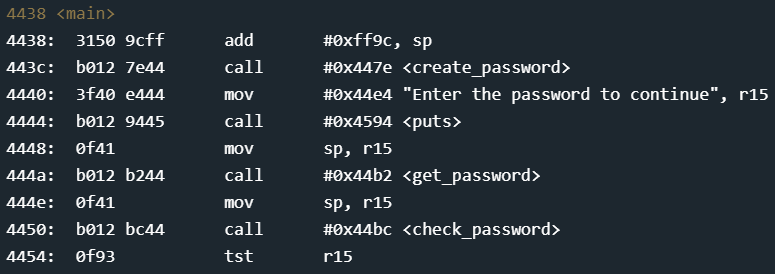
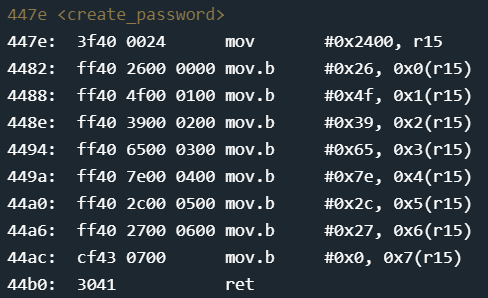
First thing first with this challenge: toggle minimal mode. Otherwise the scrolling is twice as ridiculous. I usually like to start with an objdump, which is conveniently provided. Scrolling through it I notice something interesting once we get to <Main> :



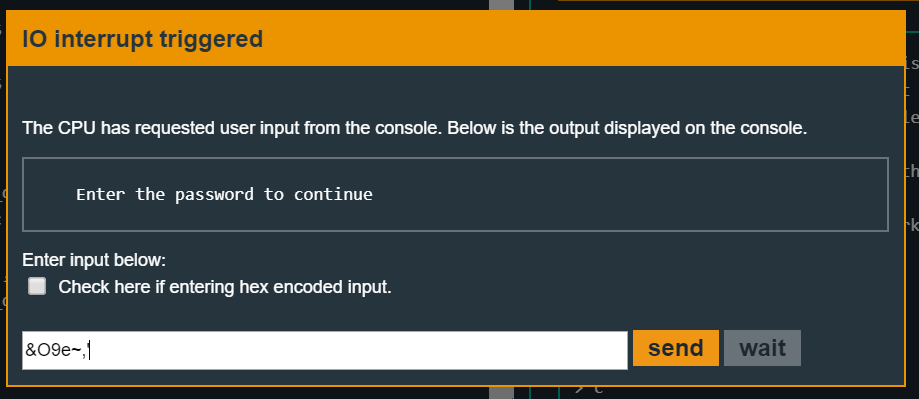
I’m gonna bet that the <create\_password> function looks *pretty* interesting. Let’s find that.

Scrolling a bit further down we find <create\_password> :

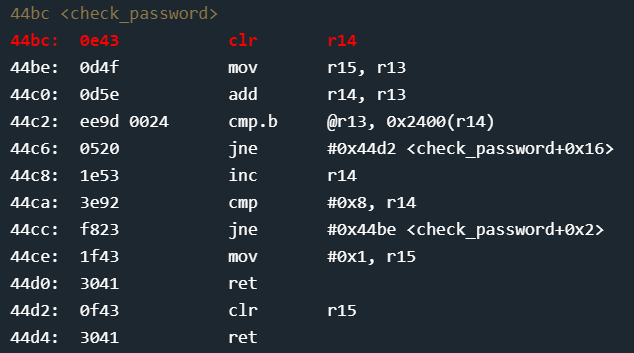


This actually confused me at first, so I decided to set a breakpoint and step through it. When it moves 0x2400 into r15 it is storing the memory address 0x2400. After that it is doing mov.b which I am going to assume means *move byte* because I could not find this in the copy of the instruction manual I had, but that is how it performs. Notice the ‘0x#(r15)’ as the destination. That is telling it to go to the memory location of r15 (0x2400 in this case) and then go to the position (the 0x#) and then move the byte from the left side of the argument into that location. This gives us seven hex values (which could correspond to ASCII characters) and a 0...why does that seem familiar...oh yeah! Possible character values followed by a 0 could be a string, since 0 is the null terminator when reading characters! Since this function is called <create\_password> we can reasonably guess that it is generating a fixed, seven character password. After converting the hex to ASCII we have: [&O9e~,'] without the [].

