Critical Thinking 2

Gregory Norris

CSC 450 – Haseltine

Colorado State University Global Campus

Critical Thinking 2

C and C++ are not type-safe languages. This means that strings of input which are inserted into a C or C++ program have the potential to be executed as code. Accepting user input in the form is strings is generally unavoidable, yet it also creates several openings in security which must be addressed. It is important to note that there is practically no difference between data and executable code in a C++ program apart from the context it resides in. This week’s assignment is to write a program which takes user input and reverses it. Since the user is providing input, special care must be taken to ensure that the user can not enter something which will break the program.

**C Style Strings**

In C programming language, strings are handled by fixed-size character arrays. These arrays are vulnerable to several different errors and exploits which can cause undesired operation of the supporting program. Since a C-style character array stores strings as single-characters within a fixed array, it is vulnerable to buffer overflow when the input given exceeds the capacity of the character array. Strings are stored as characters in the array, with a null character ‘\0’ at the end which delineates the end of the string. A string ended with a null character may be shorter than the character array. C-style character arrays which are copied without a null character may confuse additional operations working with the string, as they are unable to tell where the string ends, this could lead to the execution of random code within the program which creates vulnerabilities such as Null Terminator Errors.

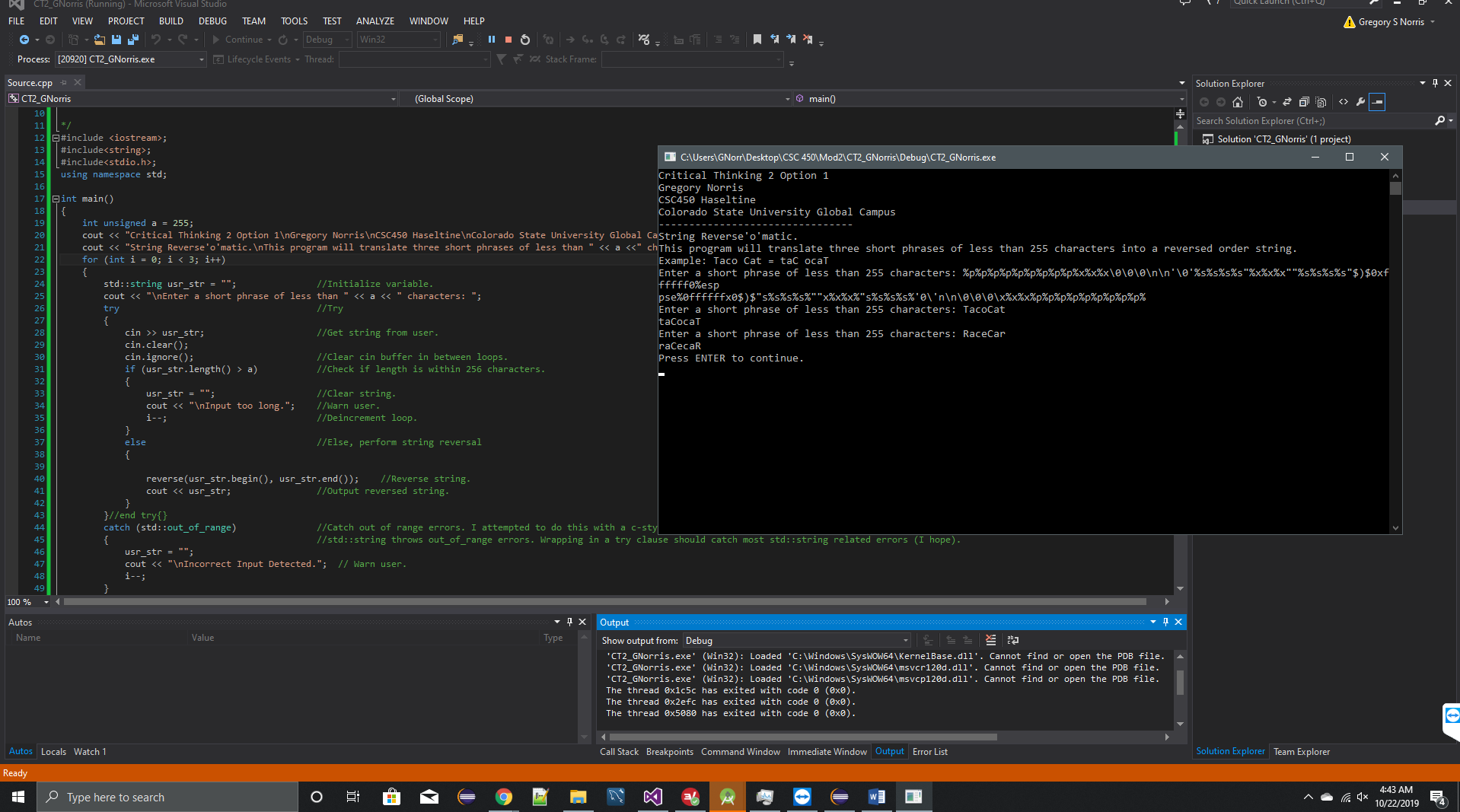
**C++ Style Strings**

C++ has a strings class which overcomes many of the limitations of a C-style fixed-length character array. C++ strings can have a variable length and are generally not vulnerable to buffer overflow since they can allocate memory dynamically. It is possible to insert enough data into a C++ style string to cause issues by reserving too much system memory.

**Critical Thinking Code Assessment**

The Source.cpp file attached contains my submission for this week. Initially I attempted to complete this assignment using C-style strings to become more familiar with the types of vulnerabilities which must be accounted for. I attempted to use both cin.width() and \_fgets() for accepting user input. The c-style program was prone to several errors and would crash when I provided strings such as “%p%p%p%p%p%x” and including semi-colons and escape characters like \0. After testing, I realized that I would likely have to write a sanitization class to replace system characters and this seemed like a lot of effort when C++ strings handle many of these details by default.

I rewrote the program using C++ std::string and found the program much easier to work with. When considering how the program accepts input, I couldn’t find much information related to the security of cin and cout, although I found tons of exploits utilizing formatted strings and printing, such as printf(%s) and decided not to use formatted printing. Although std::string does not usually cause buffer overflow, I wrapped part of the program in a try statement to catch std::out\_of\_range errors in the event that any part of the program attempts to read or write data stored outside of the string. This re-written program passed my initial string tests and seems to be relatively secure. For added measures, I also included statements to flush out the cin buffer after a successful transfer to the string variable. I am not aware of any specific exploit regarding cin, but it seems like best-practice to purge this information immediately after it is no longer needed.

****