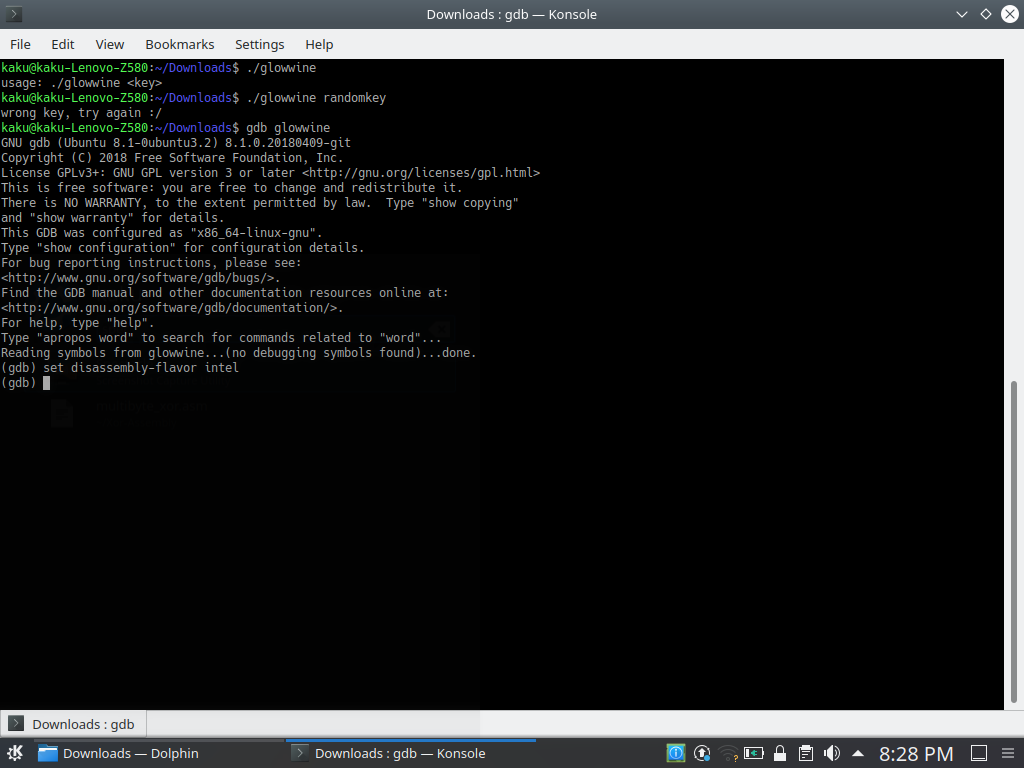
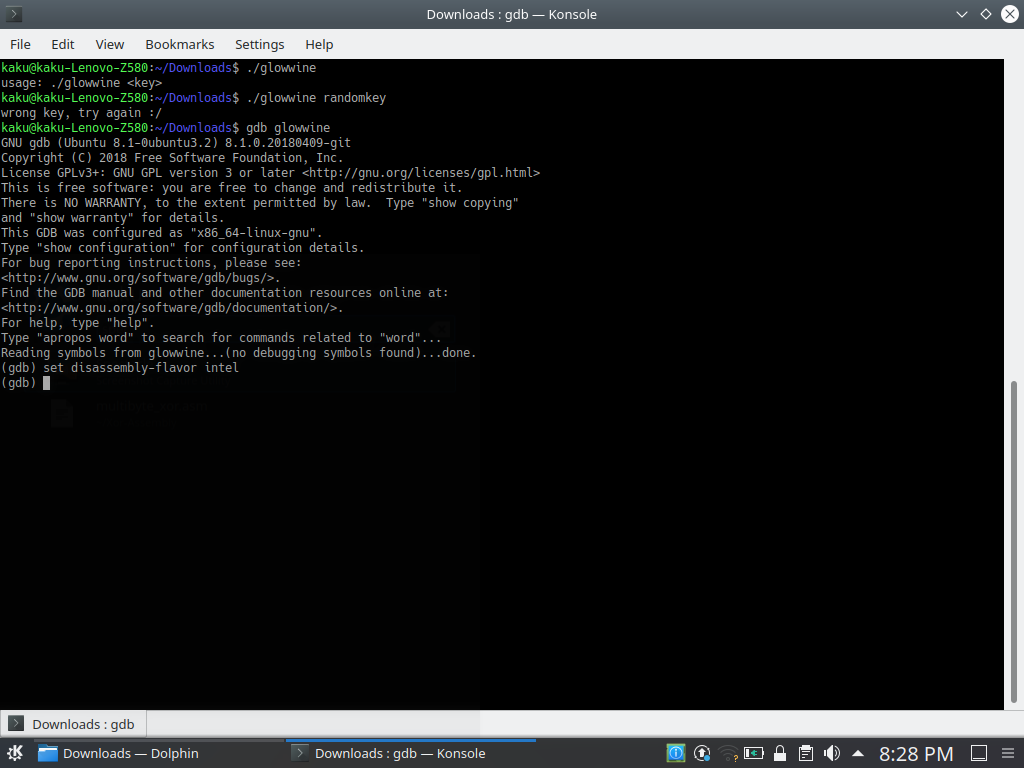
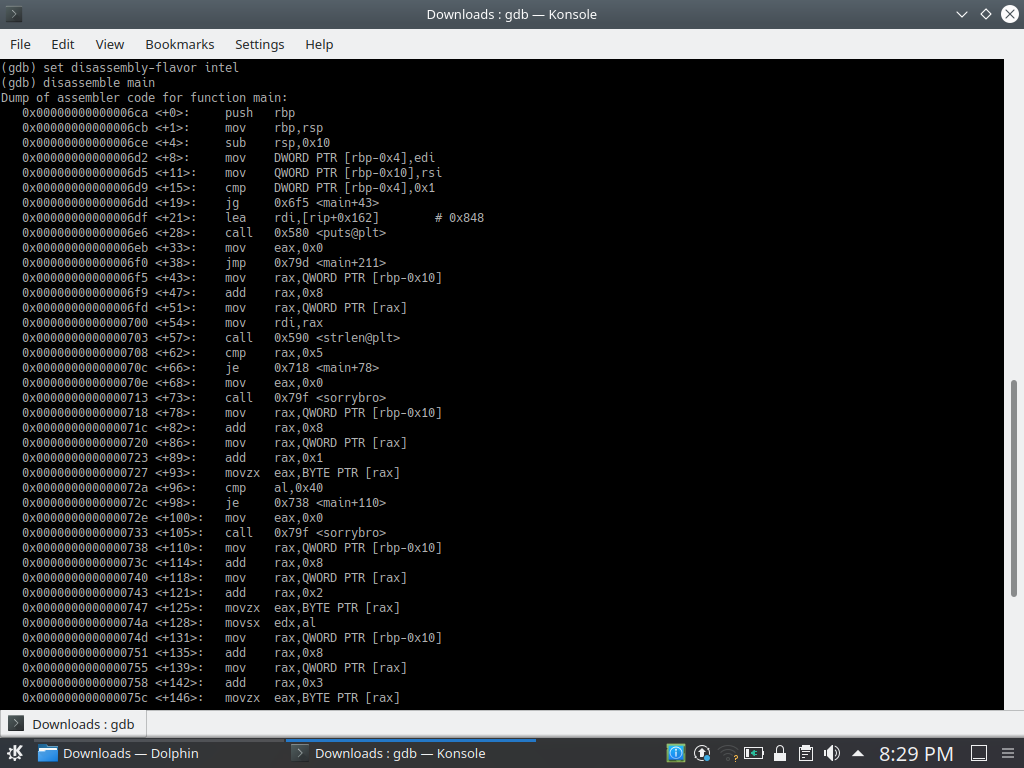
1. I am in the folder where the executable file glowwine is present. I executed the file using ./glowwine, it shows the message usage: **./glowwine <key>**, okay perharps I should pass the key as the argument while running the program so that it has some key to check

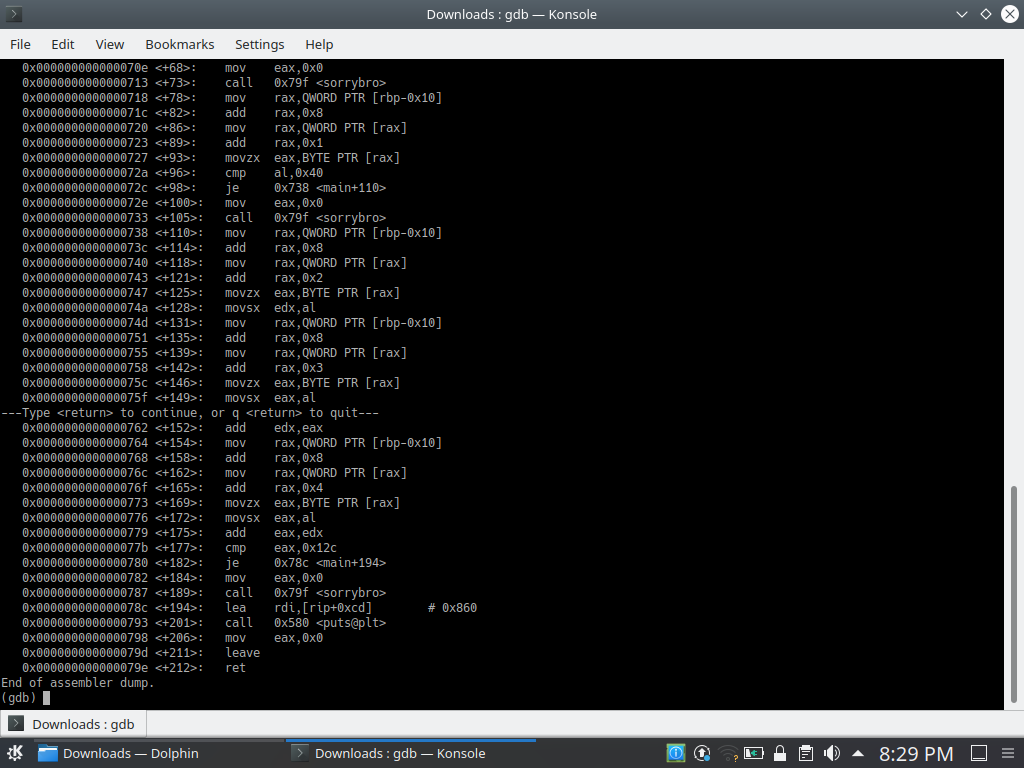


1. Now I executed the command -> ‘**./glowwine randomkey**’. It shows the message ‘Wrong key try again’. So, okay the program checks the key provided to it and returns the check status