Continuing through the program it asks us for the password, so let’s try to enter our guess now:

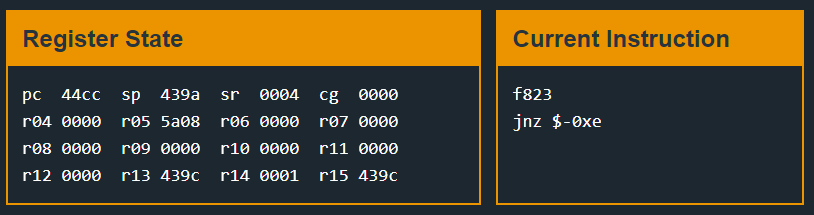


Having already set a breakpoint for when it calls <check\_password>, we’ll send that password through and continue the execution.

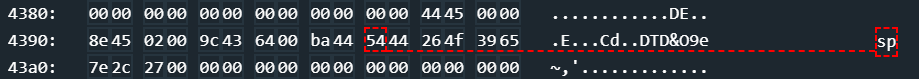


This is interesting! First we clear r14, then we move r15(which is an address pointing to the password we entered) into r13. Then we add r14 to r13? Wtf is that about...let’s keep stepping through and see if we can figure it out. The next is cmp.b … gotta mean compare byte. So we compare the byte where r13 is pointing and then compare the byte at location #0x2400(r14). Check to see if they are not equal, jumping to the return if they are not. Otherwise we increment the value of r14 and then compare it to 0x8. That would make sense if we are moving through a 7 character password, plus the 0 for the null character. When then jump back to the beginning and repeat. When r14 gets added to r13 that must be so we move to the next character! This makes sure we are comparing the correct character locations.

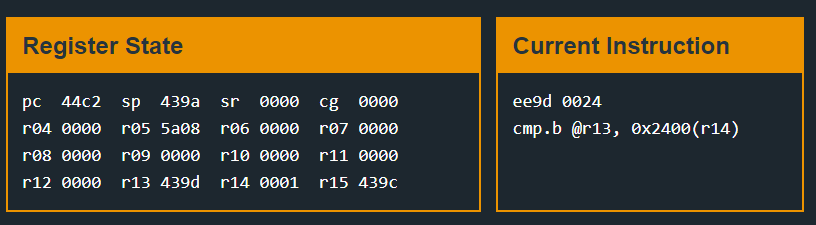
Looking at the registers, let’s check the value of r13 before we move onto the next iteration of <check\_password>:



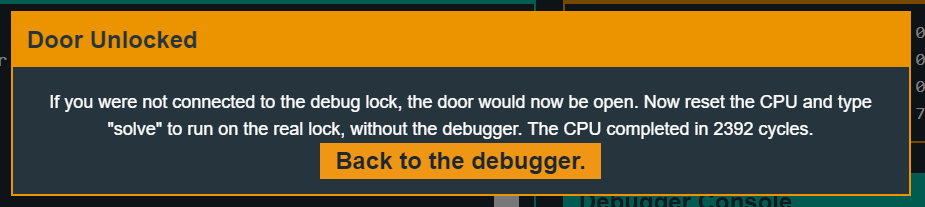
If we look at 0x439c in memory we see the & which is the first character of our password...and following it are the other characters from our password!



Once we start the next iteration, r13 gets reset to the base value of r15 and then r14 gets added to it, in this case 1.



If we keep comparing the characters and they are all correct, we will return back into <main> with the appropriate flags set!



Got it! Now we just have to put it into the real lock and collect they payday! Onto the next challenge!!!