Terry Christopher Johnson Jr

07/03/2025

SNHU - CS 405

Module One Activity: Numeric Overflow Coding

This assignment critically examines a significant data integrity vulnerability inherent to a banking application, specifically focusing on numeric overflow and underflow within the fundamental add\_numbers() and subtract\_numbers() functions. The original implementation permitted integer values to exceed their maximum representable limits or fall below their minimum thresholds, leading to erroneous "wrapped around" results. Additionally, floating-point calculations were susceptible to yielding infinite values when limitations of representation were breached. The primary objective was to bolster the security and reliability of these mathematical operations by implementing defensive programming strategies designed to detect, prevent, and report such arithmetic errors.

To address these vulnerabilities, a comprehensive modification of the add\_numbers() and subtract\_numbers() template functions was undertaken. For integer data types, proactive validation measures were instituted prior to each addition or subtraction. These validations involved comparing the anticipated result against the boundaries defined by std::numeric\_limits<T>::max() and std::numeric\_limits<T>::min(), employing a robust approach such as evaluating the condition (result > std::numeric\_limits<T>::max() - increment) to prevent wrap-around during the validation stage itself. For floating-point data types, the std::isinf() function was utilized post-operation to ascertain whether the result had ascended to either positive or negative infinity, indicating that a computational limit was exceeded. In all instances of imminent overflow or underflow detection, the execution of the operation was immediately halted, thereby averting the completion of any erroneous calculations and ensuring that no incorrect values were generated or disseminated.

To provide clear communication regarding the success or failure of these operations, the return type of both arithmetic functions was redefined to std::pair<T, bool>. In this structure, the first component (T) encompasses the computed result—or a safe initial value if an anomaly was detected—while the second component (bool) serves as a critical status indicator: true signifies successful computation, while false indicates the identification and preventive handling of an overflow or underflow scenario. Correspondingly, the test\_overflow() and test\_underflow() functions were refined to effectively interpret this boolean status, delivering explicit console output that either presents the correct numeric outcome with the designation "Overflow Status: false" for successful operations or indicates "Operation Failed (Overflow Status: true)" when a critical issue was identified. This methodology, which employs C++ standard library features such as std::numeric\_limits and if constexpr for type-specific compile-time logic, guarantees a robust and efficient solution across a spectrum of numerical data.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.

The implementation effectively addresses the identified vulnerabilities, highlighting the significance of secure coding practices in financial applications. By systematically detecting, preventing, and reporting numeric boundary violations, the modified code greatly enhances data integrity and contributes to a more secure and reliable banking system. This approach establishes predictable and repeatable behavior for essential arithmetic operations.