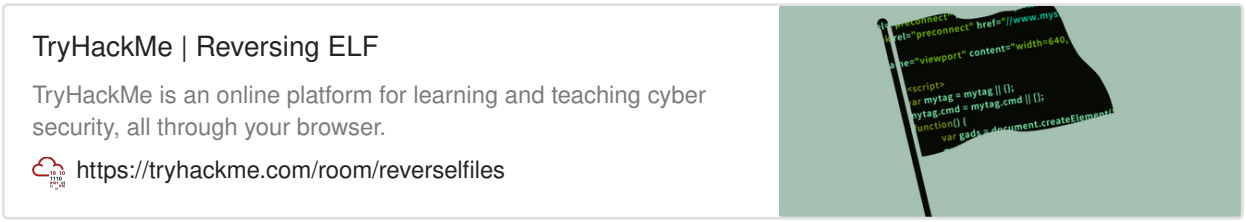
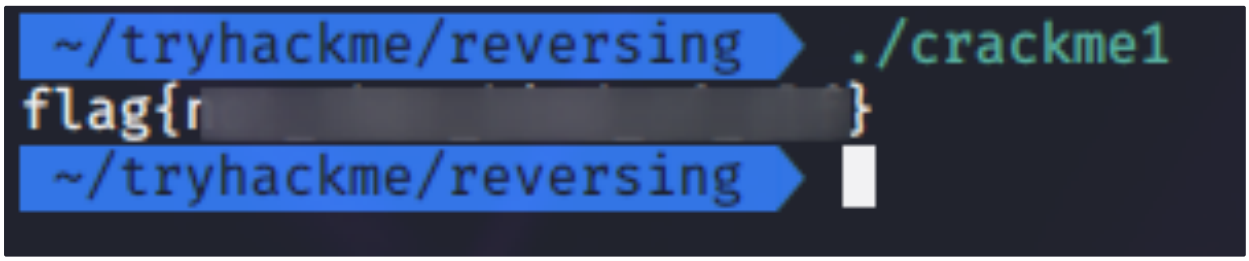
(Write-up) THM: Reversing ELF with Radare2

Note: Sorry if there are any misspelled words in the write-up (I am not a native English speaker).



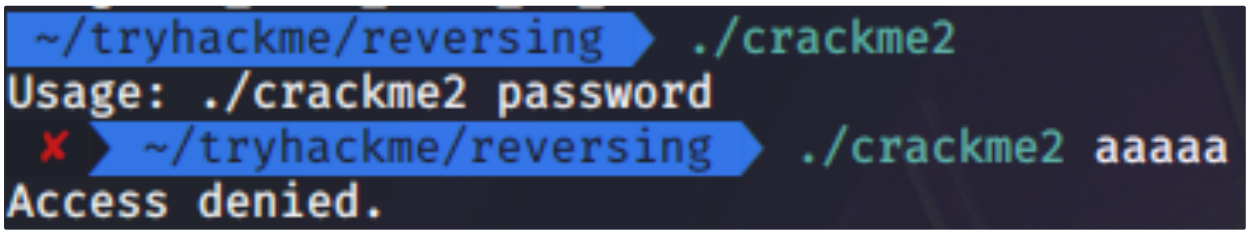
Crackme1

To get the first flag we just need to run the binary. But first we need to give it permissions to run with `chmod +x crackme1` .

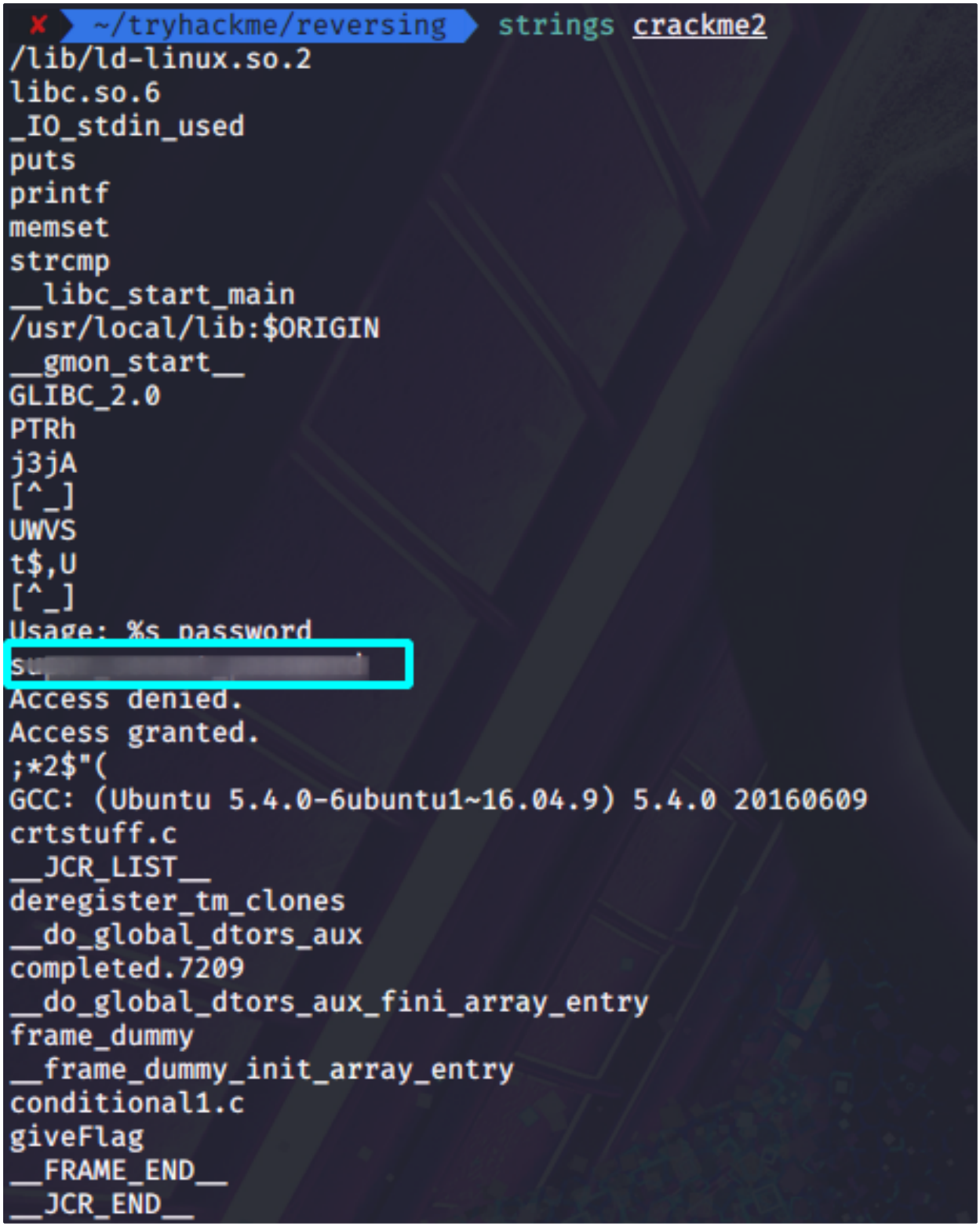


Crackme2

The second crackme asks for a password as an argument. If the password is wrong it outputs "Access denied".



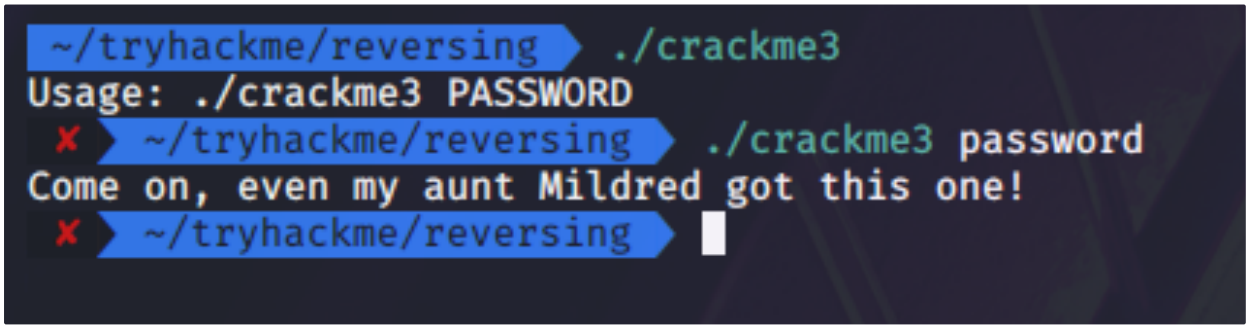
Running the strings command reveals us the super secret password.



