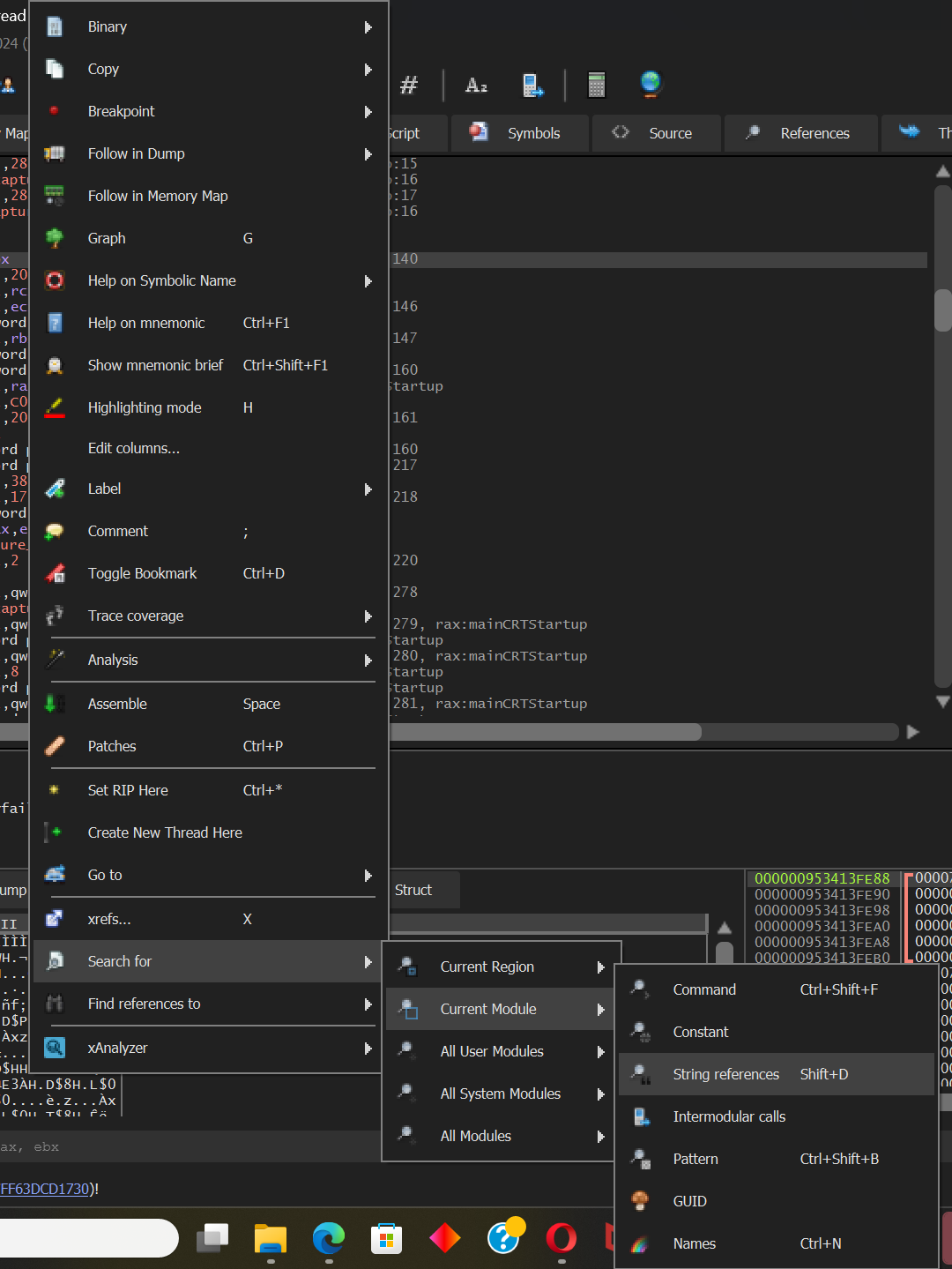
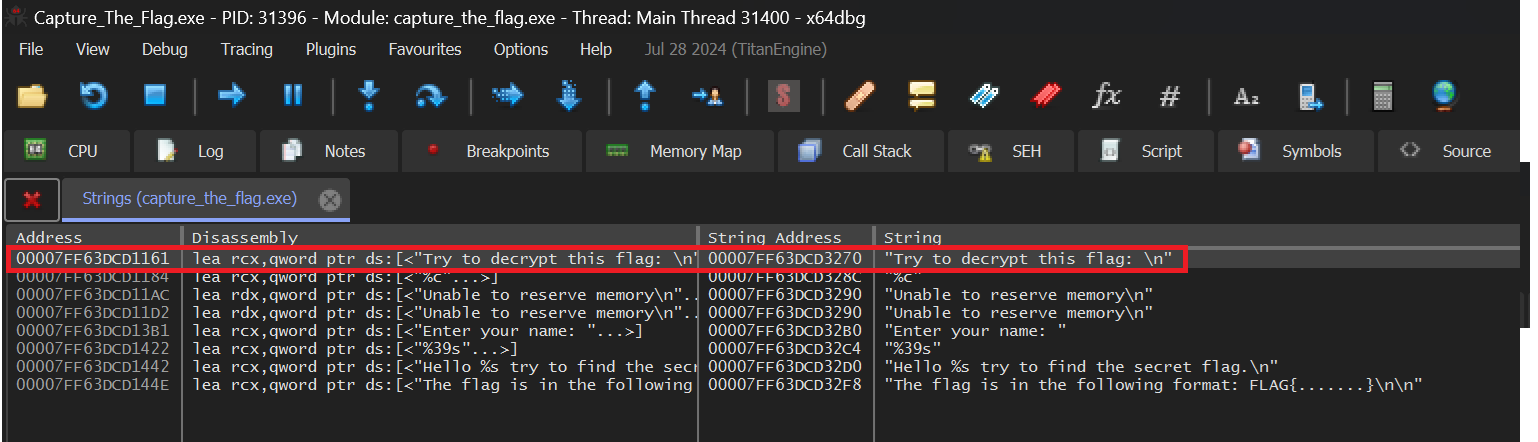
