This week, our critical thinking assignment involved creating a program that requests three numbers from the user, places them in variables, creates pointers that reference those variables, and prints out all this to the user. When using input and pointers, there are a few general things to look for when it comes to the security of a program, including memory management, memory failures, and input validation.

Memory leaks can cause unexpected behavior in systems. These can occur when the program does not delete the dynamically allocated memory for several reasons. In this program, we can see this possibility when the program terminates before completing and deallocating the memory with the delete function. If the program terminates early, then prt1, 2, and 3 will still be assigned to memory and stay there until deallocated some other way or overwritten, which can take up significant resources in the system when working with larger models. Another memory issue can be an all-together failure of memory allocation. If we have an issue with allocating memory, the program will throw an exception, but this program does not have any way of handling this exception. We could use a try/catch block in this to specifically handle memory allocation and catch a std::bad\_alloc exception and handle it appropriately. Here we can print out a message to the user saying the memory allocation failed and clean up the system of any allocated memory or other actions taken during runtime. Lastly, we need to ensure we do not create a dangling pointer. If we deallocate the memory a pointer is referencing, there is still a possibility of accidentally using that pointer to access that location further on in the program. A dangling pointer can cause havoc on any function or action that inadvertently calls it during run time or compilation. We can set up several stopgaps in our system to avoid these memory issues. To avoid dangling pointers, we can set our pointers to null after using the delete function to remove the memory that the pointer references. To avoid memory leaks and un-handled allocation failures, we can use a try/catch system to deallocate these variables in memory in the case of wrongful termination of the program or failure of memory to allocate.

The next issue is input validation. First, we must verify that the input is of the correct form our program requires. This program assumes that the user will enter valid input, but accidents and maliciousness exist, so we cannot consider our code secure if we rely solely on this assumption. We can verify that the user has entered three integers to avoid this by checking with an if statement. In addition, we can perform a bounds-check on the input to verify that input values do not exceed the range of an int variable. If they do, it will result in a buffer overflow error in our program and cause unexpected behavior or errors as the overflow overwrites surrounding memory to compensate. We would have to verify our input against both positive and negative integer limits as we are using signed integers, so we can set our if statement to compare our input to std::numeric\_limits<int> min and max to verify that our input is within the bounds of an int before writing it to memory. In addition to this, we can, at the same time, check to make sure that the user indeed entered three integer inputs. If the user enters less than three, we will run into an error in assigning our variables. If they entered more, they could lead to erroneous data or an overflow. Once we have checked that our user entered only integer values and that they only entered three, we can use std::cin. ignore () to clear the remaining characters from the input. Taking these actions ensures we have exactly three integer inputs from the user.

Lastly, it is always good practice to use an analysis tool that can scan our code and point out potential security flaws. These can be extremely helpful in verifying code prior to deployment. This, in addition to thorough reviews by the development team, quality assurance, and outside agents can dramatically improve the security of our code.