James Hahn
CS0449, M/W 3-4:15pm

=== File #1 - jrh160_1 ===

For this file, I ran the program and found that when I entered the word "james", the register $edi would be "ames". Then, I examined the $esi register and found the passphrase "BGNdaSfXyOvCmnUmvchqaeIeXaGm" and this seemed like it was the answer; sadly, it was not. Finally, I ran my strings program on the file and found the string "SBGNdaSfXyOvCmnUmvchqaeIeXaGm". This definitely seemed like it was the answer because compared to the previous string stored in the $esi register, it just had an added S character at the beginning of the string; the program would chop off the S just like it did with "james". I passed this into the program and it worked! I assume it was the chomp() function that chomped off the first letter in the passcode,
which allowed this passphrase to work.

Answer: SBGNdaSfXyOvCmnUmvchqaeIeXaGm

=== File #2 - jrh160_2 ===

For this file, I initially tried running my strings program, but I did not find any useful information from that. So, I immediately decided to use the disas command in the gdb debugger.
It took me a while of entering in input, such as "james", "banana", and "new york city", but I always saw a recurring value of "MO" from the register $edx after chomp() was called, so I eventually tried that as the input and it turned out to be the correct answer. Along the way, from the assembler dump, I noticed there were calls to the computer's clock and occassionally I would spot that some registers would store the current day ("Monday") and another would store the month ("October"), so I tried the program on different days and noticed it takes the first letter of the day (M from Monday) and the first letter from the month (O from October) and combines them into a 2 character string for the answer, so there are a wide variety of answers for this file.

Specific Answer: MO on a Monday in October
General Answer: The first letter of the current day, capitalized, combined with the first letter of the current month, capitalized. The input should be a 2 character string.

=== File #3 - jrh160_3 ===

This file was by far the trickiest. Immediately, I was thrown off; this program didn't have a main() function. So, I did "b *0x0" to set a breakpoint at address 0, which immediately caught the first line that the program came across and broke there, which allowed to to see the first function that was called, _dl_start. This took me about a day to figure out. Little did I know, this was the least of my worries.

I started putting breakpoints at every single function I came across and eventually became so desperate that I resorted to stepping line by line through the program. After calling the "info functions" command in gdb, the debugger output, on the first line, "getchar", so I figured that would be a useful function to break on.

With several runthroughs of examining getchar and the registers during the calls, it was evident that there was a register that stored the count, or number of characters, in the input string. Quickly, I discovered that the string had to be 10 characters long because if you have a breakpoint on getchar and only pass in 1 character at a time, the function will be called 10 times before it moves onto more sophisticated calculations and other function calls.

And perhaps the longest section of this challenge was to find the actual calculations taking place to validate the input string or to find a series of strings that were being read in from some external files, as I noticed the program depended heavily on many dynamically linked libraries. So, after about 4 or 5 days straight of looking at this program, its x86 instructions (which I was very unfamiliar with), and all of its different functions, I was able to decipher some hints. I found places where the program looped through the input one character at a time, performing a calculation on each one, and then deciding if it was "valid" or not. In these series of instructions, I found that a character was read in, "g", for example, its ASCII value was stored in a register, so 103, and then 101 was subtracted from that file, so we'd currently be at the value 2 in the register. Next, the program would shift 1 left by this number (so 1 << 2), which resulted in 4. The last calculation was an AND between the calculated number thus far (4) and 0xe281; it took me the longest time to find this AND because there were many other ANDs in the program and 0xe281 seemed like an extremely random number to AND with. Anyways, this AND ended up with the value 0.

**Side note: The above calculations can be visualized as follows (in this example, the current character is stored in $eax):

```
        1 << ($eax - 101) & 0xe281
```

**Side note: Above, when the program subtracted 101 from the current character's ASCII value, it later uses this value, compares it to 15, and sets up a sort of "if statement" in this psuedocode:

```
if( (char_ascii_value - 101) <= 15 && (above calculations != 0) ):
        //do something
```

So, the key trick to this program was that it also kept a counter variable, while looping through the input string, to keep track of when the current character met the criteria brought up in the aforementioned psuedocode if statement, where the ASCII character value - 101 has to be less than or equal to 15 and the calculations mentioned in the previous paragraph are not equal to 0; if the criteria was met, the counter variable was incremented. After looping their the entire string, it checked if the counter variable was equal to 7 or not. If it was equal to 7 (there were exactly 7 of these "special" characters in the string), then the program would print out a Congratulations message, and otherwise, it would print out that the input was Incorrect. The last most important requirement, or instruction/call that I noticed was a call to toLower(). This call converted all of the characters to their lowercase equivalent, so "E" to "e", or "G" to "g", before it performed any of the above calculations.

To wrap up, after countless hours of exhausting testing of inputs, I noticed several lowercase characters that met the previously mentioned requirements: e, l, n, r, s, t. During this process as well, I tested many different kinds of characters, but none of them met the criteria of the program. One last thing to mention, because I noticed the toLower() method earlier, I remembered that the uppercase version of the valid characters should be acceptable as well, and they were.

And finally, to summarize:

This program takes an input string of exactly 10 characters in length and the string must contain exactly 7 of these following characters: e, l, n, r, s, t, E, L, N, R, S, T (repeats of these characters are allowed as long as the user is trying to meet the requirement of there being 7 of them). For the remaining 3 characters, they can be anything EXCEPT for the valid characters that were just mentioned.

Example of acceptable answers: elnrsteja, %ESrtslr%%, and ajelnrste