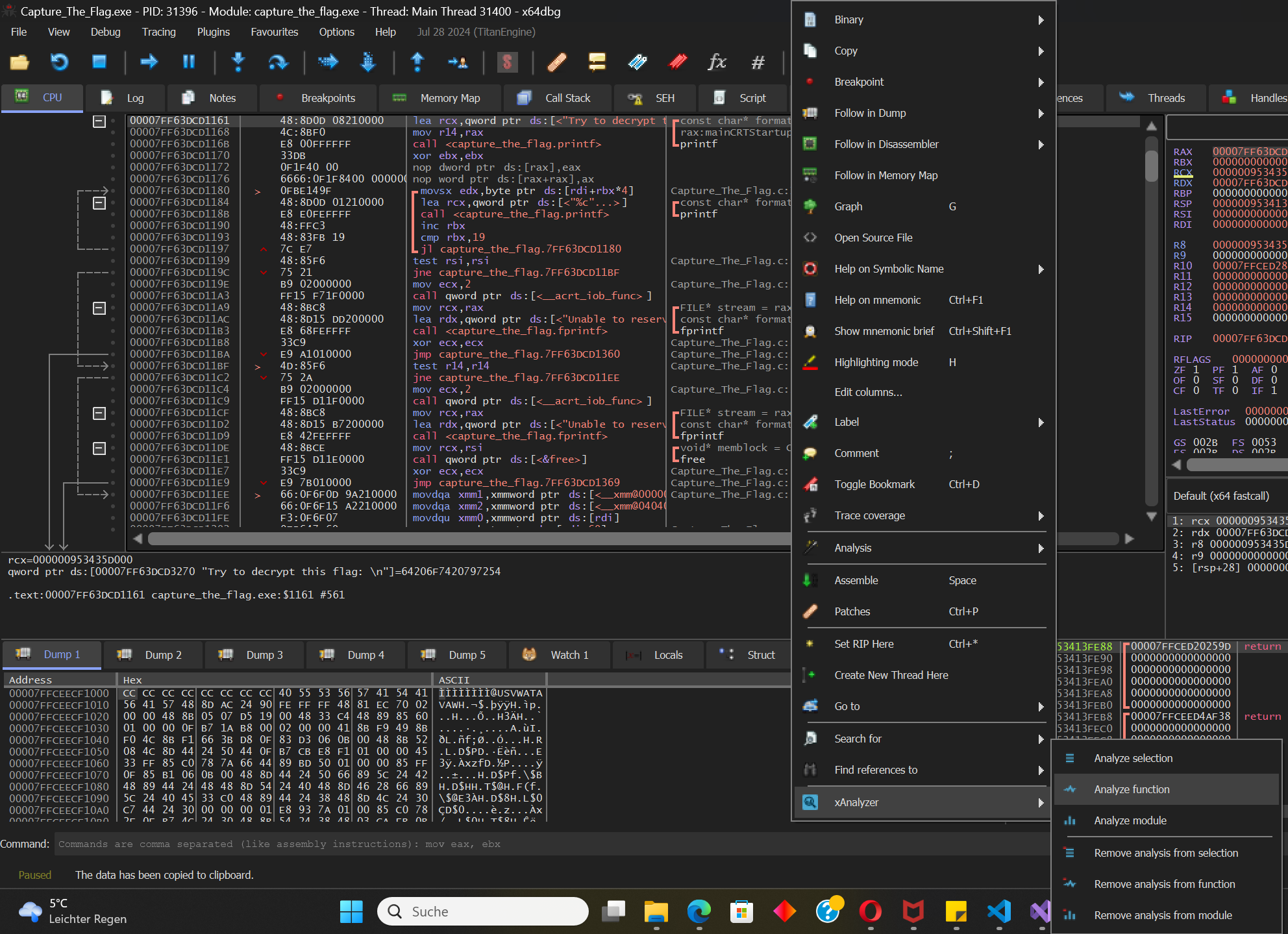


