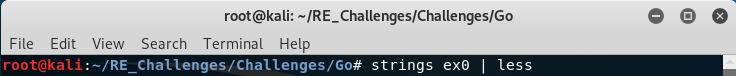
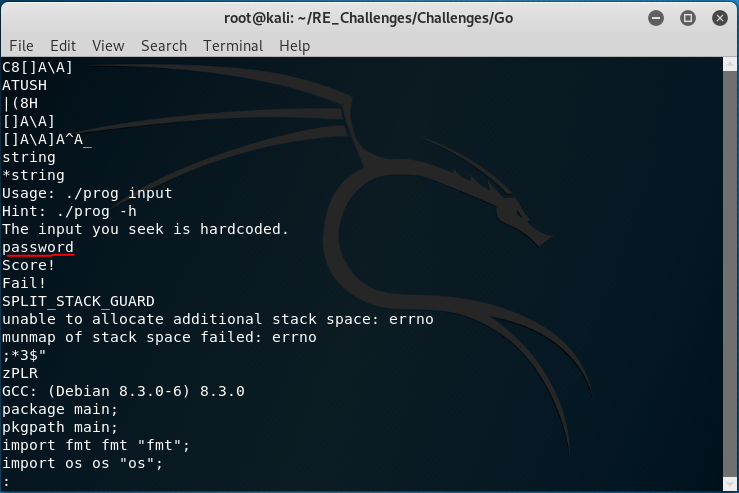
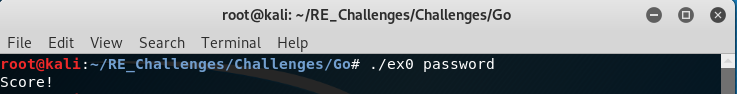
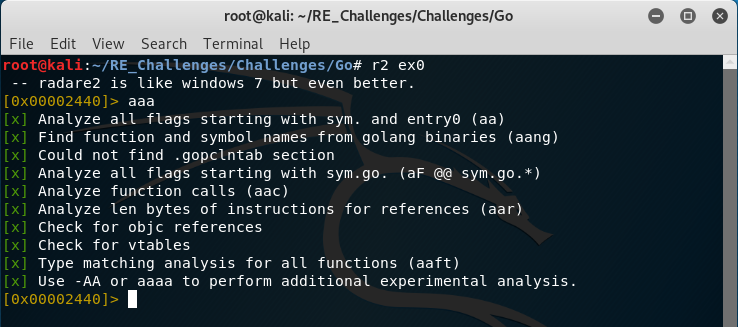
***Ex0***

If we look at the built-in help given with ‘-h’ as the sole parameter to the exercise we know that this challenge will be rather straight forward since we are just looking for something hard coded in. Lets start by using “strings” then move to “radare2”.

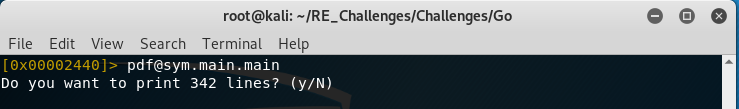
With “strings” just dumping all strings to stdout, the terminal, we will want to use a “pipe” and the program “less” so we can scroll through the output of “strings”.

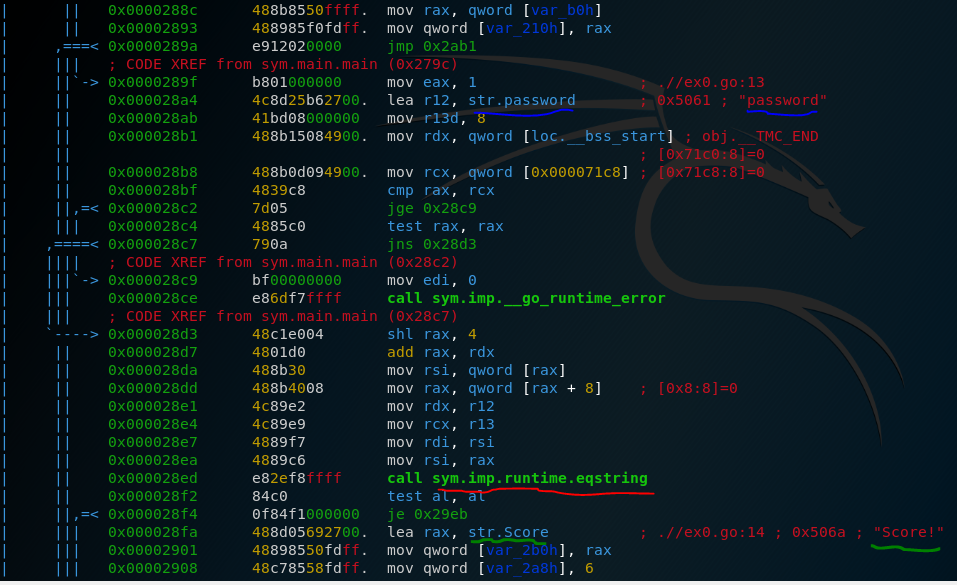
If we scroll down a good bit, we come across the strings we will get as output from running the binary. Paying attention to the order of each string, we see the word “password” right before our desired “Score!” and undesired “Fail!” messages, let’s see if that is the input the binary wants.

As expected with the first challenge it was rather easy and straight forward. Now that we know what we are looking for we can practice some with other tools such as radare2.

To start we want to open ex0 with radare2 using the “r2” alias then perform analysis, “aaa”, on the file.

Unlike C programs Go programs even when written with just a few lines of code will produce a lot of functions and those few lines of Go code don’t even start until “main.main” is called, similar to the C “main” function. With this in mind we should isolate our attention to “main.main” to not get lost in the runtime components of the executable. We can achieve this by printing the disassembly of the function at the address of the symbol main.main, “pdf@sym.main.main”.

Notice this message, this should give a good indicator that we definitely do not want to get lost in the rest of the binary since our short program is 342 lines of disassembly.

 If we take the simple approach of looking for the “Score!” message, unlined in green, we can try to quickly infer the logic going on around it so we will know what to give the binary in order to get our glorious message. For the next step we should look at function calls for any clues as to what is going on. We have two immediately before we see the first reference to our “Score!” string. The runtime\_error function shouldn’t be any of our concern,at least for now. That just leaves us “runtime.eqstring” which sounds logical for how the program could be checking our input. With this as direction we should look for strings before this call and if we look a few lines up we see “password”, underlined in blue. This string has its base address loaded into the r12 register and as we follow the execution path to “runtime.eqstring” we can see that the contents of r12 are moved into the rdx register as a parameter to “runtime.eqstring”.