Program 1

Password: ycMBieqJSCxyJlGEozlp

The output of mystrings for program 1 was much longer than the other two programs. I inferred that this was to hide the password as a raw string in the program. I then wrote a python script to test all outputs as a password.

```
import subprocess
import sys
import shlex

mystrings = subprocess.Popen(shlex.split("./mystrings tdm43_1"), stdout=subprocess.PIPE)
(strings, unused) = mystrings.communicate()

for password in strings.split("\n"):
        call = subprocess.Popen("./tdm43_1", stdin=subprocess.PIPE, stdout=subprocess.PIPE, shell=True);
        (output, err) = call.communicate(password);
        if(output != "Sorry! Not correct!\n"):
                sys.stdout.write(password + "\n")
```

The only output was ycMBieqJSCxyJlGEozlp.

Program 2

Password: tdm43 (or the username of whoever is executing the programming)

The python script didn't work on this one. This implies that the password is being generated by the program. Looking at the much more manageable mystrings output I could now see distinct sections of output, notably including a section at the top that looked like functions and a section right below that looked like raw strings. The functions included the standard IO bunch and getenv. getenv takes an environment variable as a string and returns its mapping as a string. Guessing that this function was the password generator and that the environment variable was 'user' from the raw strings portion mentioned above, the password was found. I then looked at the disassembly output of main in gdb to check that the program was getting the name of the user, then comparing it against the user input which as best as I can tell it was.

Program 3

Password: 8 of any of <>|~& and 8 characters that are not <>|~& in any order.

Neither of the two previous techniques got me anywhere. Opening up gdb to see what main looked like I found that I couldn't set a breakpoint at main. I then spent several hours getting adjusted to using gdb and reading x86 assembly. During this time I discovered that I could set a breakpoint at getchar, print out the back trace to find the address of the function that called getchar and then scout out the disassembly from there. I'm not sure what this function calling getchar is as there were some dynamic linking/loading done at the early part of the program, but I'll be calling it main. After stepping through main for sometime, I became confident that the program was reading in 16 characters and then

comparing each character in a block of assembly that looks like a switch statement. If any of the characters matched one of the cases the program would increment some variable on the stack, and after every character was compared if the variable equaled 8 it would call printf instead puts. The switch case had 5 cases for the asci characters >&<|~ and from there a valid password could be constructed.