Search for the strings in the module:





Go to address: 00007FF63DCD1161

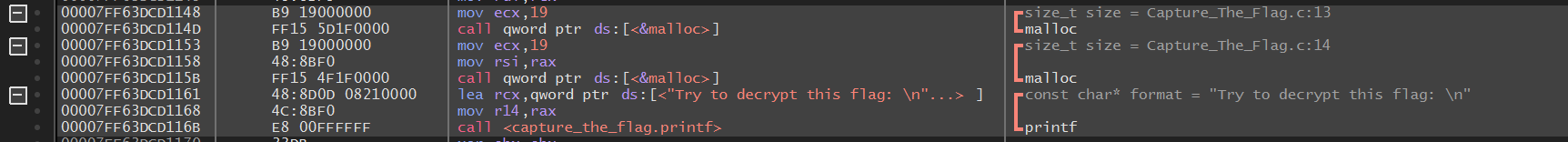


*I recommend to install the xAnalyzer plugin and then use this function: „Analyse function“*

*This function will display API calls in the comments column.*

*( Link:* [*https://github.com/ThunderCls/xAnalyzer*](https://github.com/ThunderCls/xAnalyzer) *)*

At the top of the function we can see two calls to malloc (This is a function from the standard C library and is used to allocate memory on the heap.)



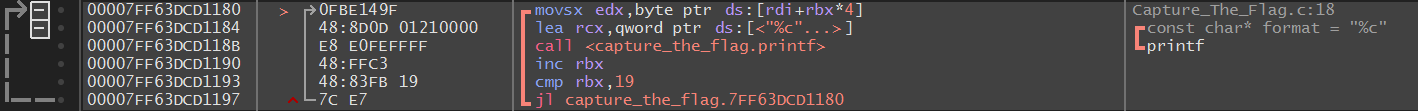
00007FF63DCD1148 | mov ecx, 19

All numbers in x64dbg are hex numbers. 19 hex = 25 decimal

So we reserve 25 bytes. The address of the new allocated memory can be found in rax. After that the value from rax goes to rsi. So rsi holds the address of the new allocated memory.

