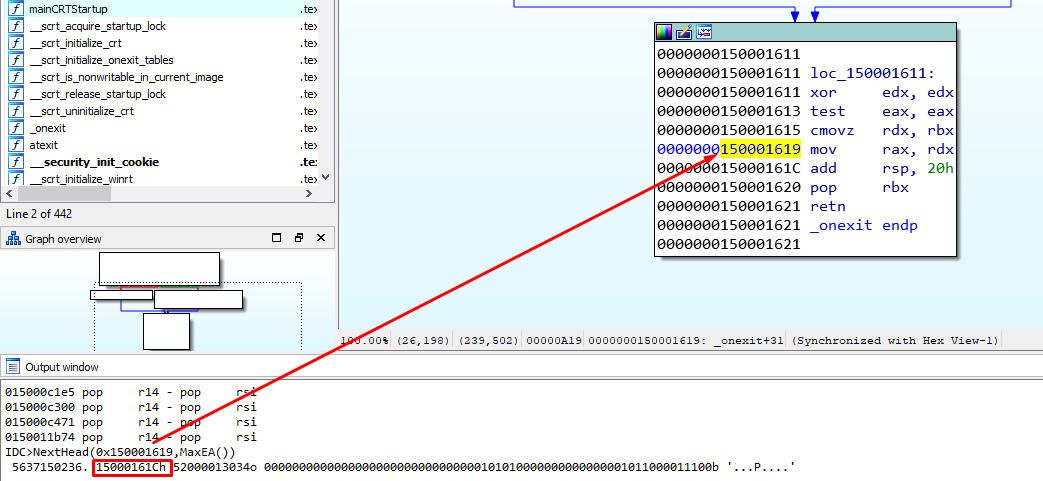


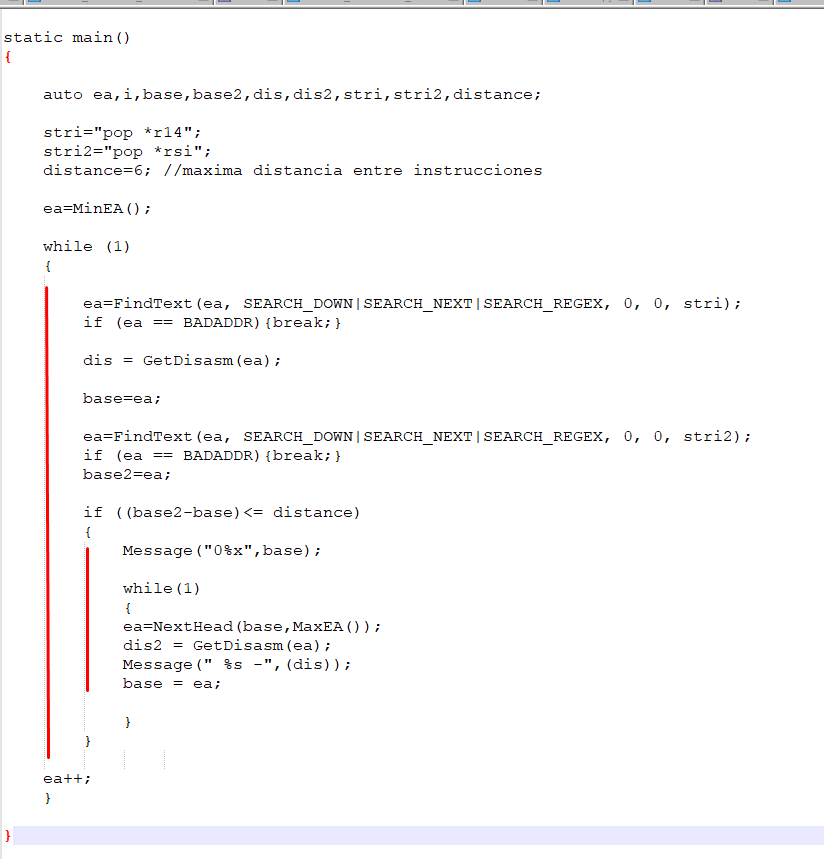
The problem now is that the intermediate instructions between the two results are not printed, since they can be separated from each other. It would be good to print the intermediate instructions.