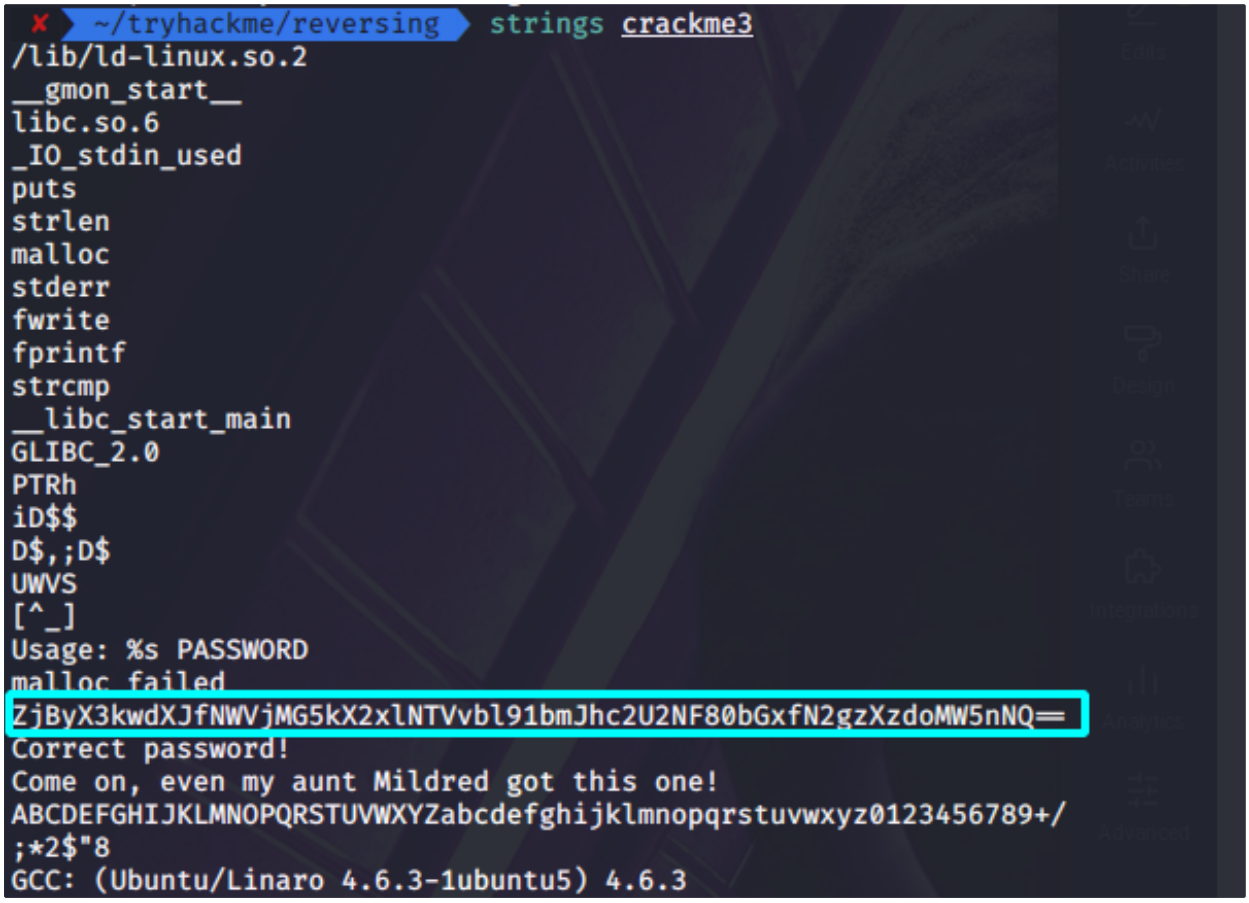
Now we can run the binary with the super secret password as argument to get the flag.

Crackme3

This crackme, as the previous one, asks for the right password as an argument.



Running the strings command on the binary reveals a base64 encoded string.



Decoding the string gives us the flag.

Crackme4

This crackme is similar to the previous ones, where you need to provide a password as an argument, but this time, when we run it, it outputs a message saying that the string is hidden and that it uses strcmp.

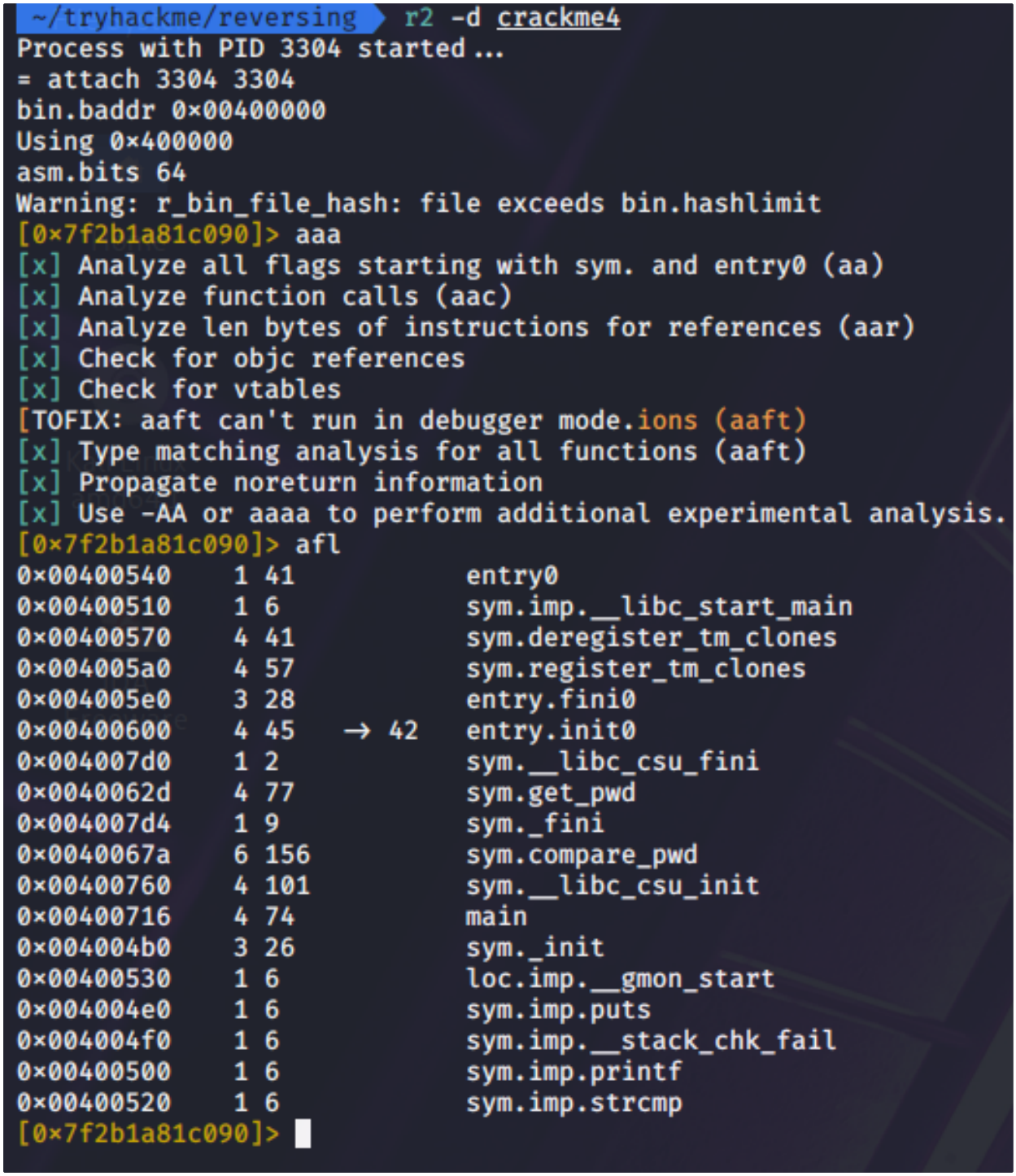
```
~/tryhackme/reversing ./crackme4
Usage : ./crackme4 password
This time the string is hidden and we used strcmp
~/tryhackme/reversing ./crackme4 password
password "password" not OK
~/tryhackme/reversing
```

If we see the man page of strcmp, with the command `man strcmp`, we can see in its description that it compares two strings and returns 0 if they are equal.

```
DESCRIPTION
       The  strcmp()  function  compares  the two strings s1 and s2.  The locale is not taken into account (for a locale-
       aware comparison, see strcoll(3)).  The comparison is done using unsigned characters.

       strcmp() returns an integer indicating the result of the comparison, as follows:

       • 0, if the s1 and s2 are equal;

       • a negative value if s1 is less than s2;

       • a positive value if s1 is greater than s2;

       The strncmp() function is similar, except it compares only the first (at most) n bytes of s1 and s2.
```

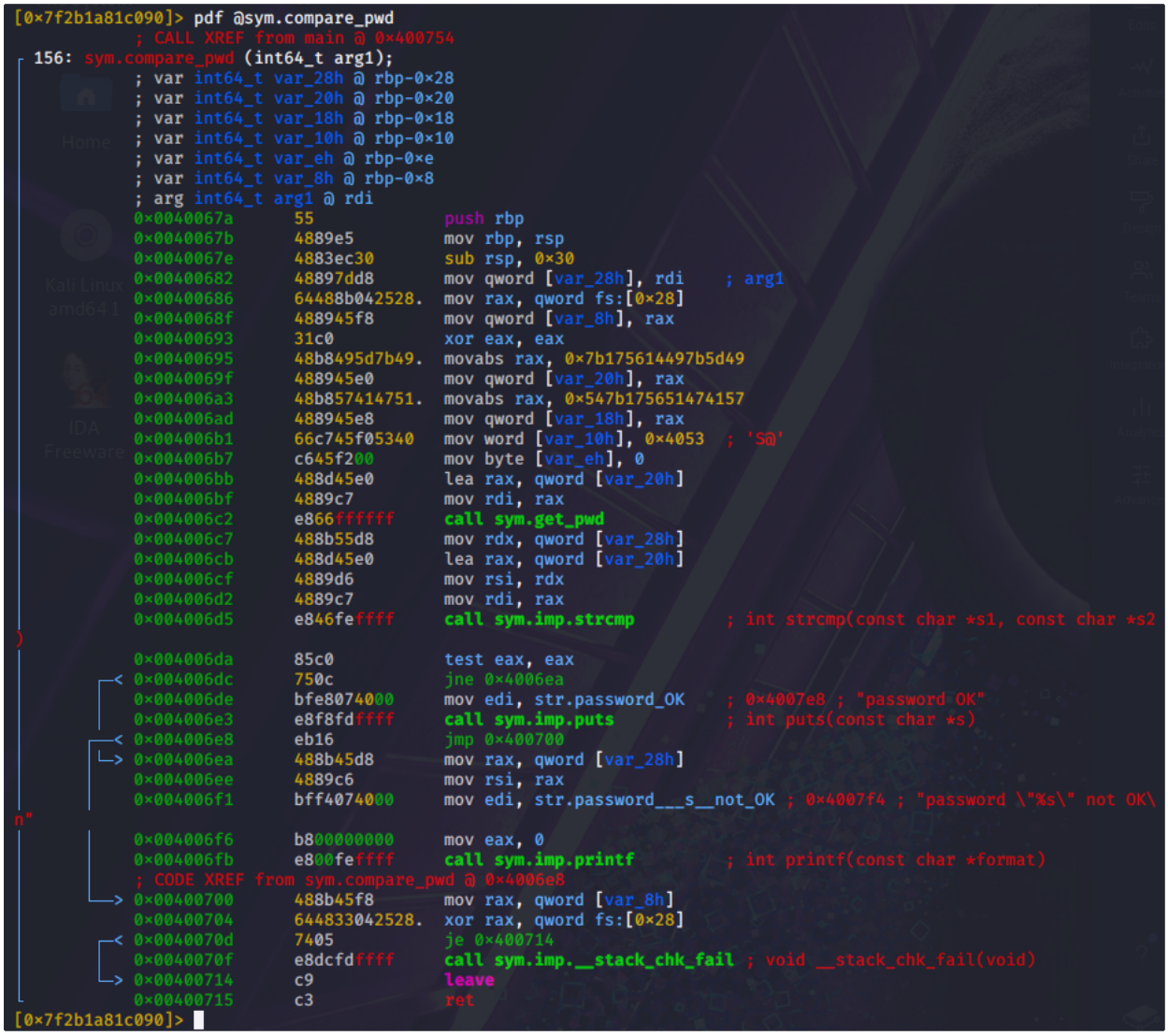
To solve this and the rest of the crackmes I will use `radare2`.

