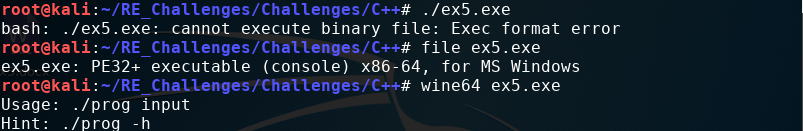
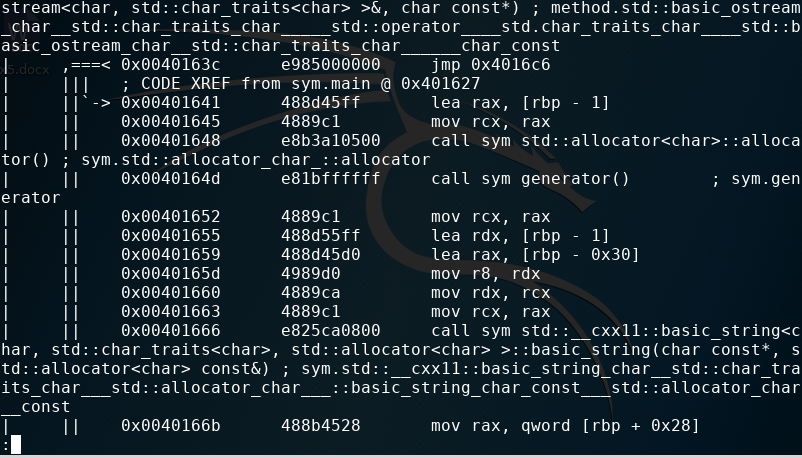
***Ex5***

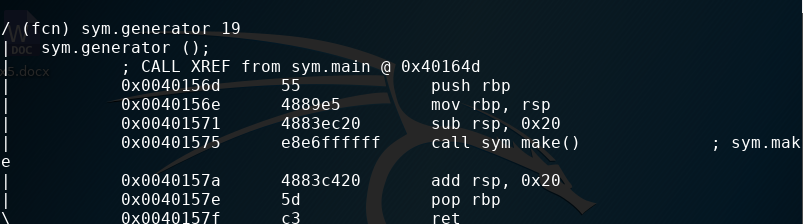
This challenge takes a new approach solely meant to keep your mind open to the fact that there is more to reverse than just linux elf binaries. As we should always start, let’s just run the program!

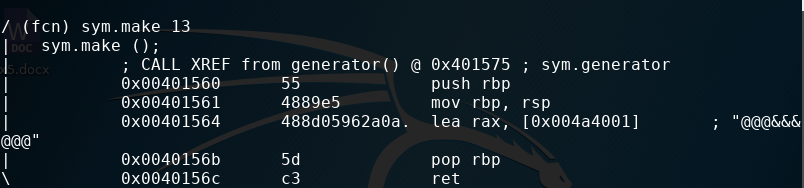
Upon running the program we notice unlike all other challenges that it does not run! So lets see what this file even is with the “file” program. This reveals that we have a Windows PE32+ file, in short a windows executable. Unlike linux, windows uses the executable format known as “portable executable”, PE, there are two version PE32 & PE32+. PE32+ represents 64-bit executables, how might we run a windows program under linux you may ask? Well, there is a program names “wine” which stands for “Wine Is Not an Emulator” meaning it does not emulate the code but acts as a translation layer for windows api calls to linux calls. Since we are dealing with a 64-bit PE file we have to use the program “wine64”.

Now that we can run the program we can easily start reversing like normal. As with most walk through documents we use radare2 as our weapon of choice. You may notice that using a [pdf@main](mailto:pdf@main)does not work, that is because radare2 sets the PE main function as sym.main, upon running a[pdf@sym.main](mailto:pdf@sym.main)we find afunction named generator being called right before a reference to the iconic “Score!” string.

 From here we should start looking into the generator function to see what return value it gives.

As with main we need to ensure we type [pdf@sym.generator](mailto:pdf@sym.generator) to look at the generator program.

Lucky for us this program seems very straight forward and gives us an apparent direction to take, inspect the make function!

Even more luck! This function is very short and simple which just returns the string “@@@&&&@@@”. Maybe it is that simple and the only challenge for this exercise was to get a windows program running under linux.

Hmm, that didn’t work? Even though this is not what we expected we did get a little insight into why is might not have worked. Bash says that & was an unexpected token, maybe the symbol itself is a problem.

With computers there is a concept known as an escaping character. Escaping characters allow for an alternative interpretation of a character from it’s original meaning. It would be worth while trying this for our challenge. To escape a character you prepend a backslash, \, to the character.

