**Static Code Analysis**

Justin Starr

Department of STEM

CS 405 – Secure Coding

Professor Mike Alesso

August 4, 2024

**Static Code Analysis**

**Screenshots**

VisualStudio

**A screenshot of a computer

Description automatically generated**

Cppcheck

A screenshot of a computer

Description automatically generated

**Process Summary**

For this assignment, I began by creating a new VisualStudio Project and copying the code from the QuestionableCode.cpp file to my project. I compiled it and was then able to see a list of the warnings and messages it produced. I then configured Cppcheck appropriately and then ran the analysis. From here, I was able to determine which errors were similar to each other, meaning the wording was slightly different, but they were the same errors, and which errors (including warnings and messages) were not the same. Once I was able to identify the differences, I was able to complete the following summary. It is important to note that the static analyzer is able to help prevent bugs and undefined behavior that VisualStudio alone does not. Static code analyzers are also great for analyzing the low-level details of an application, the details that are often over looked by developers (Meeting Cpp, 2020). They also provide performance suggestions that are not given in VisualStudio, which can help create better code.

**Issue Identified by VisualStudio**

There is one message that is displayed in VisualStudio that is not seen in the results from Cppcheck. On line 65, I get a message saying: Int-uninit – Local variable is not initialized. We can see in the code that when the variable ‘buf’ is declared, it is not initialized. To fix this issue, the C++ core guidelines recommend that all variables are initialized when they are declared.

**Issues Identified by Cppcheck**

The remaining four warnings that are displayed in VisualStudio are found in the results from Cppcheck.

In Cppcheck, errors are when code is executed there is either undefined behavior, such as a memory or resource leak. Warnings are when code is executed, there could be undefined behavior. Style messages are when there are stylistic issues, such as functions that are unused, code that is redundant, issues with operator precedence, and other possible mistakes. Performance messages are essentially performance suggestions at run time that are based on common knowledge, however, fixing them does not guarantee any measurable improvements will be achieved. Additionally, information messages occur where the Cppcheck configuration could be improved (Cppcheck, n.d.).

The errors, warnings, and messages from CppCheck that were not identified in Visual Studio are as follows:

5 – Severity: Information (No risk) – missingIncludeSystem – Lines 5 – 9, displays information that Cppcheck does not need standard library headers to get proper results. Essentially, it is telling us that the static analysis can be performed without including these header files. No mitigation strategy is necessary for informational messages displayed in the Cppcheck results.

1 - Severity: Warning (Risk) – assignmentInAssert – Line 128 - says the assert statement modifies the variable ‘z’. Because assert statements are typically removed from release builds, variables should never be modified inside an assert statement. Risk. The mitigation strategy for this warning is to first determine whether the variable modification is needed in the code, and if it is, the variable modification should be implemented in the portion of code that requires it and not inside the assert, especially because asserts are typically removed for release builds.

1 – Severity: Warning (Risk) – assertWithSideEffect – Line 130 - says that the assert statement calls a function that may have desired side effects when that function is called. Again, since assert statements are typically removed from release builds, the code inside the assert statement won’t be executed. Therefore, if the code is needed in a release build, it would cause a bug in the code. The mitigation strategy for this warning is the same as the previous warning. The variable assignment should occur in the code, where appropriate, and outside of the assert.

1 – Severity: Warning (Risk) – uselessAssignmentPtrArg – Line 110 – says that the assignment of function parameter has no effect outside the function. It suggests that it might have been forgotten to dereference it. The mitigation strategy for this warning is to make sure pointers are used to ensure changes that are made within the function affect the original variable. dereferencing ‘tok’ would ensure that any changes made within the foo() function will affect the original ‘tok’ variable.

1 – Severity: Error (Risk) – autoVariables – Line 60 - this error states that the address of a local auto variable is assigned to a function parameter. This is a dangerous assignment because local auto variables are reserved from the stack which is freed when the function ends, therefore making the pointer to the local variable invalid after the function ends. This relates to Common Weakness Enumeration (CWE-562). Potential mitigations include using static analysis tools to find instances in your code where the addressed of a stack variable may be returned or exposed, such as in this case.

1 – Severity: Style (Risk) – returnNonBoolInBooleanFunction – Line 99 – here, a non-boolean value is returned from a function whose return type is bool. This is a risk because it can lead to runtime errors, undefined behavior, and potential security risks. In this case, a mitigation strategy is to consider if the function should return a different type or modify the function to return the correct Boolean value.

2 – Severity: Performance (No risk) – functionStatic – Lines 51 and 104 – Says that both functions from each of these lines could be static and should not result in compiler errors. Changing these to static could bring a performance benefit since there is no ‘this’ instance passed to the function. In both instances, it says that it could be considered to move the functions to an unnamed space based on the design and task of the function, particularly when the function must not access members of class instances. The mitigation strategy for these performance issues is to evaluate whether or not these functions should be made static or if they should be moved to an unnamed space. Making them static could bring a performance benefit.