We can start the binary in debug mode with radare using the command `r2 -d crackme4`. Then we analyze the binary with the command `aaa` and check for its functions with `afl`.

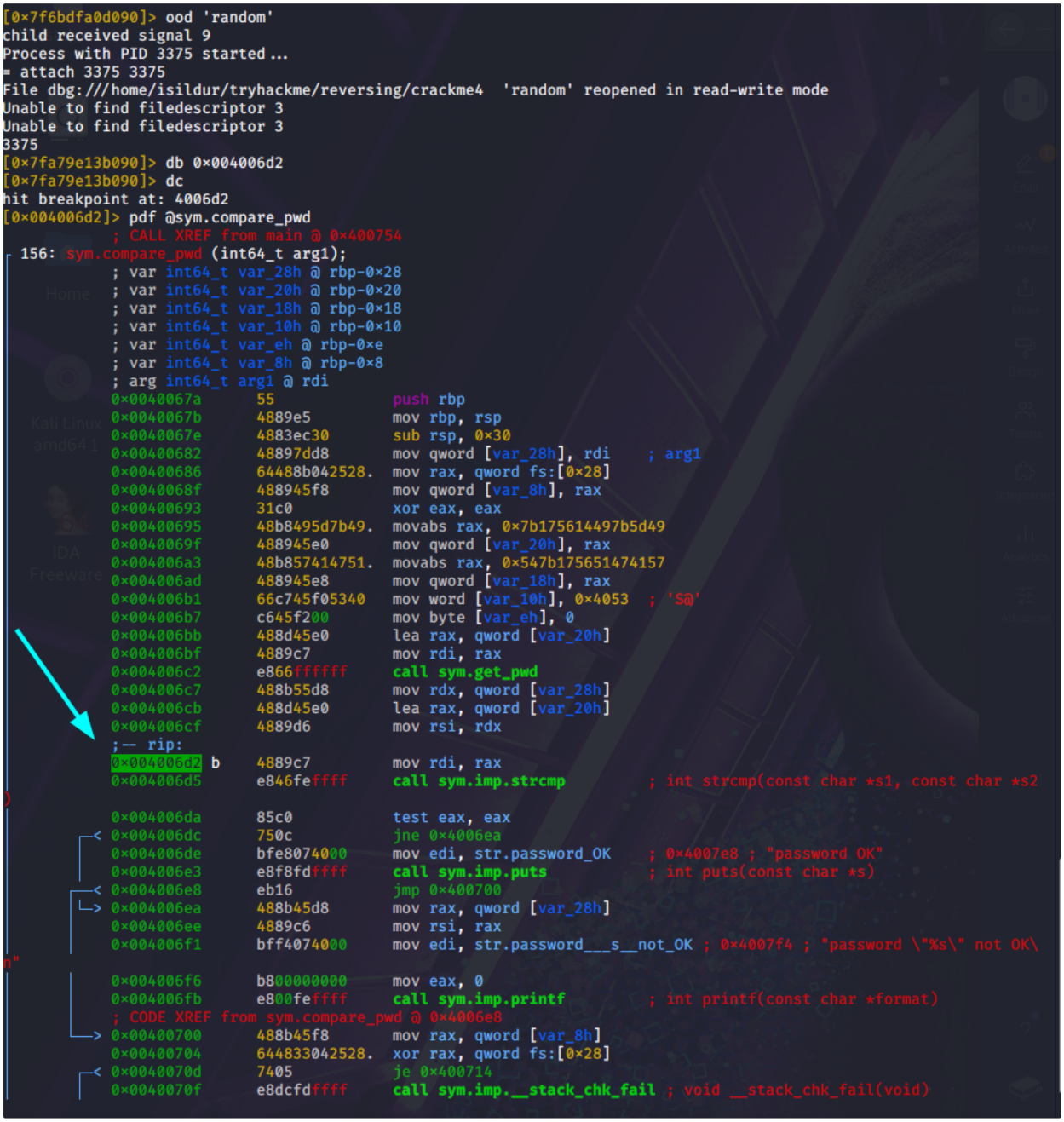


We see that the binary runs a lot of function, but the most important here and in most binaries is the function `main`. We can analyze the main function running `pdf @main`.

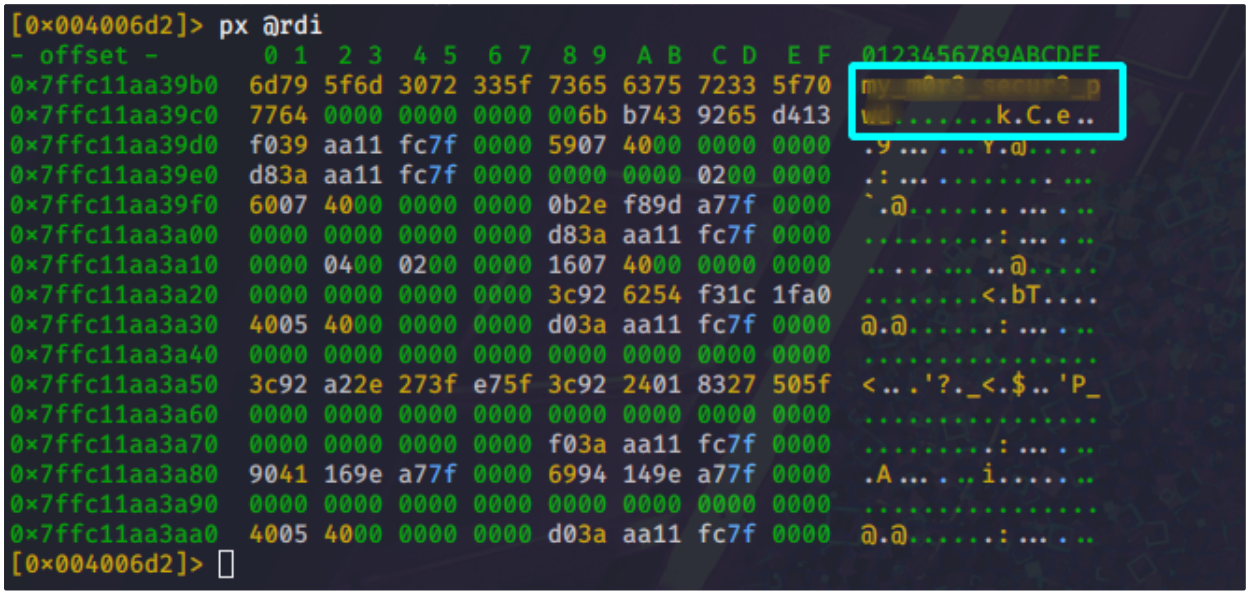
Analyzing the main function shows us a lot of assembly code that we can ignore for now, what is important for us is that the main function calls another function, `sym.compare_pwd` , before it returns. We can analyze this function with the command `pdf @sym.compare_pwd` .



Once again, we can ignore much of the assembly code and focus only on the important instructions of this function. As stated before, this binary runs strcmp to compare two strings. Here we can see the strcmp at the address `0x004006d5`. It will compare the string in the register `rdi` with the string in the register `rax`. If both strings are equal, it will set the 32bit `eax` to 0 and return the "password ok" message. The register `rax` holds the input we used as an argument, so what we need to know is the string that `rdi` holds before the function executes the strcmp. To do that, we set a breakpoint at the address of the instruction `mov rdi, rax` with the command `db 0x004006d2`. Then, we execute the program with `dc` until it reaches the breakpoint and stop. Since we need to provide an argument with the program, we cad run `ood 'argument'`, and then set the breakpoint and run it (the string we use doesn't matter).