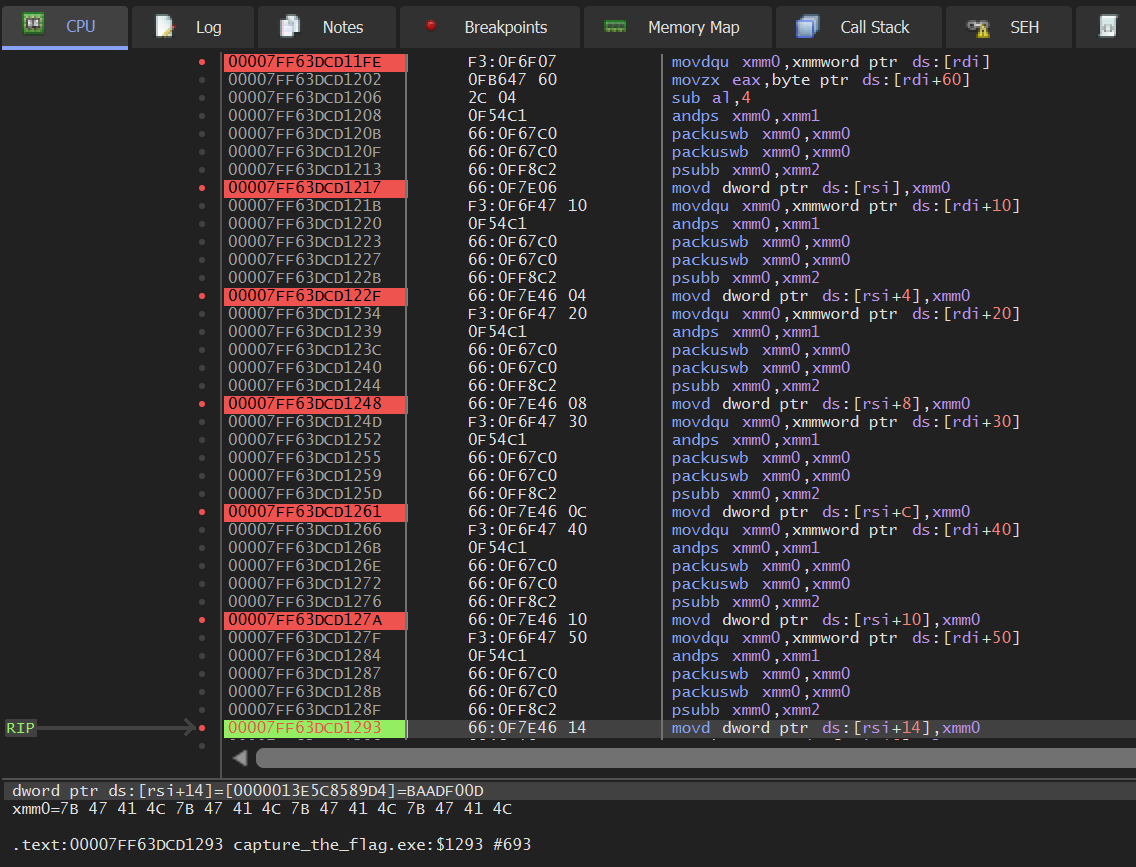
At address 00007FF63DCD1153 we can see again mov ecx, 19 and then we see the command that saves the allocated memory from the first malloc call in rsi. (mov rsi, rax). After that there is another malloc call and the new reserved memory address is again in rax. Then we can see that the new allocated memory address gets saved in r14 (mov r14, rax).