1 – Severity: Warning (Risk) – nullPointerRedundantCheck – Line 110 - This warning states that either the condition ‘tok’ is redundant or there is a possible null pointer dereference: tok. A mitigation strategy for this warning is to check the results of all functions that return a value, and verify that the value is non-null before acting upon it.

1 – Severity: Style (Risk) – variableScope – Line 65 – This style message says that the scope of the variable ‘buf’ can be reduced, however when fixing this error, one should be careful of inner loops that might rely on the scope that was initially set. A mitigation strategy for this message is to check and see if the declaration can be moved inside the if statement. If so, it should be

3 – Severity: Style (Risk) – shadowVariable – Lines 134 – 136 – these three messages state that all three variables ‘x’, ‘y’, and ‘z’ all shadows outer variable. This can lead to confusion and unexpected behavior, which can lead to bugs. A mitigation strategy for this message is to determine where the variables should be declared and limit their scope. In this function, the variable ‘z’ is the only variable that needs to be declared globally, outside the try/catch block because it is used in an assertion. However, we can continue to limit the scope of the variables ‘x’ and ‘y’ and maintain their declarations inside the try/catch block and remove their global declarations.

1 – Severity: Error (Risk) – invalidContainer – Line 88 – This error says that when using iterator (iter) to local container ‘items’ may be invalid. This can cause problems in code, if for example, elements within a container are moved from one position to another, and the initial iterator still points to a location that is no longer valid. The potential mitigation for this error is to utilize static analysis tools to check for unreleased resources.

1 – Severity: Style (Risk) – unusedStructMember – Line 42 - This message says that the class member ‘A::x’ is never used. This can lead to further bugs in the code and potentially introduce weaknesses. The mitigation strategy for this message is to remove unused variables from the code.

4 – Severity: Style (Risk) – unreadVariable – Lines 67, 110, 118, and 119 – These messages show that in each of the four lines a variable is assigned a value that is never used. In line 67, ‘buf[count]’ is the variable that is assigned a value that is never used. In line 110, ‘tok’ is the variable, line 118, ‘x’ is the variable, and in line 119, ‘y’ is the variable. This can also lead to further bugs in the code and potentially introduce weaknesses. The mitigation strategy for this message is to also remove unused variables from the code.

**Similar Errors & Warnings in VisualStudio and Cppcheck**

The remaining four warnings that are produced when compiling the code in VisualStudio are similar to four of the warnings and errors that are produced by Cppcheck.

The first warning in VisualStudio - C26495 – Variable ‘A::x’ is uninitialized, is the same as the warning produced in Cppcheck – Severity: Warning (Risk) – Line 43 uninitMembervarPrivate – member variable ‘A::x’ is not initialized in the copy constructor. Leaving variables of native types, pointers, or references uninitialized when the class is instantiated can lead to bugs or undefined behaviors. Mitigation strategy is to ensure all member variables are initialized.

The second warning in VisualStudio – C6386 – Buffer overrun while writing to ‘buf’ is similar to the error in the Cppcheck results – Severity: Warning (Risk) – arrayIndexOutOfBoundsCond - Line 67 -Either the condition ‘count == 1000’ is redundant or the array ‘buf[10]’ is accessed at index 1000, which is out of bounds. This can cause a buffer overrun, which can potentially expose sensitive data. To mitigate this, a check should be implemented to ensure the variable ‘count’ is not accessing an out of bounds index for the array ‘buf[10]’.

The third warning in VisualStudio – C4297 – ‘MySpecialType::DontThrow’: function assumed not to throw an exception but does, is similar to the error in the Cppcheck results – Severity: Error (Risk) – thrownInNoexceptFunction – Line 53 - An exception is thrown in function declared not to throw exceptions. To mitigate this, any exceptions that throw errors in an noexcept function should be removed, or the function should not be declared noexcept.

The fourth warning in VisualStudio – C4806 – ‘==’: unsafe operations: no value of type ‘bool’ promoted to type ‘int’ can equal the given constant, is similar to the Cppcheck result – Severity: warning – compareBoolExpressionWithInt – Line 130 – Comparison of a Boolean expression with an integer other than 0 or 1. This is happening because the function my\_function is a bool type function and is expected to return a Boolean value of either 0 or 1. However, the function returns a value of 3, which is of type int. To mitigate this, the function my\_function() should be changed to an int-type function to allow an integer value, such as 3, to be returned. That or the logic within my\_function() should be modified so that only a Boolean value is returned.

**References**

Cppcheck. (n.d.). *Manual*. <https://cppcheck.sourceforge.io/manual.html>

Meeting Cpp. (2020, February 14). *Things I llearned from the static analyzer – Bart Verhagen –*

*Meeting C++ 2019* [Video]. YouTube. [Things I learned from the static analyzer - Bart Verhagen - Meeting C++ 2019 (youtube.com)](https://www.youtube.com/watch?v=__Wz2DG1t24)