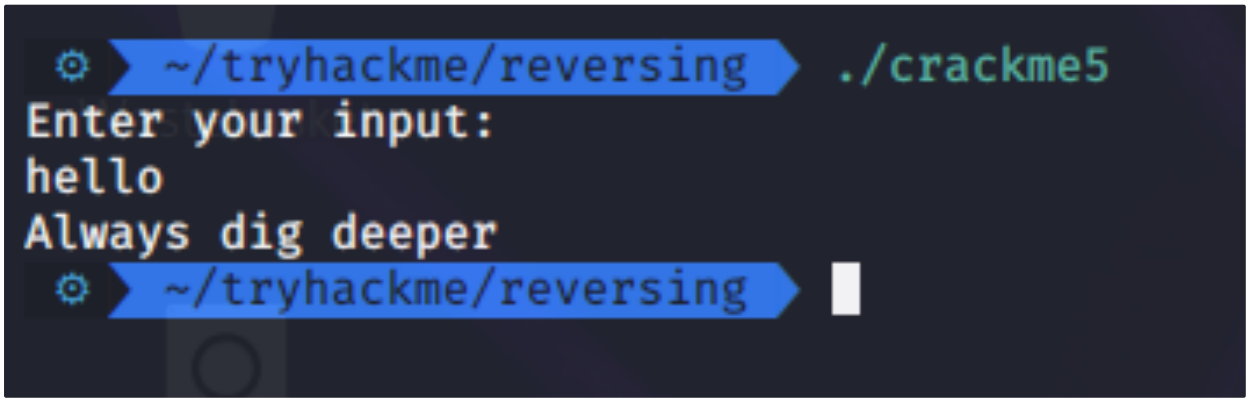
Now we've reached the breakpoint. To know the value of `rdi` we type `px @rdi` .



We revealed the content of `rdi` before the strcmp, and we can check that it is indeed the correct password by running the binary with it as argument.

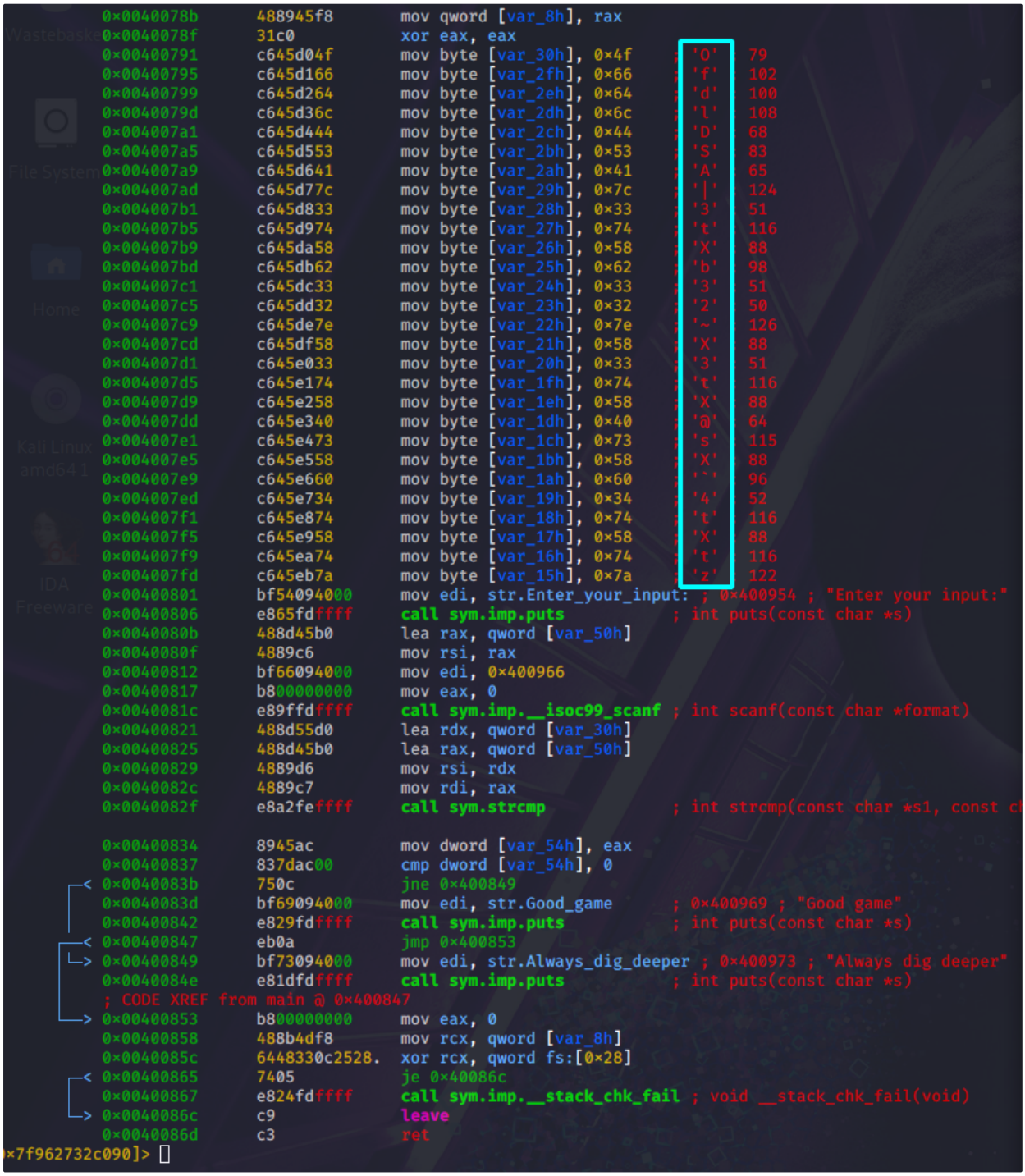
Crackme5

This binary asks for user input instead of of arguments.

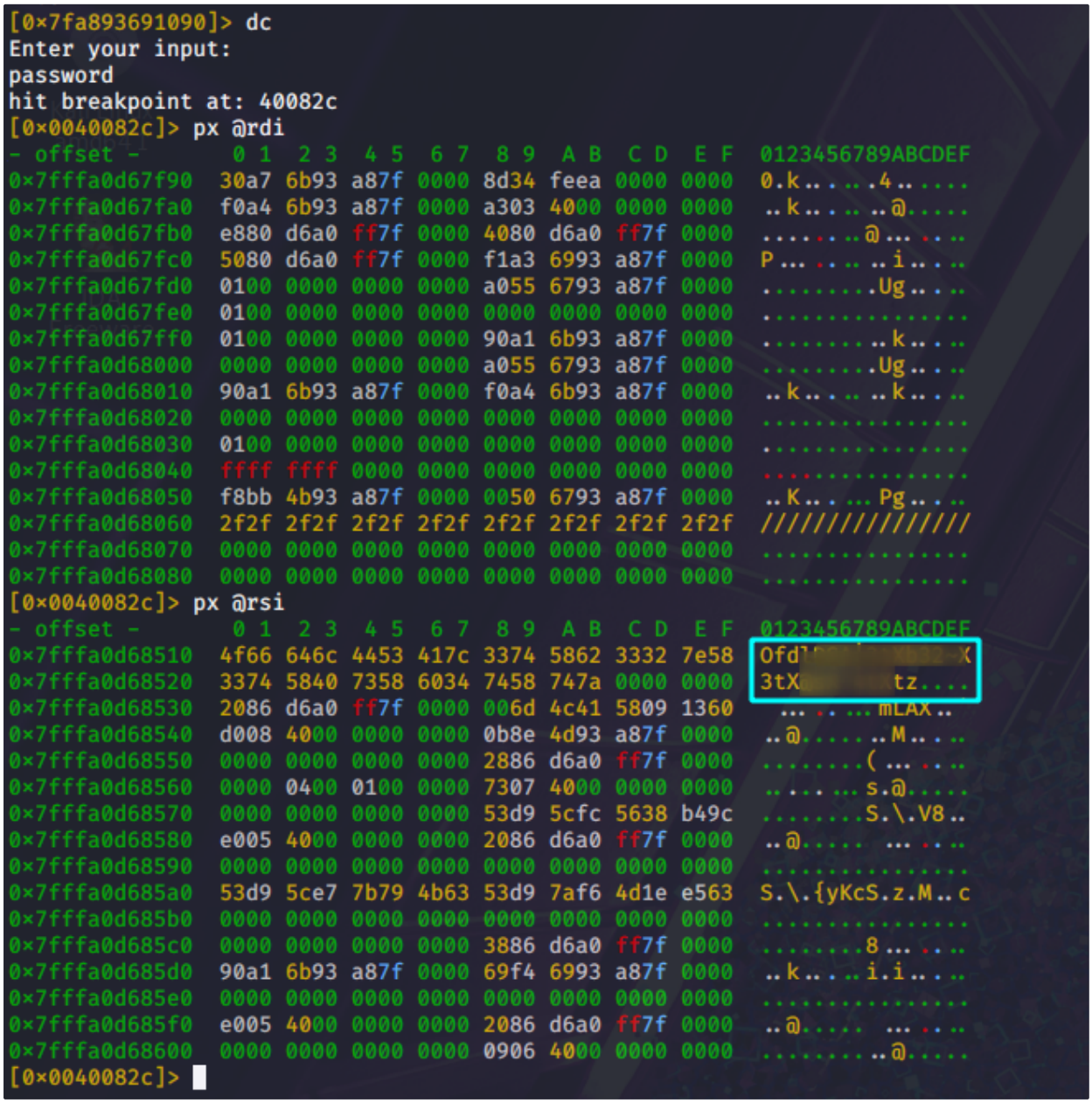


Once again lets debug it with radare2 and disassemble the main function.