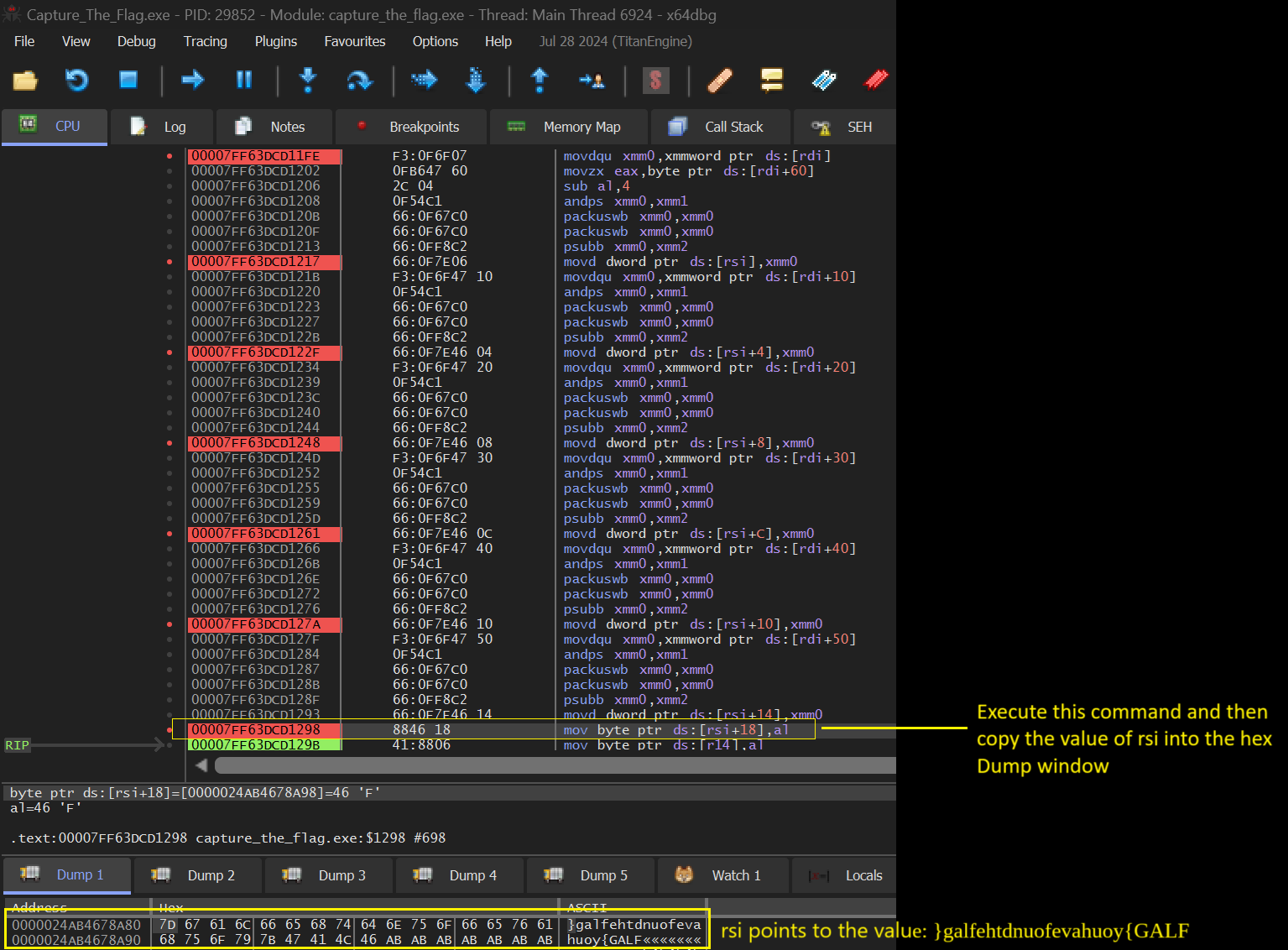
Now there is a call to printf. This will print the following string: Try to decrypt this flag



Here is another call to printf. But in this case printf is gets executed inside of a loop. This will print the encrypted flag.

