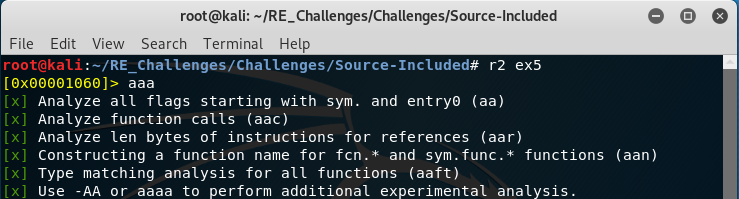
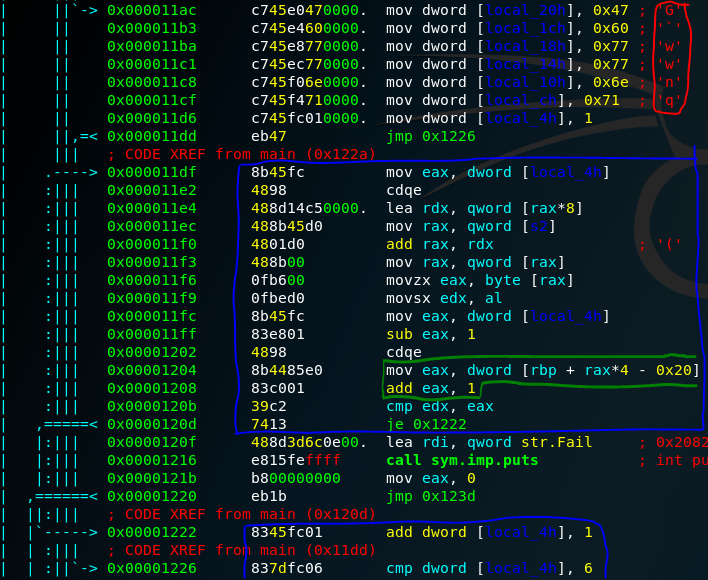
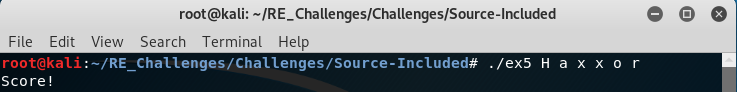
***Ex5***

We can think of this challenge as a mash up of exercises 1 and 2, so hopefully those are still fresh one your mind because this one will require just a little more attention to the assembly instructions. Don’t worry if you still are struggling with reading instructions, it will come in time. Like all things it just takes practice and just about all you can do is practice. However, just reading the instructions line by line and referencing what each mnemonic means is a pretty painful way of learning. You are reversing, not recreating, there is no sense in wasting your time trying to be able to read every single instruction and knowing what the program is doing to such a degree that you can rewrite what the original author of the program wrote line for line. Use any and all tools at your disposal to learn as much as you need as you need it, that is why the source code for the exercises is provided. Other tools to look at learning to make your job easier is any or all of the assortment of decompilers that are available for free (or paid if you have money). I would recommend ghidra’s decompiler and radare2’s r2dec as good one to start playing with. Granted, do not always trust what you read from a decompiler! The problem that a decompiler tackles is realistically an impossible challenge and will never create a line for line replica of the original source code but it is good for getting a quick overview of what the program “could” be doing from the perspective of a higher language than assembly.

We start off as usual with radare2 and basic analysis commands. Have you tried playing around with permutations to the analysis command or ever leaving it out yet? Give it a shot, you won’t break anything. Well, ever if you do isn’t that half the fun anyways?

This one will be fun, where to begin? Like always we want to look inside the main function with a “pdf@main” to get the higher level overview of what a program is doing. At least if it has a main function and yes even C programs do not “have” to have the function main, if you do not link your program with glibc or any other libc library your program will begin at “\_start”. This does not mean that the programs you have been reversing start at “main”, from the context of the developer you can view every C program linked with a libc as having main be the very beginning. However, code is ran before this which we will not get into details right now but I will note if you “[pdf@main](mailto:pdf@main)” and scroll to the very top radare2 shows you the function prototype for main. What is different? It has a parameter called envp! Yet if you look at the source code of every program we have reversed so far and will reverse this parameter doesn’t exist! That is because this is set up and used in the background before main is actually called so we as programmers will never need to give the program this info.

On to the fun stuff, we have in the red block 6 hardcoded characters being pushed into the stack and then a loop structure in the blue block. As mentioned, this is a challenge similar to ex1 and ex2 so we have good reason to suspect caesar ciphering to be included in this challenge. That is what the green block is for. Since we are pushing each of those 6 characters onto the stack we need a way to reference them and add one, the mov command is using arithmetic in its source operand to calculate the offset into the stack relative to the base pointer of the stack. The value of rax prior to this move is the loop index counter i-1 and is being multiplied by 4 because each of those hex character values are not actual char types but of type int, char=1 byte & int= 4 byte for our architecture.

Well seeing that each of the values we stored on the stack is being referenced and incremented by one we can increment the hex of each value and pass that to our program to make the comparison pass each loop iteration.