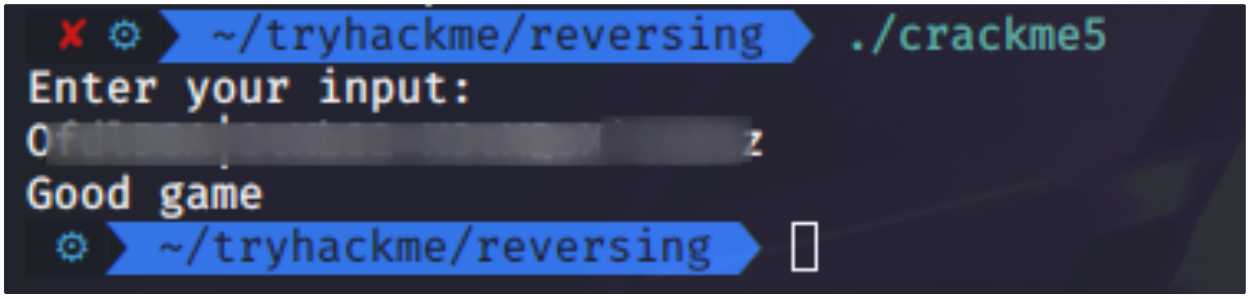
At the beginning of the function we see a different character assigned to each variable. Although quite troublesome, if we just assembled all this characters in a single string we would find out that the answer was always exposed to us.



We have once again a strcmp, this time in the main function. As before, we set a breakpoint before the strcmp, which compares the registers `rdi` and `rax`. This time, `rdi` doesn't store anything of value for us, but if we investigate the `rsi` register, we see that its content is a string with the same characters assigned to each variable at the beginning of the main function.

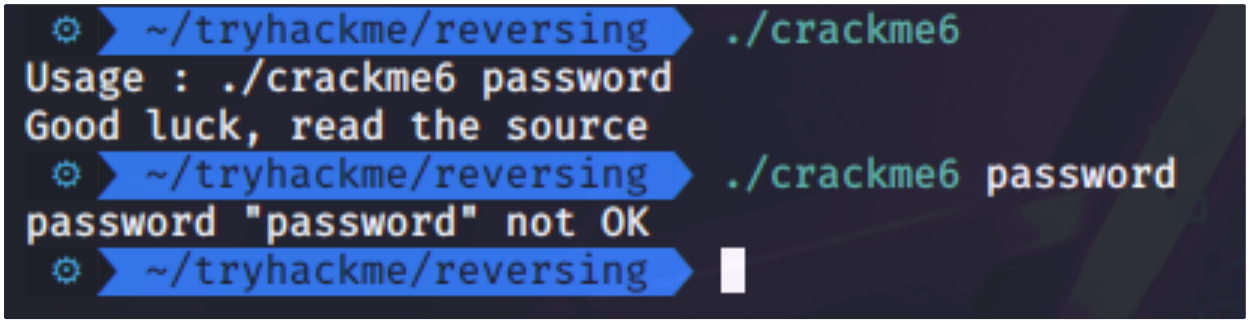


If we use this string as the input we get the "Good game" message.

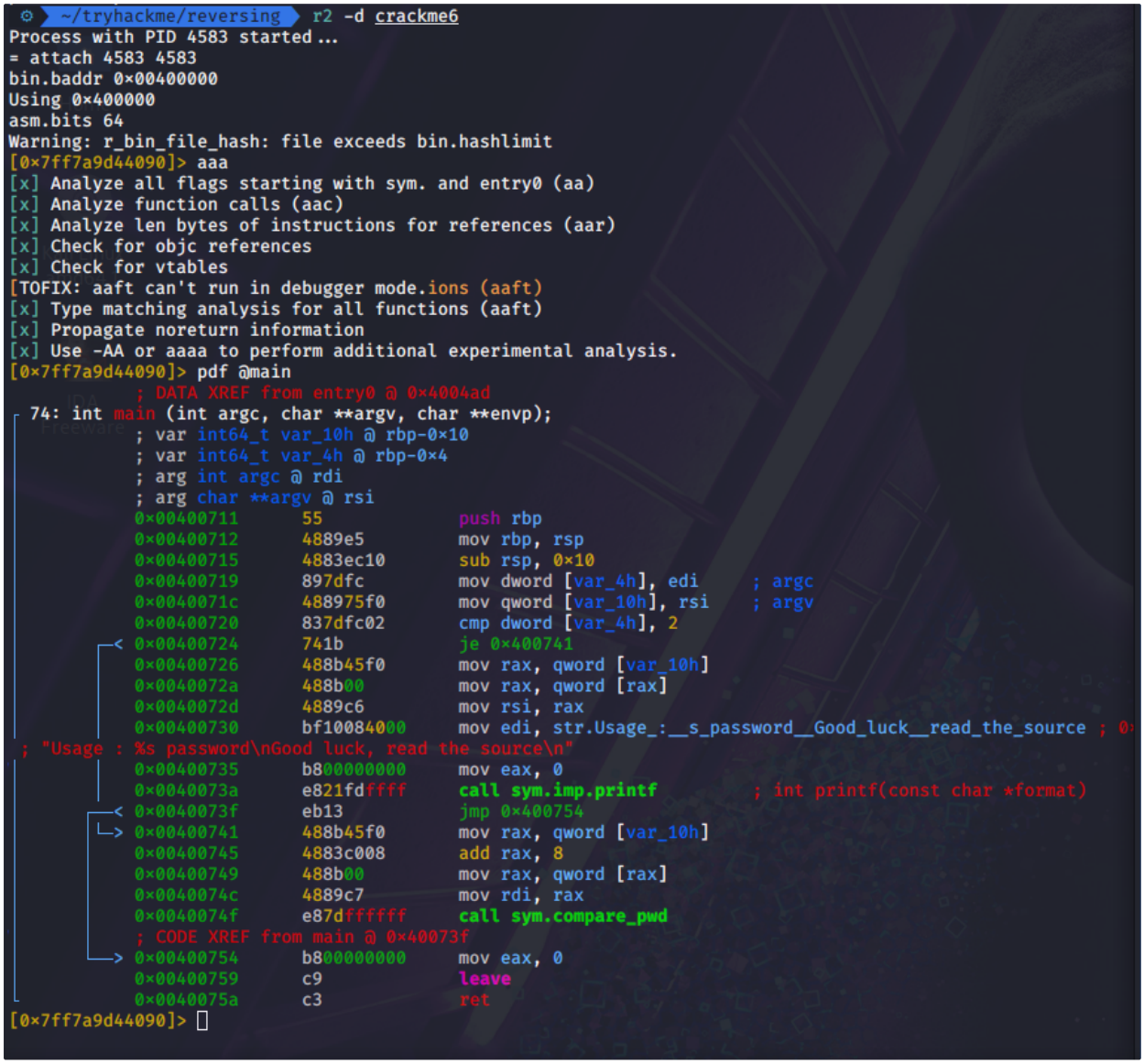


Crackme6

Crackme6, like other previous crackmes, asks for a password as argument.

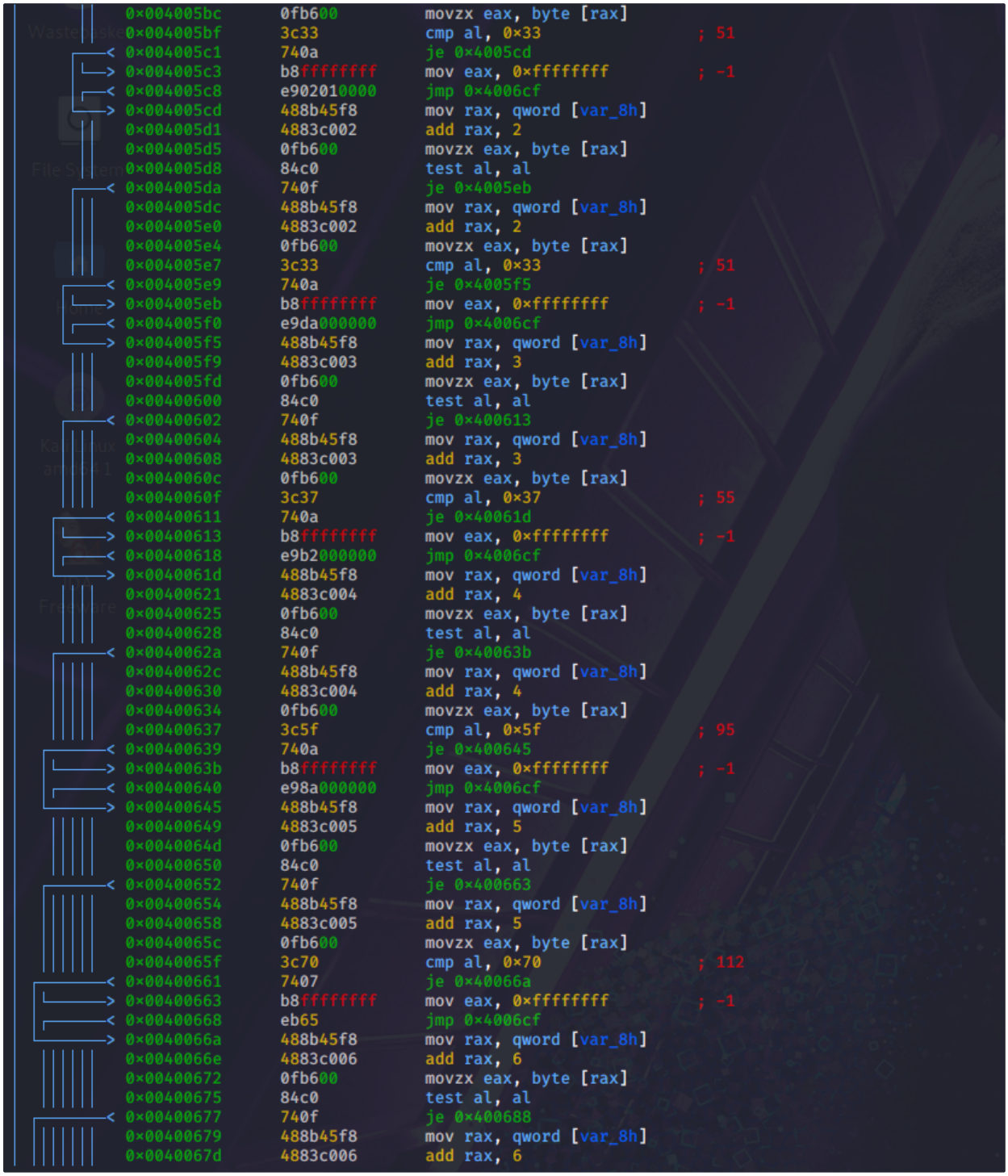


Let's analyze the main function.