2. Now, I decompile the program using gdb, using the command -> ‘**gdb glowwine**’ i.e. I execute the program without passing on any arguments

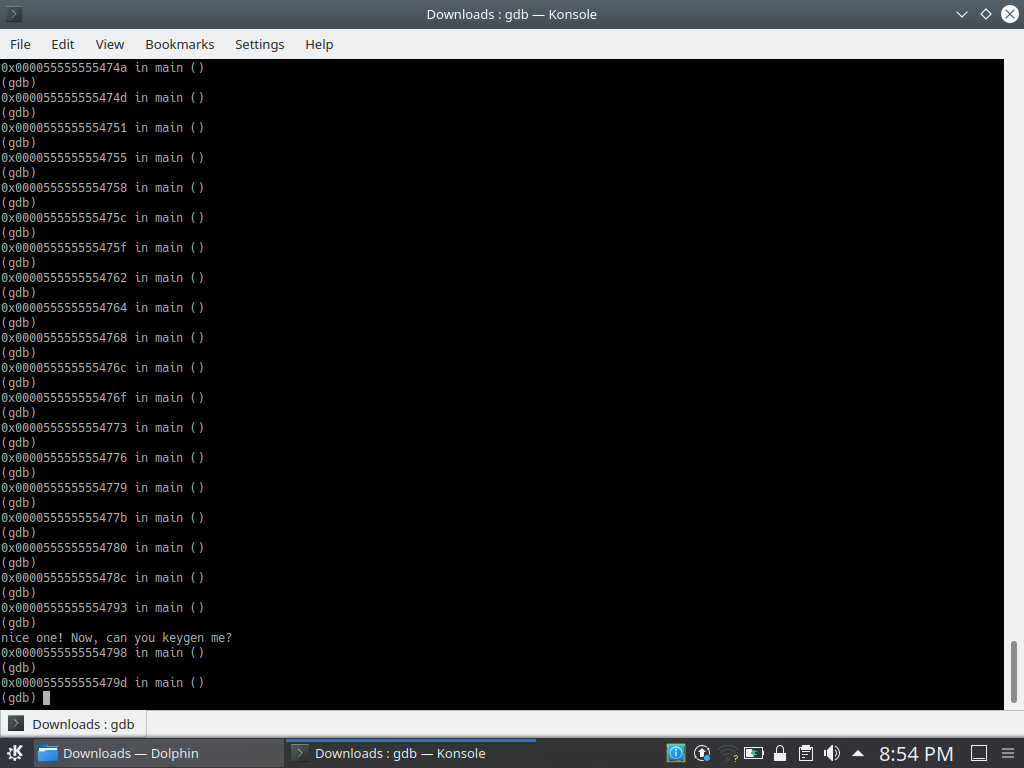


1. I add the first breakpoint in the program at the starting of the main function using the command-> ‘break \*main
2. I use the command ->’**set disassembly-flavor intel’** to change the settings of the disassemble which show the data of the decompiled program in a more simplified form without any vague signs.. just for simplicity
3. Then decompile the program to get the list of the cool decompiled code in terms of each instruction in terms of registers, using-> ‘**disassembly main**’. Take its screenshot as it will come handy a lot of time and to save ourselves scrolling up for a number of times.





1. To view the meaning and working of the first few instruction in the program you can refer to this link <https://github.com/holbertonschool/Hack-The-Virtual-Memory/tree/master/04.%20The%20Stack%2C%20registers%20and%20assembly%20code>, as the instruction have been well explained and I thought I should share this..