I will use instruction:

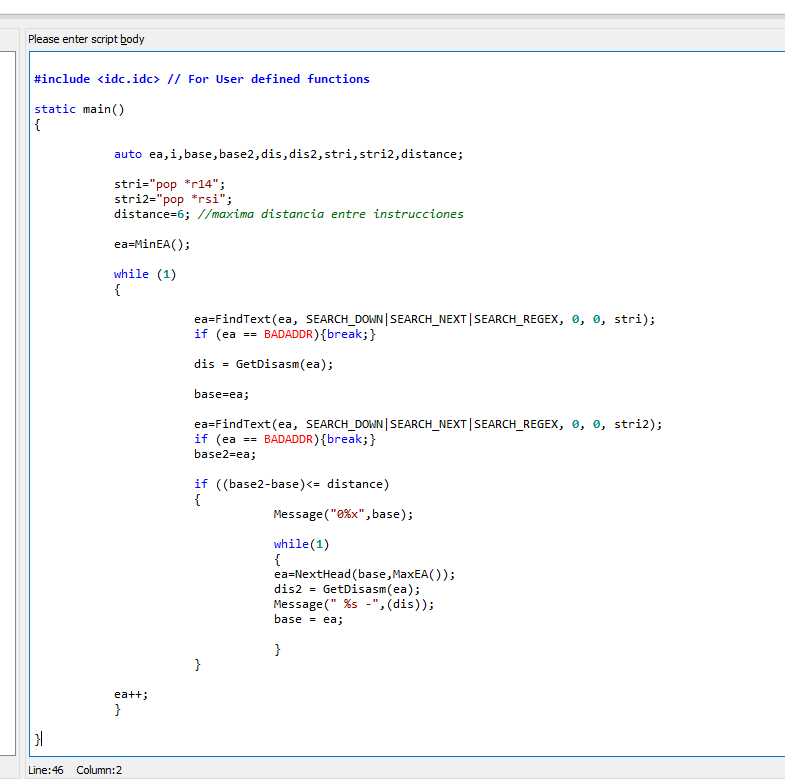
**NextHead (base, end\_addr);**

Which gives me the address of the next instruction to the current one.

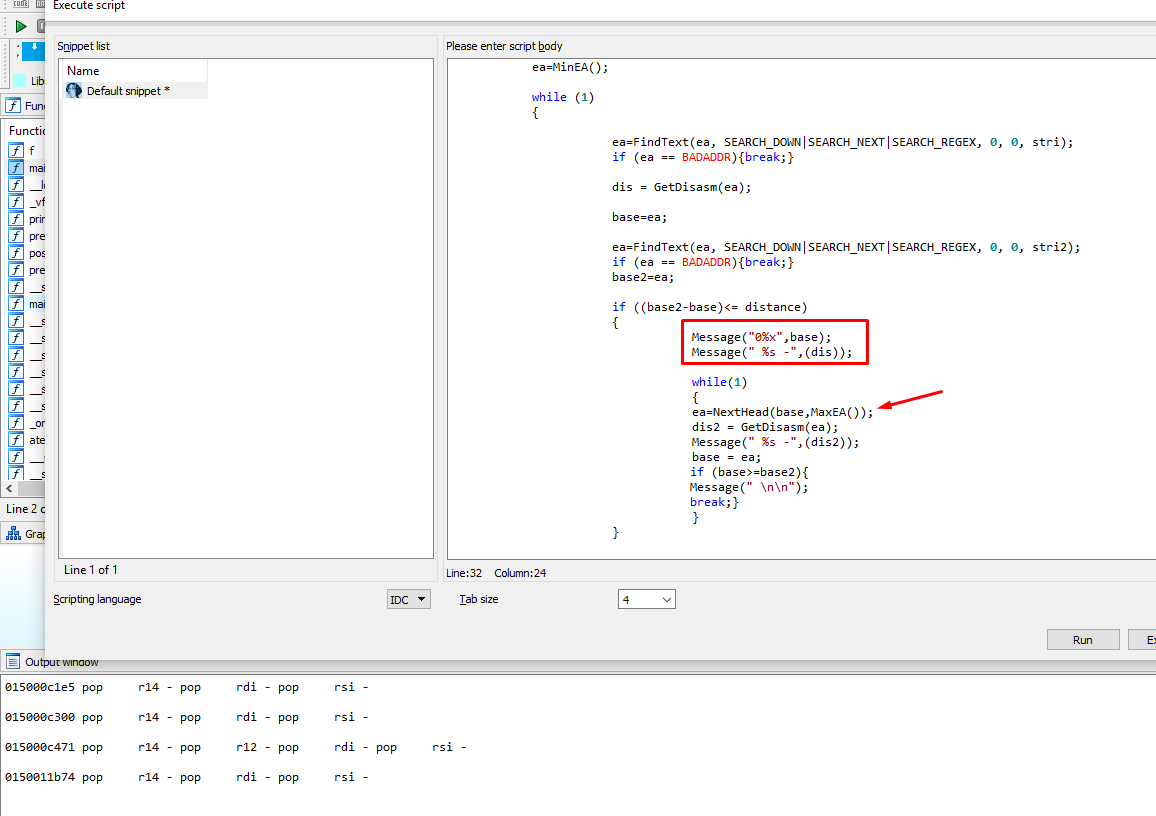




I copied the script to NOTEPAD ++ and aligned it a bit.