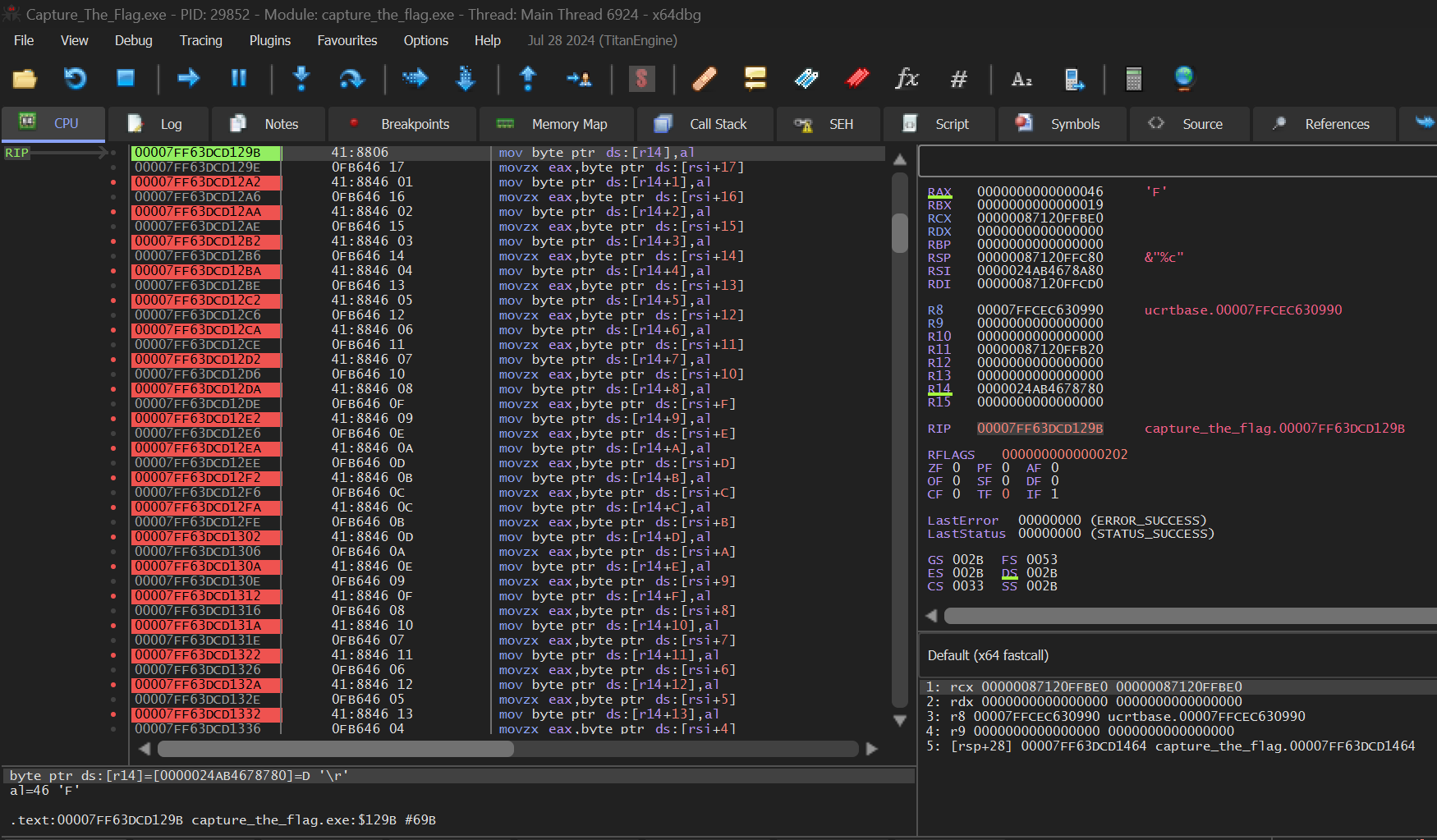
We know already that there are two calls to malloc so there is a good chance that we can find the flag in one of the allocated memory areas. When there is a call to malloc then there is a also a good chance that we can find a call to free. (Free is also a function from the C library it is used to free allocated memory.) The easiest solution now is to look for the free function set a breakpoint and then execute the program. Now we can check if we can find the flag in one of the allocated memory areas. But at first lets keep investigating the program:

The encrypted flag is stored in an integer array. The encrypted flag gets decrypted and saved in a char array.



Set a breakpoint on 00007FF63DCD1298 then execute the command. Copy the value from rsi into the hex dump window. rsi holds an address that points to this value in memory:

**}galfehtdnuofevahuoy{GALF**



We know that have decrypted an integer array and that we have saved the result in a char array. The result can be found at [rsi] and the result was: **}galfehtdnuofevahuoy{GALF**

00007FF63DCD129B | mov byte ptr ds:[r14],al

r14 holds the address of our allocated memory. The value from al goes to [r14].