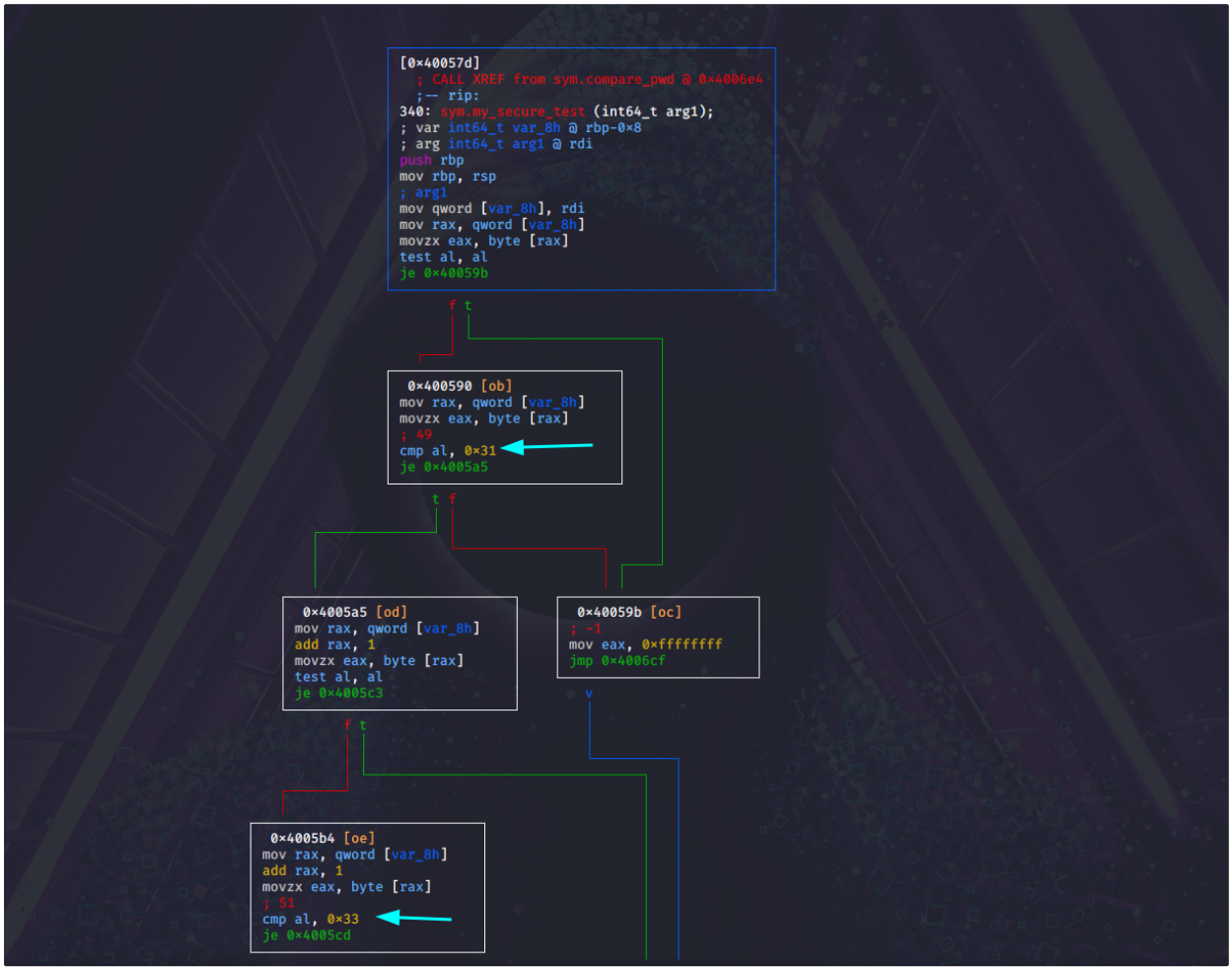


Once again let's focus on the essential instructions of the function. What caught my attention here is the `sym.compare_pwd`, so let's analyze it.

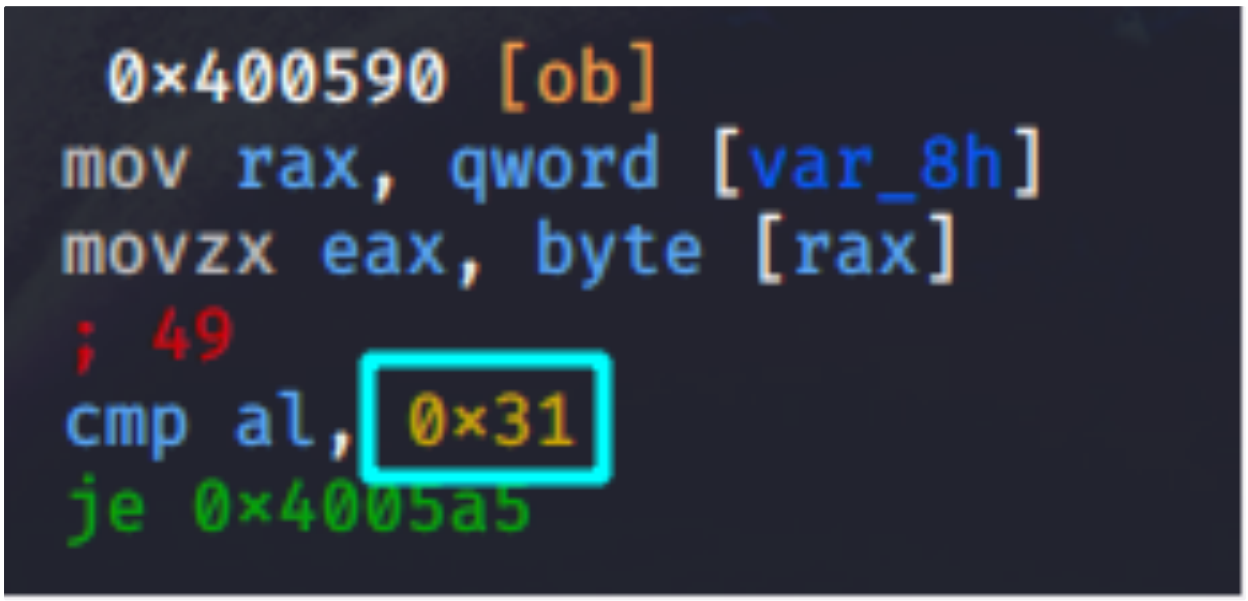
We can see that it is in this function that the password is checked with our input, but we cannot know what our input is actually being compared to because that value comes from the `sym.my_secure_test` function. So let's take a look at it.