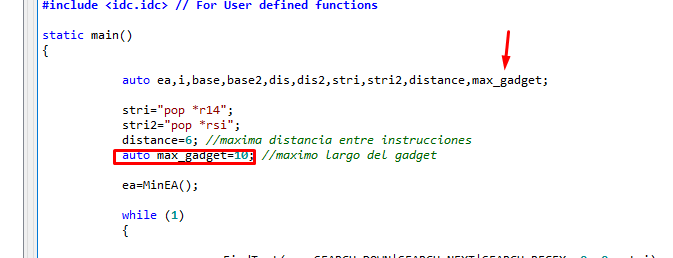


Now I am missing the output condition as long as it prints, if it doesn't continue printing to infinity

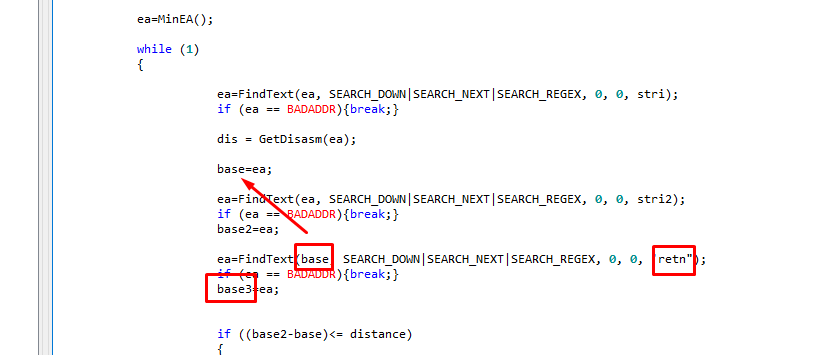


There it is. Now it prints all the intermediate instructions between the two instructions.

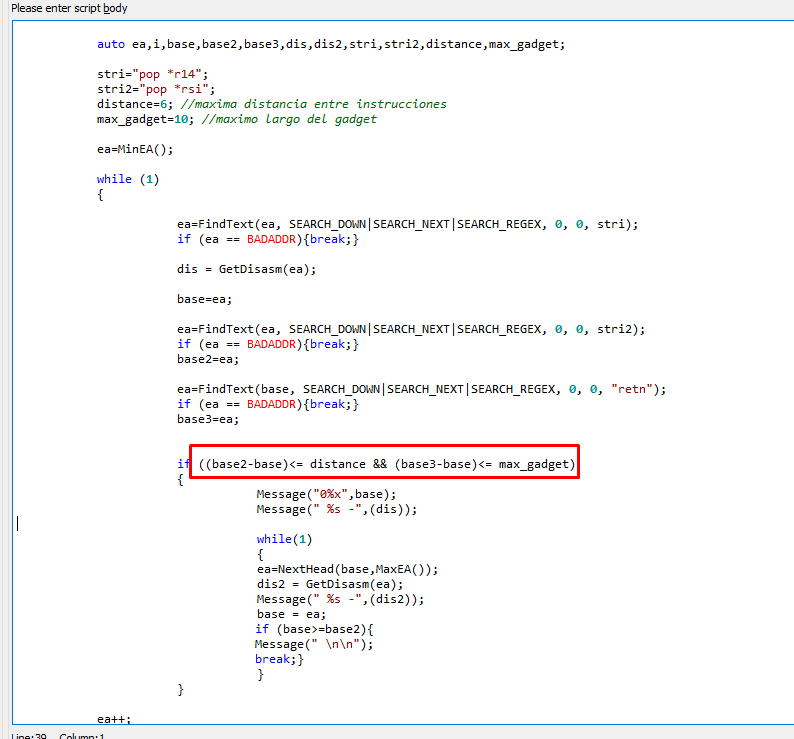
Now I am going to add a maximum gadget length.



So I must also search for a “retn” from the base, that is, from the first match I find.

