Reverse 1: Hello World

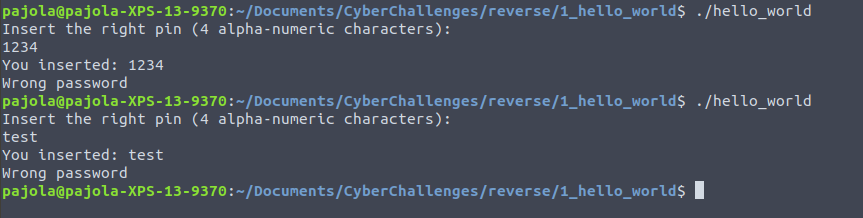
Welcome to the new (and last) chapter of our hacking exercises. In this new section we are going to infer and “exploit” vulnerabilities … in a hard way.

The first exercise “hello world” should kindly bring you in this new world. First of all, you should remember this rule:

*“You* ***do not*** *have the source code and we are going to give you the information of which file you can use”*

This because in a real scenario you often do not have any code on which you can exploit / control. If we give you the source code is only because you might need a re-compilation of the code based on your OS and architecture.

But now we can move on and start the exercise. Since we have only the executable file, we can execute it and do some tests.



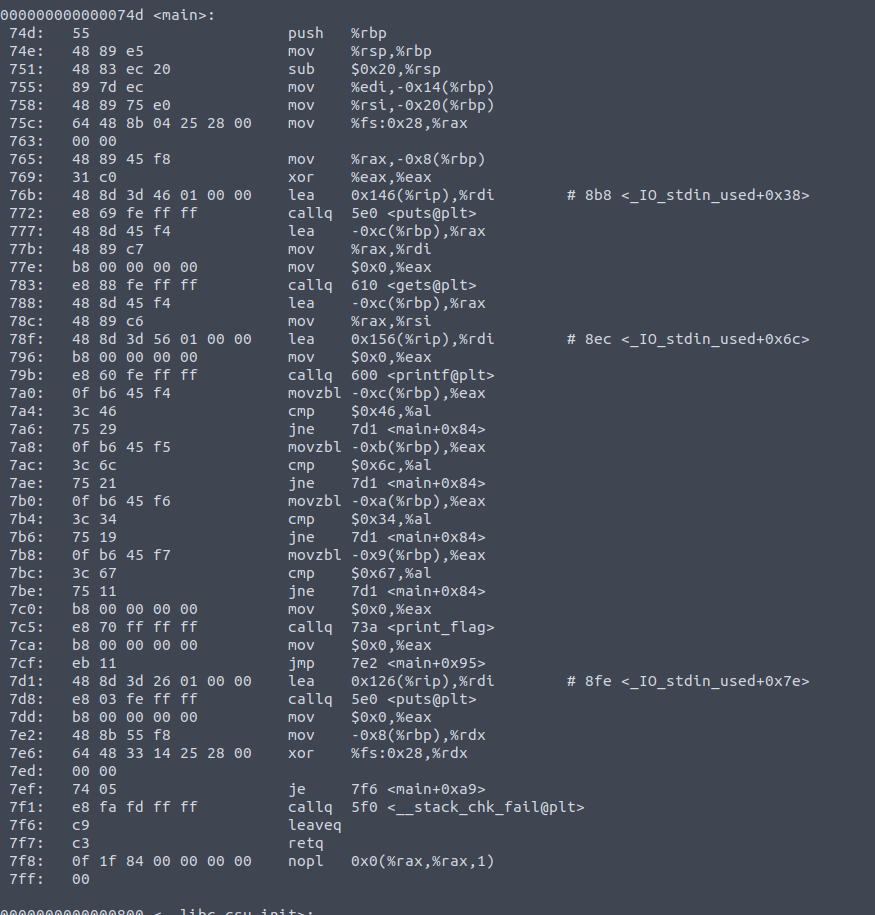
It seems that:

1. The password is alpha-numeric;
2. The password length is 4;
3. We need to guess the correct password / pin.

Based on our lessons on encryption, we could do a brute force attack, since the vocabulary size is around 40 digits, and we need to try only 40^4; however, this is not the goal of reversing, so we need to find another way. From the pwn lesson, we know a really useful command, which is *objdump*. So, we can type on our terminal the following:

*objdump -d hello\_world*

Now, we can focus only on the *main* function:



It is assembly and you might be lost, but not give up. Let’s try to understand what the main is doing, for example, we can search for:

1. What functions are called in the main?
2. Are there some weaknesses that we can exploit?
3. Can we infer something from this code?

Let’s do some considerations:

* When we move with *callq* or *jump* instructions, on the fourth column we can see some info between “<” and “>”. You can notice that they all refer to the *main* or standard functions (e.g., *gets* and *puts*); this means that the entire execution flow in which we need to focus is on the *main*;
* Between *74d* and *79b* we have in sequence a call to the function *puts* (output), *gets* (input) and *printf* (output); probably, we can link these three instructions to the user-machine interaction:
  + *Puts* : “Insert the right pin [...]”;
  + *Gets* : we insert a 4 digit alpha-numeric value;
  + *Printf* : “you inserted [...]”;
* After *79b*, there are 4 blocks of similar instructions *movzbl - cmp - jne*; we might not be familiar with assembly, but don’t worry: Google exists also for these reasons:
  + [*Movzbl*](https://www.cs.auckland.ac.nz/references/macvax/op-codes/Instructions/movz.html) : Move Zero-Extended Byte to Long, i.e., we are fetching some stuff;
  + [*Cmp*](https://www.tutorialspoint.com/assembly_programming/assembly_conditions.htm): compares two numeric data fields;
  + [*Jne*](https://www.tutorialspoint.com/assembly_programming/assembly_conditions.htm) : jump not equal, and it is based on the output of a compare (cmp) instruction;
* Where these jne are jumping? Always on the same instruction: *7d1*. It means that when a mismatch occurs, we jump to this point. Just after this instruction, there is a *puts*: probably the message error? We should also notice something else, which is what there is just after the last block of *movzbl-cmp-jne* and before the instruction in *7d1*: there is a *printf ...*
* Based on the previous observation, we may think that since the PIN has 4 digits, and that the code contains 4 blocks of comparison, what the code is doing is, for each digit that we inserted, looking for matches. Ok, we can focus on *cmp* instructions: there is always a *hex* value which is compared to [%al](http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html) (a register). Those values are: 0x46, 0x6c, 0x34 and 0x67.

We did our job in a good manner! We can just convert these hex values to a characters and we have the password!

* 0x46 -> 70 -> F
* 0x6c -> 108 -> l
* 0x34 -> 52 -> 4
* 0x67 -> 103 -> g

