Lee Kitchen

CS 405

30SEP2024

5 3 Activity: Static Code Analysis

In this static testing exercise, I employed Visual Studio and Cppcheck to analyze a provided codebase for common coding issues. Both tools detected multiple warnings, errors, and potential vulnerabilities. Below is a detailed breakdown of the findings from each tool, along with a comparison of their strengths, weaknesses, and risks associated with the identified issues. For each issue, I have described the nature of the problem, assessed the associated risks, and suggested mitigation strategies.

**Visual Studio Analysis**

Visual Studio's built in static code analysis found several notable issues:

* Uninitialized Variables :
  + Warning C26495 : Variable 'A::x' is uninitialized (Line 42).
    - Risk : Leaving a variable uninitialized can lead to undefined behavior when that variable is used, potentially causing unpredictable runtime errors.
    - Mitigation : Initialize all member variables in constructors to avoid undefined behavior and ensure predictable program execution.
* Buffer Overrun :
  + Warning C6386 : Buffer overrun in 'buf' (Line 66).
    - Risk : Writing beyond the bounds of an array leads to memory corruption, potentially causing crashes or security vulnerabilities like buffer overflow attacks.
    - Mitigation : Ensure that array accesses are within the valid range. Use bounds checking or a more robust data structure (e.g., `std::vector`) that automatically manages array bounds.
* Exception Handling :
  + Warning C4297 : The function 'MySpecialType::DontThrow' is declared noexcept but throws an exception (Line 52).
    - Risk : Throwing an exception in a function marked as `noexcept` can terminate the program. This could result in unexpected crashes.
    - Mitigation : Ensure that no exceptions are thrown from functions declared with `noexcept`. If exceptions must be thrown, remove the `noexcept` declaration or refactor the function.
* Type Safety :
  + Warning C4806 : Unsafe comparison between `bool` and `int` (Line 129).
    - Risk : Comparing a boolean value with an integer may lead to logic errors or incorrect program behavior.
    - Mitigation : Avoid comparisons between incompatible types (e.g., `bool` and `int`). Refactor the code to ensure that comparisons are type safe.
* Optimization Suggestions :
  + Several functions (e.g., `foo`, `work\_with\_arrays`, `do\_something\_useless`) were flagged as candidates to be made static.
    - Risk : While not critical, making these functions static can improve performance by avoiding unnecessary object references.
    - Mitigation : If the functions do not depend on instance variables, declare them as static to improve efficiency.
* Console Output and Debug Error :
  + Console Output : "Welcome to the Questionable Code Test!"

Assertion Failure : `assert(my\_function() == 3)` failed (Line 129), triggering an abort call.

* + - Risk : The failed assertion indicates a logic error in `my\_function()`, causing the program to abort during execution.
    - Mitigation : Investigate and correct the logic in `my\_function()` to meet the expected return value of `3`. Alternatively, modify the assertion to align with the correct function behavior.

**Cppcheck Analysis**

Cppcheck provided additional insights into the code, revealing different types of issues:

* Memory Issues:
  + Error : Dangerous assignment in function (Line 59), where a local auto variable's address is assigned to a function parameter, potentially leading to stack corruption.
    - Risk : Local variables are stored on the stack and are destroyed when a function exits. Assigning a pointer to a local variable's address can lead to invalid memory access after the function ends, causing crashes or undefined behavior.
    - Mitigation : Avoid returning or assigning pointers to local variables. Use dynamic memory allocation or pass by reference.

* + Error : Function 'MySpecialType::DontThrow' throws an exception despite being marked as `noexcept` (Line 52).
    - This issue mirrors the Visual Studio warning and carries the same risks and mitigation strategies.
* Array Index Out of Bounds :
  + Warning : Possible out of bounds access on `buf[10]` (Line 66).
    - Risk : Accessing an array out of bounds can lead to memory corruption, program crashes, or security vulnerabilities.
    - Mitigation : As with the Visual Studio warning, ensure that array indexing stays within bounds, possibly by using safer data structures or adding bounds checks.
* Assertions and Side Effects :
  + Warning : Variables modified inside assert statements (Lines 127, 129) may cause issues when assertions are removed in release builds.
    - Risk : Code inside assertions is not executed in release builds, leading to potential logic errors if the code performs necessary actions.
    - Mitigation : Avoid modifying variables inside assertions. Refactor such code to ensure that it executes correctly in both debug and release builds.
* Uninitialized and Unused Variables :
  + Warning : Uninitialized member variable 'A::x' (Line 42).
    - Again this mirrors the Visual Studio warning and carries the same risks and mitigation strategies.
  + Style : Unread variables and unused struct members (e.g., `buf[count]` at Line 66).
    - Risk : Unused variables increase code complexity and may indicate underlying logic issues.
    - Mitigation : Remove or repurpose unused variables to maintain code clarity and correctness.
* Shadowing Variables :
  + Style : Variables `x`, `y`, and `z` shadow outer variables (Lines 133 135).
    - Risk : Variable shadowing can lead to subtle bugs, as inner variables may unintentionally override outer variables, causing unexpected behavior.
    - Mitigation : Avoid shadowing variables by using distinct names or adjusting variable scope.

* Function Optimization :
  + Style : Cppcheck suggested that certain functions (e.g., `Token::next`, `MySpecialType::DontThrow`) could be made static for performance gains.
    - Risk : While not a critical issue, making functions static can improve performance by avoiding unnecessary instance references.
    - Mitigation : Apply static declarations where appropriate to optimize performance.

Visual Studio and Cppcheck both offer valuable insights but excel in different areas. Visual Studio is highly effective at identifying performance issues, buffer overflows, and type safety violations. However, it misses some low level memory issues and offers fewer recommendations for style improvements. On the other hand, Cppcheck is better suited for detecting memory related problems and provides more extensive warnings about style issues such as variable shadowing and unused variables. For instance, Cppcheck identified the dangerous assignment of a local variable’s address (Line 59), a critical memory issue that Visual Studio did not catch. Cppcheck also flagged the modification of variables inside assert statements , highlighting a potential issue in release builds that Visual Studio overlooked. On the other hand, Visual Studio effectively identified a buffer overrun issue in `buf` (Line 66), which Cppcheck also detected but with additional context about possible array out of bounds access. Both tools identified the issue with 'MySpecialType::DontThrow' throwing exceptions despite being marked noexcept , underscoring the importance of proper exception handling in critical functions.

The combination of Visual Studio and Cppcheck provides comprehensive static code analysis. Visual Studio is adept at catching performance and high level type safety issues, while Cppcheck excels in detecting memory errors and offering style recommendations. The identified issues—ranging from buffer overflows and uninitialized variables to memory corruption and variable shadowing—pose varying levels of risk, all of which can be mitigated with proper refactoring, memory management, and adherence to best coding practices. Together, these tools enable more robust, secure, and maintainable code.