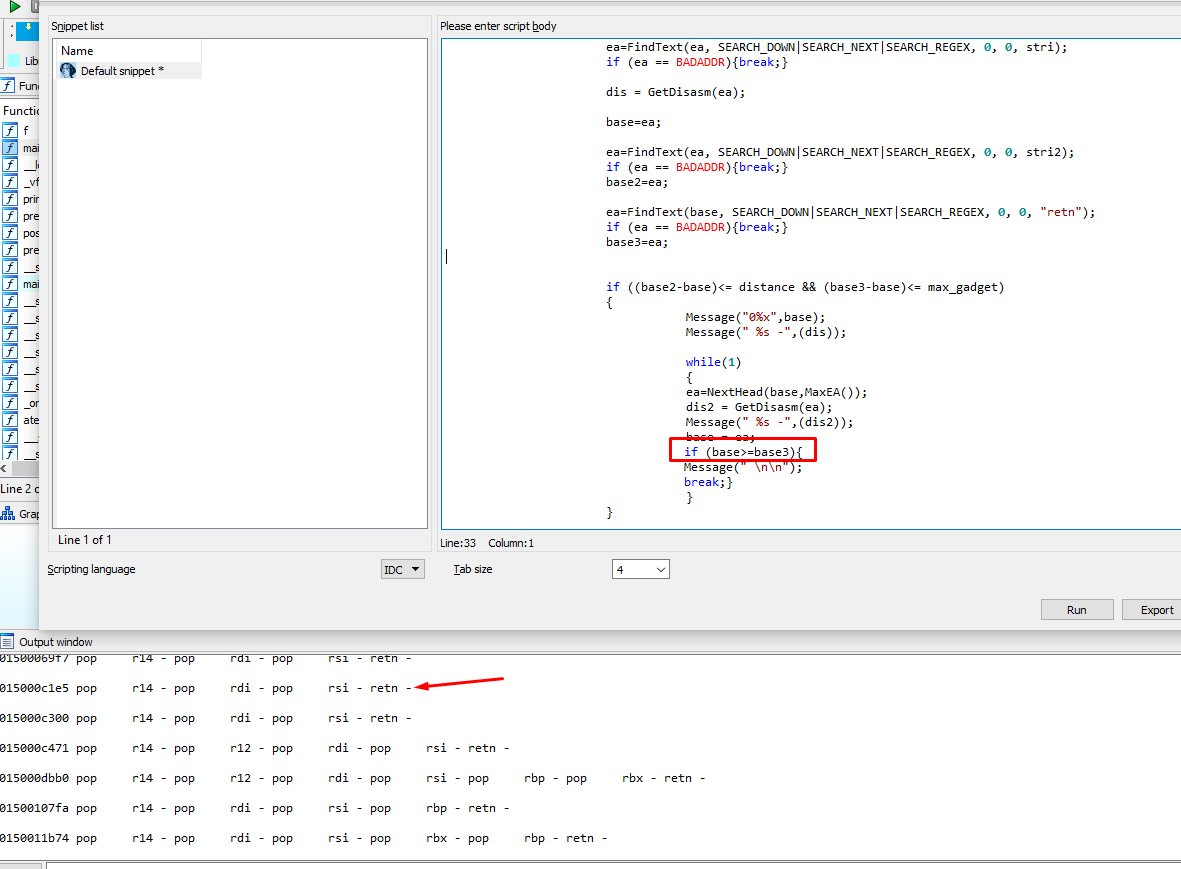


I add the variable base3 and do a search for "retn" from the base and save the result in base3.



So now the new condition is that both instructions are separated by less than "distance", and that there is a "retn" at a distance less than max\_gadget from the base.

Now we must make it print not until base2 but until base3, that is, until retn.



Not too bad, you can improve the script to find more instructions. The important thing here is to practice, here is the final script.

SCRIPT FINAL

#include <idc.idc> // For User defined functions

static main ()

{

auto ea, i, base, base2, base3, dis, dis2, stri, stri2, distance, max\_gadget;

stri = "pop \* r14";

stri2 = "pop \* rsi";

distance = 6; // maximum distance between instructions

max\_gadget = 10; // maximum gadget length

ea = MinEA ();

while (1)

{

ea = FindText (ea, SEARCH\_DOWN | SEARCH\_NEXT | SEARCH\_REGEX, 0, 0, stri);

if (ea == BADADDR) {break;}

dis = GetDisasm (ea);

base = ea;

ea = FindText (ea, SEARCH\_DOWN | SEARCH\_NEXT | SEARCH\_REGEX, 0, 0, stri2);

if (ea == BADADDR) {break;}

base2 = ea;

ea = FindText (base, SEARCH\_DOWN | SEARCH\_NEXT | SEARCH\_REGEX, 0, 0, "retn");

if (ea == BADADDR) {break;}

base3 = ea;

if ((base2-base) <= distance && (base3-base) <= max\_gadget)

{

Message ("0% x", base);

Message ("% s -", ​​(dis));

while (1)

{

ea = NextHead (base, MaxEA ());

dis2 = GetDisasm (ea);

Message ("% s -", ​​(dis2));

base = ea;

if (base >= base3) {

Message ("\ n \ n");

break;}

}

}

ea ++;

}

}

Of course they can export it if they don't lose it, and save it to a file and run it from the menu as many times as they want.

I think it is a nice idea to practice, improve this script and add more things to it, it is a basis for you to have fun for a while with IDC which is not as terrible as they paint it out there.

Until part 19.

06/02/2020

Still in 40tena :-)

Ricardo Narvaja