A careful review of the given code has shown a handful of possible vulnerabilities and potential error-related issues, including input validation, buffer overflow, and securing input handling with '>>.' Throughout these last few modules, input validation has been brought up repeatedly and with good reason. Input validation is essential, which brings me no joy to say. This is module 5 and it still seems to be an issue in my submissions, but changing it now feels disingenuous. When taking in any input, we should always check to ensure that the input is the type and size our system is expecting. We run into numerous issues if we take in input for a double that is larger than the memory size allocated to a double. Another simpler issue with the input in this system is that it needs to ensure the values set in are positive. If a user were to enter a negative number for overtime hours or rate of pay, this would drastically impact the system's calculation of final pay. While the system will not crash the system, the output will be wildly erroneous. Verifying that the input is of the correct size, sign, and type would avoid many simple issues.

Buffer overflow is another issue that is easy to overlook. As stated, if the user enters an input larger than the memory allocated for a double in this program, an overflow can occur where the system writes the excess data to memory locations not allocated to that element. Overflow can overwrite previously written data and lead to issues further on in the system when it tries to call the memory location that has just been overwritten. In addition, it can also lead to serious security vulnerabilities. If the user had malicious intent and used the overflow to inject their code into a memory location in our system, they would have the ability to inject any number of nefarious things.

An overflow can be avoided with a simple check to ensure that the input the user gives is precisely the input our system expects. One easy way to do this is to switch from the '>>' input operator to the 'std::getline()' to bring the input in as a string, then validate it by checking length and content, then finally parse it into a double. In hindsight, this strategy seems simple and obvious and will be an essential tool in further modules. Validation through this method will avoid any overflow, no matter what the user decides to input. Previous code I have written used this method, but not as a security measure, simply as a convenience. Thinking of this as a type of security system has reframed my understanding of using input. Implementing the 'getline()' function as described here is a simple way to avoid many of the issues related to buffer overflow and will lead to a more secure way to handle user input.

Another issue, one with error-related occurrences, is a simple lack of error handling. While this program is simple and will likely not result in many errors during runtime, there is still the possibility that one may arise. In that scenario, no systems are in place to handle those errors, causing the system to crash. Putting in a block that can catch an error and handle it appropriately is key to a smooth system. For example, a simple check to ensure the user entered input. As is, this program will drop to the next line and continue waiting for input without letting the user know anything is amiss. Having the system check and display a message when the input is empty, and that the system is awaiting further information, would make this program more user-friendly. Another example would be during calculations. If, for whatever reason, the calculations done by the system fail, there should be a block that handles that failure, advises the user, and possibly restarts the program, asking for the input again. These systems are necessary for the program to avoid failing or crashing without advising the user of the issues.