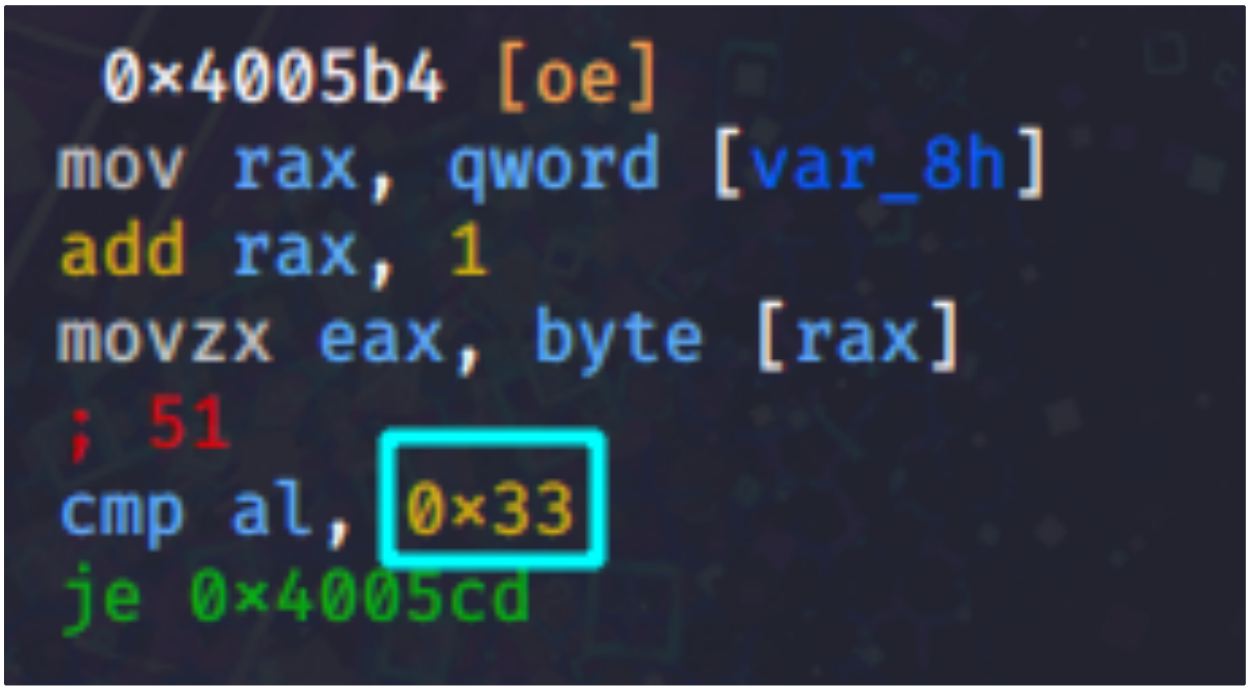


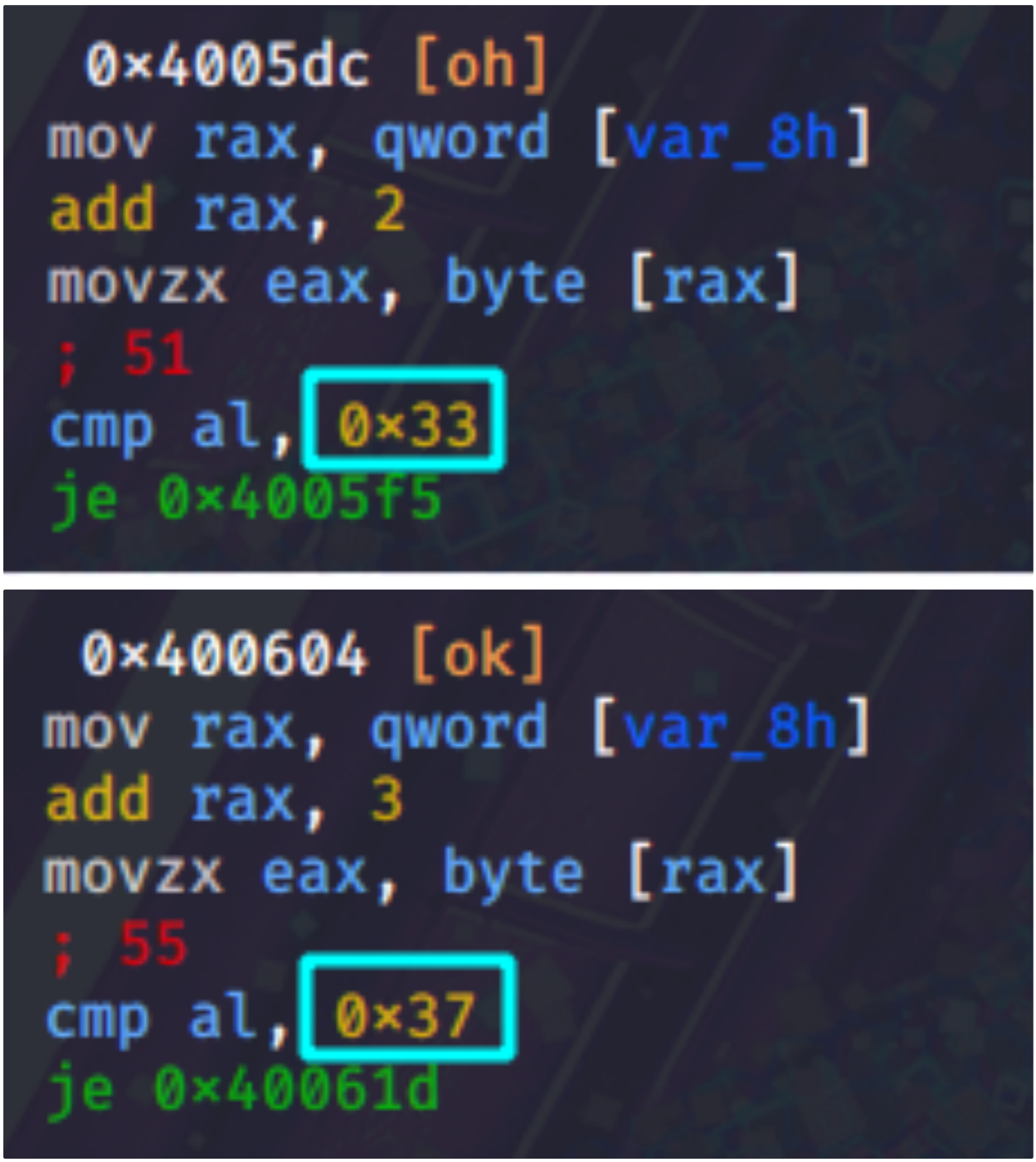
Ok! This function is too big for a single screenshot, and its really hard to analyze with all those jumps, specially in text mode. This where graph mode comes in handy. To enter graph mode just type `vV` . Once in graph mode we can use the same commands as in text mode, but we need to start them with `:` instead.

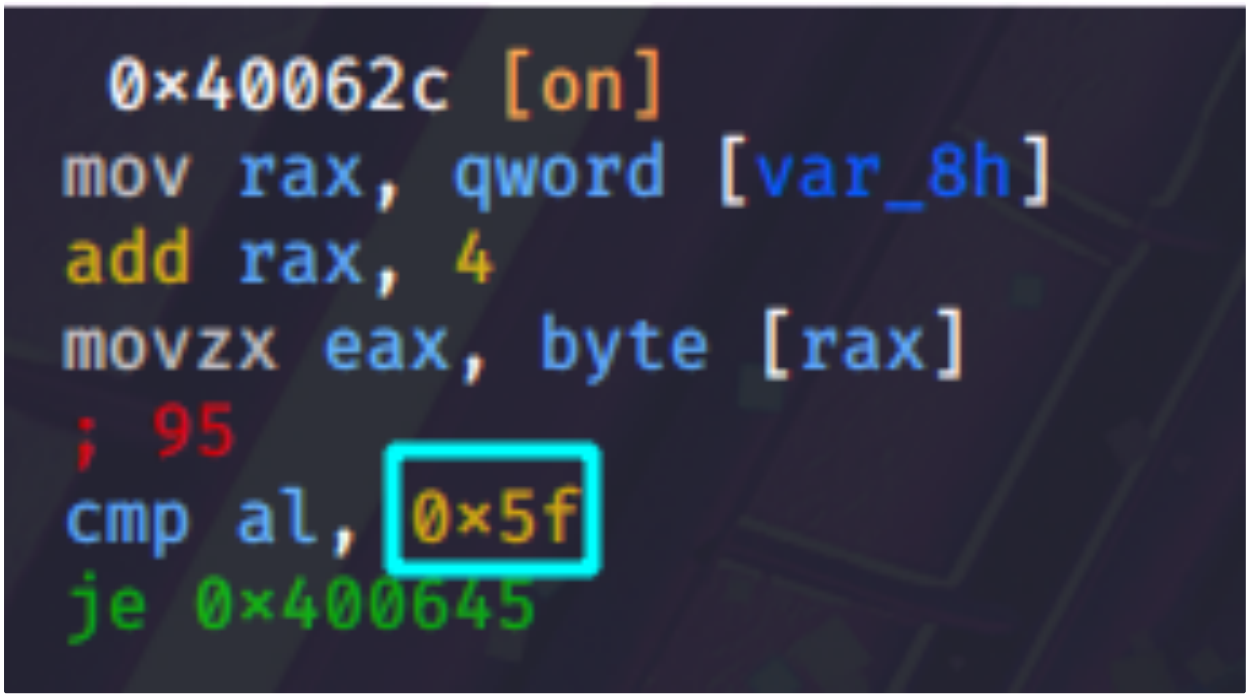


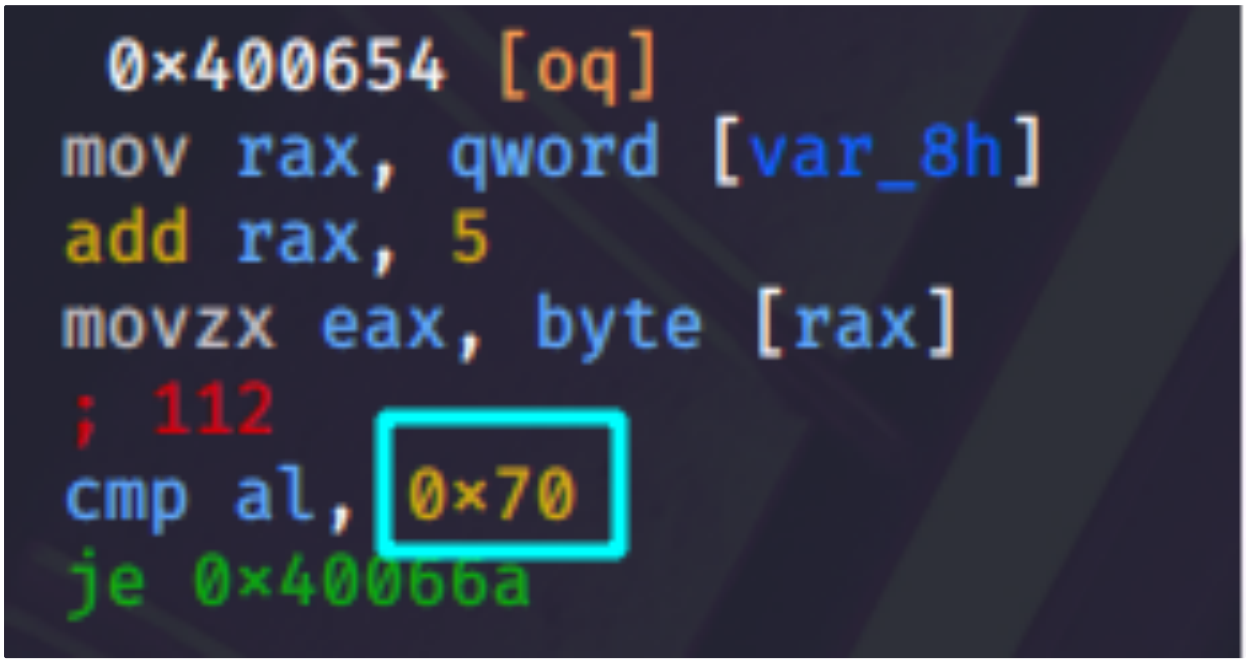
Graph mode makes things more easy to understand. We can see that this function compares the least significant 8 bit of the `rax` register (`al`) with some hex value, then if its equal it increments the `rax` register and compares it again with another hex value. It seems that it is comparing every char in our input with these hex values.

