0.1 Problem name and description

1. **Problem Name.** `level1` by `sudo0x18`
2. **Description.** This is a text-based program written in a C/C++ family language. Whenever you execute the program (called `level1`), you are prompted to put in a password, which presumably has two outcomes: success and failure.

0.2 General Approach

Given the fact that the program is CLI-based as well as written in a C-family language, I intuitively just want to disassemble the program using `gdb` and just walk through the call stack, since it seems like it's just a simple string comparison program.
Here is just a list of what I did:

1. Obtained filetype for the binary as well as printable strings within the binary
2. Ran the program through `gdb`, creating breakpoints at the password checking function.
3. Analyzed the disassembled instructions.

0.3 Prerequisite Knowledge + Keywords

1. Basic knowledge of `asm` language and
2. Basic commands such as `mov` , `cmp` , `j` , and `call`
3. **Keywords.** String comparison

0.3.1 Tools Used:

1. **Disassembler.** `gdb` (GNU Debugger) or some other program like Ghidra, IDA Pro, etc.

0.4 Walkthrough

0.4.1 Step −1: Running Malware Analysis Commands

As per the advice of other users, it appears that principally it maybe helpful to run a compiled program through the following commands:

1. `md5sum`: Hashing command that executes that **Message Digest 5** hashing algorithm directly on the compiled program's binary, which will return some value. in the world of malware analysis, compare this hash with what the original program author deems as the original hash for the program. Different hashes $\Rightarrow$ tampering.
2. `sha1sum`: Hashing command that executes the **Secure Hash 1** cryptographic algorithm. Same principle as `md5sum`.

Nothing interesting there.

0.4.2 Step 0: Getting Metadata on File

Now, we can actually do helpful things. First thing that I want to do is checkout the **filetype** of the file, as well as look at any strings that are encoded within the file.

1. **Check filetype.** I used the `file` command in Linux and found that `level1` is a ELF 64-bit binary, which is pretty typical for all Linux executables.
2. **Find printable strings within binary.** used the `strings` command to find all notable strings within the binary.

Notable things that I got from this process are that I can can see not only the prompt for putting in password, but I also see the prompts for when I get the password correct or incorrect. Notable functions also included in the binary are `main` and `checkPass`, which of course, likely refers to checking the password that I put in. `gdb` time.

0.4.3 Step 1: Running the program through `gdb`

My initial idea was to just run `gdb` and create a breakpoint at the function `checkPass` and see what happens whenever we type in an incorrect password.

I used the following commands:

```
  $ gdb level1
gdb> b checkPass
   gdb> run
```

0.4.4 Step 2: Running through instructions using `disassemble` and `ni`

Whenever I break in the `checkPass` function, I decide to step through the instructions by using the `ni` (next instruction) command. After stepping through the function quite a bit, I get the following intuitions:

1. Whenever `checkPass` is called, our input is presumably passed from `%rax` into `%rdi`
2. We are checking characters of the string *individually*, since we can observe the usage of the `cmp` and the `jne` commands.
3. It makes sense that if the comparison doesn't work, that `checkPass` returns a `false`, which signals to `main` that an incorrect password was inputted.

The following `cmp` statements are made, with their corresponding letter:

`cmp AL, 0x73` $\Rightarrow$ s
`cmp AL, 0x75` $\Rightarrow$ u
`cmp AL, 0x64` $\Rightarrow$ d
`cmp AL, 0x6f` $\Rightarrow$ o
`cmp AL, 0x30` $\Rightarrow$ 0
`cmp AL, 0x78` $\Rightarrow$ x
`cmp AL, 0x31` $\Rightarrow$ 1
`cmp AL, 0x38` $\Rightarrow$ 8

Of course, this means that the password must be `sudo0x18`, which is correct.