2. Run the program using -> ‘**run**’. The execution stops at the starting of the main function at the first breakpoint we had set. Check the value in the registers using ‘**info registers**’. Most of the registers show some address values except some like rbx which shows 0, rdi shows 1.
3. If you again run the program using some arguments using -> **run randomkey** and then check the info of registers, then we will see that the value of rdi changes to 2. Interesting thing to notice is, if I execute the command -> **‘run arg1 arg2 arg3’** and then see the info in registers then I see the value of rdi it shows it to be 4. This means it stores the value of number of arguments +1.
4. Okay I think, enough info to take from here let’s move ahead. Use the ni command to move ahead until we reach compare function.
5. Use the ni command to move forward until we reach the function 6d9 which compares two values. It compares DWORD PTR [rbp-0x4] with 1. Tracing back, 2 instructions above we has stored the value of edi in DWORD PTR [rbp-0x4], which is actually the value of rdi. And this the value of number of arguments +1. Hmm, so we are comparing 1 with this.
6. Moving to the next command, which gives instruction to jump only if the previous compare command yields a greater result. This means, if number of arguments + 1 > 1 (number of arguments > 0) then jump to 0x6f5 other wise go ahead.
7. So I will once run this program using only -> ‘run’ and once using -> ‘run randomkey’ and see where we reach in each of the case after the jg command
8. When I use ‘run’, after the jg, if I enter ni, I am sent to 6df, moving ahead I get a message printed-> ‘usage: …’ the same we got in the beginning when I executed the program without any arguments.. now this makes sense so here is the first check which checks whether any argument is provided to the program which is key obviously.
9. Okay now I run the program using -> ‘run randomkey’ and reach the jg command. Now I move ahead and I am sent to 0x6f5 as mentioned in jg command. Let’s move ahead
10. Now I move ahead using the ni command until I reach 703 function. Which is the strlen function. Reading its documentation on the internet, it takes some value from the eax registers and stores the string length of value at that address in the eax itself. So execute this command and then use ‘info registers’. We can see that the value stored in the rax register is 9, which is actually the string length of the argument we entered i.e. randomkey, okay cool. Now to confirm you can run with some other arguments and check if the string length of the argument is the one which is stored in the rax register.
11. Alright, after you have satisfied yourself that this is the case, let’s move ahead. Now, in the next command the value in rax is compared with 5. If it is so then we are sent to 0x718 function, otherwise, we are sent to ahead to the <sorrybro> function which messages, wrong key try again. So alright this means the correct key has string length of 5. So we got our first hint.
12. Execute the program with some 5 length key this time and see where the execution goes after the 0x70c command. It will be going to 0x718.
13. Now moving ahead, use ni until you execute the 0x727 command at last and just about to execute 0x72a. The next command compares the value in al register with 0x40, which is the ASCII code for @. Now use -> ‘info register al’ to get the value stored in al in your case. Notice that it stores the ASCII code of the second character of your entered key. For eg. Let’s say I had entered **abcde** as the key. Then it would be showing 0x62 which is the ascii code of b.. Hence it compares the second character of the key with @. If correct, we move ahead otherwise we would be sent to <sorrybro> function again. So the second character of the key is @
14. Now run your program with key which is of length 5 and 2nd character as @, such as a@cde. Now moving ahead, till we reach to the 0x747 and execute it. Now check the value of the registers. rax has the value 0x63 which is the ascii code for c which is again the third character of our entered key… So now if we observe the set of instructions, first from 0x7018 to 0x727 and second from 0x738 to 0x747, there is one noticeable difference. In first, we added 0x1 to rax, while in second, we added 0x2 to rax and this makes the difference of storing the 2nd and 3rd character in the eax register in different cases.. okay so whenever we find such sets of instruction, we should understand that what would it do.
15. Moving ahead, after the instruction 0x747 we execute 0x74a which movsx al into edx. In our case the value in al is 0x63. Then executing the movsx command into edx would place 0x63 into edx.
16. Then we have similar set of instructions as described in step 20, which stores the value of 4th character in eax. Then it adds it in edx, which means now addition of value of 3rd and 4th character is stored in edx now. And then further instructions select the 5th character (from instruction 0764 to 0x776) and then value of edx is added to eax and stored in eax. This means that now eax stores the value of 3rd+4th+5th character.
17. Then next instruction compares this value to 0x12c. If they are equal then it moves to 0x78c otherwise it sends to <sorrybro> function.. which gives us what all people would do without a key.. but being a hacking enthusiast, we would try to have sum of last three characters as 300. Like ddd,cde and many such cases would be there…
18. we enter such keys, move ahead and yes!! We get the message “nice one” this is where we wanted to reach and all we got is some is constraints on the key.



1. The length of key is 5. The second character is @. The sum of ascii codes of last three characters is 300.
2. Some of such keys would be a@ddd,3@cde,v@ddd
3. Check it using directly executing glowwine with these keys as the arguments.