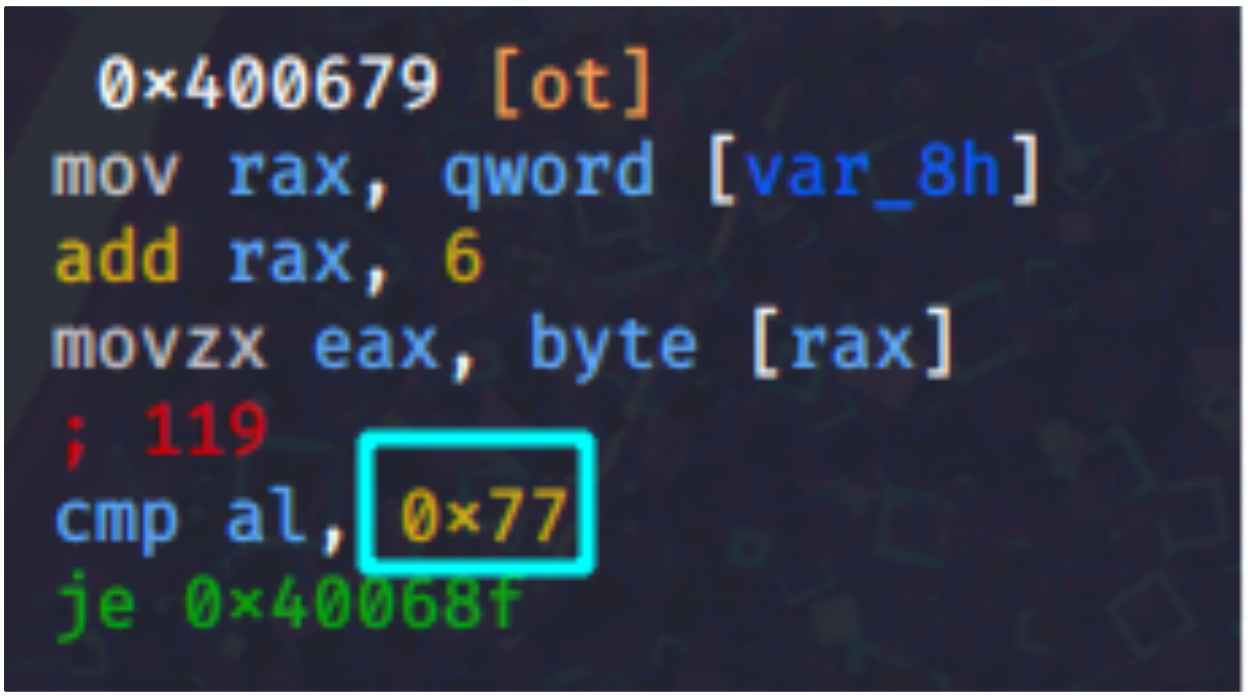

0x400590 [ob]
mov rax, qword [var_8h]
movzx eax, byte [rax]
; 49
cmp al, 0x31
je 0x4005a5

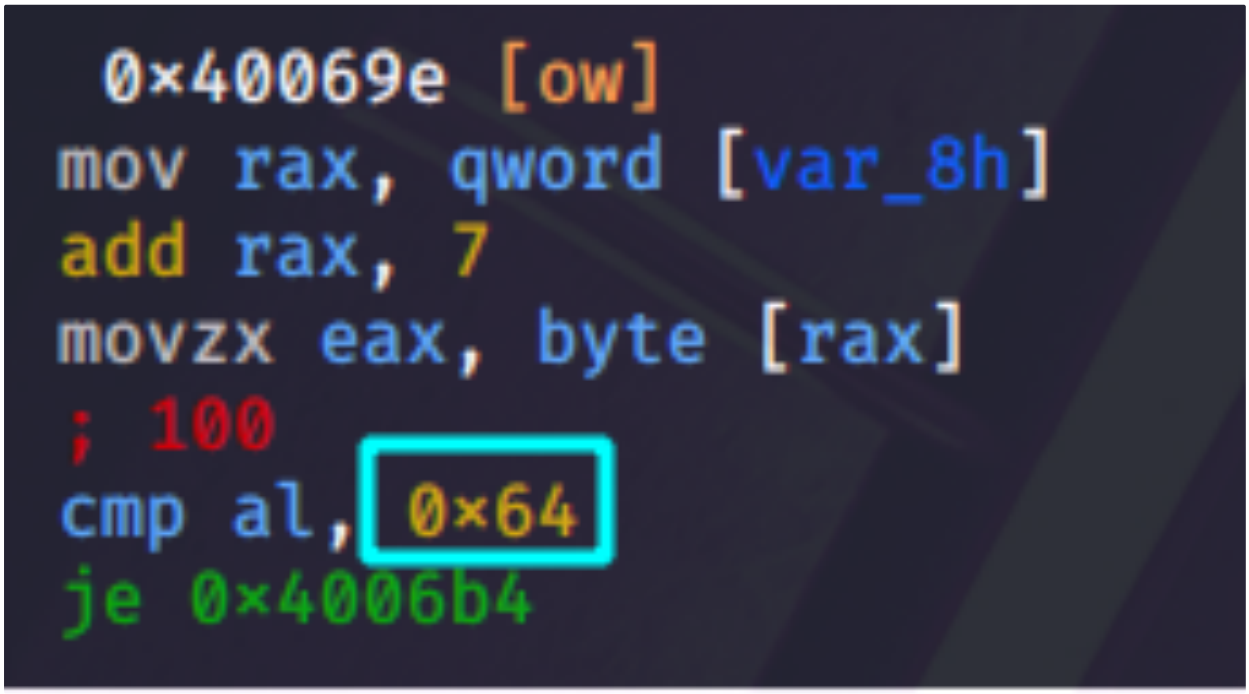

0x4005b4 [oe]
mov rax, qword [var_8h]
add rax, 1
movzx eax, byte [rax]
; 51
cmp al, 0x33
je 0x4005cd


0×4005dc [oh]
mov rax, qword [var_8h]
add rax, 2
movzx eax, byte [rax]
; 51
cmp al, 0×33
je 0×4005f5
0×400604 [ok]
mov rax, qword [var_8h]
add rax, 3
movzx eax, byte [rax]
; 55
cmp al, 0×37
je 0×40061d

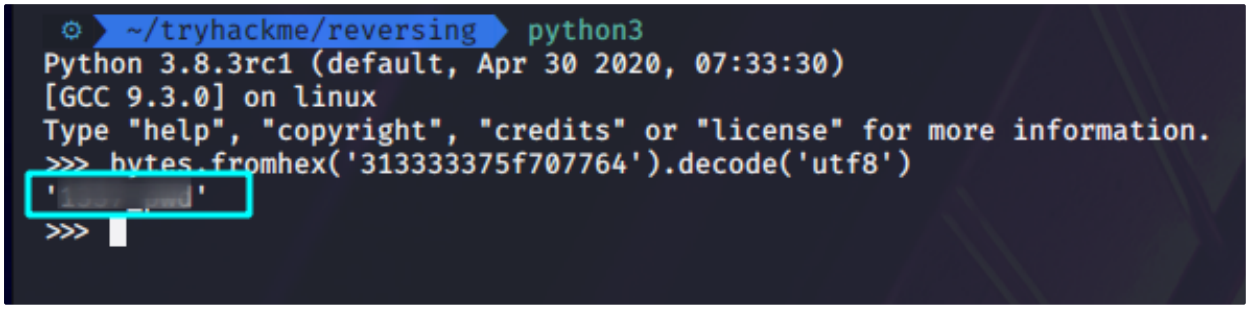

0×40062c [on]
mov rax, qword [var_8h]
add rax, 4
movzx eax, byte [rax]
; 95
cmp al, 0×5f
je 0×400645


0×400654 [oq]
mov rax, qword [var_8h]
add rax, 5
movzx eax, byte [rax]
; 112
cmp al, 0×70
je 0×40066a

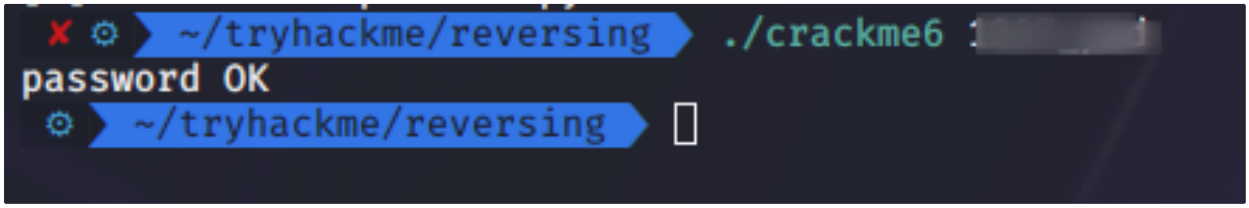




If we join all these values together and decode them into their ASCII equivalent we get the password to solve the crackme.



We can confirm it is the right password by running the binary.



Crackme7

When we execute this binary it has a basic interactive console where we can say hello and add numbers.