00007FF63DCD129E | movzx eax,byte ptr ds:[rsi+17]

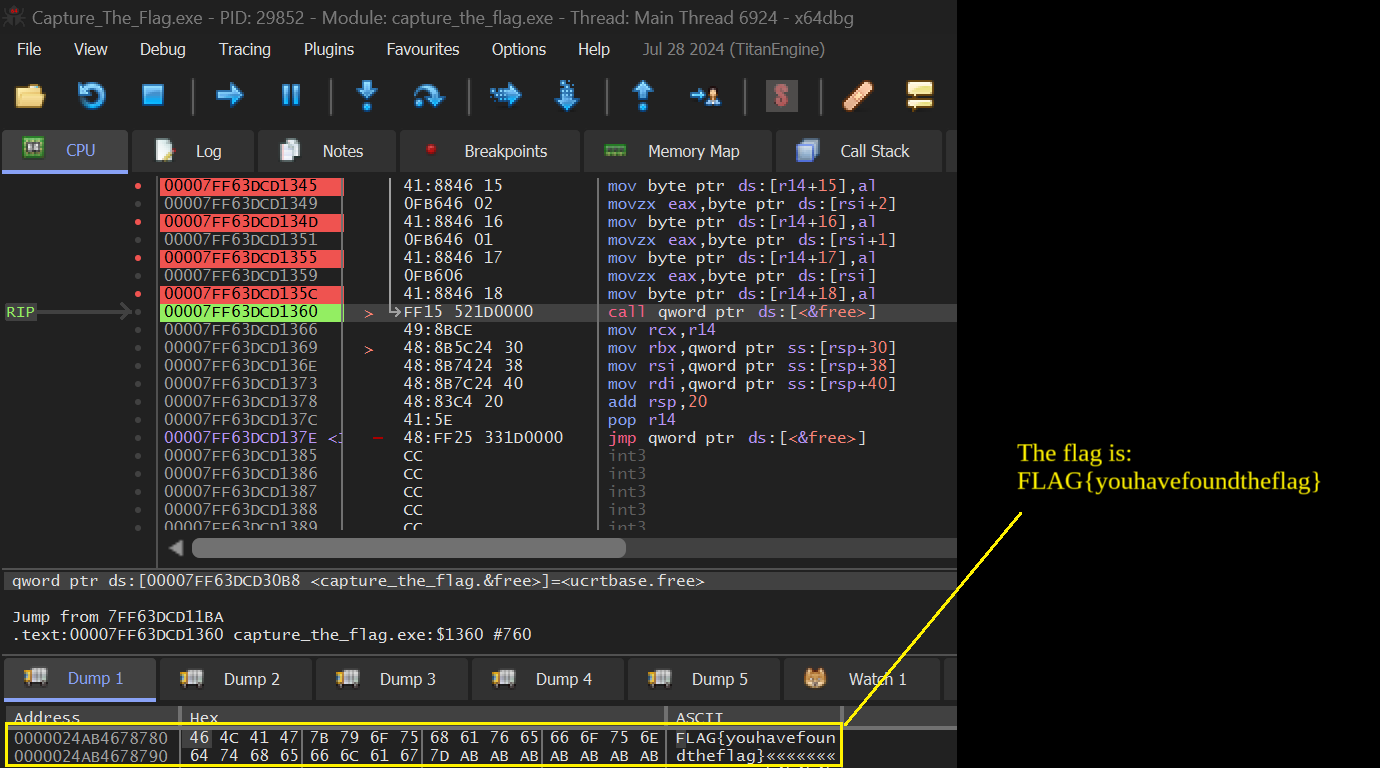
The value from [rsi+17] goes to al.

00007FF63DCD12A2 | mov byte ptr ds:[r14+1],al

The value from al goes to [r14+1]

**In short:**

The decrypted string in [rsi] is stored backwards in memory. So the code will now just copy the data to [r14] but in the correct order.



Set a breakpoint on 00007FF63DCD1360 and let the program run until this address. Now copy the value from r14 into the hex dump. The correct flag is:

**FLAG{youhavefoundtheflag}**