ISO/IEC JTC 1/SC 22/WG 23 N 0326

Markup of Proposed rewrite of WXQ and YZS

Date	25 March 2011
Contributed by	Secretary
Original file name	
Notes	Meeting #17 markup of N0325

From the Minutes of Meeting #17:

> We decide that there are two vulnerabilities, to be named "Unused Variable" and "Dead Store". Unused Variable is a vulnerability because it leaves storage to be used by an attacker. Dead Store is a problem because it indicates a design or coding error. (If the apparent errant behaviour was really intended then the variable should have been marked as Volatile.) The compiler may optimize it away and the intended communication between the processes may not occur. (Note that the new version of C++ may separate "Atomic" from "Volatile".) Moore will redraft [ACTION].

6.18 Dead Store [WXQ]

6.18.1 Description of application vulnerability

A variable's value is assigned but never subsequently used, either because the variable is not referenced again, or because a second value is assigned before the first is used. This may suggest that the design has been incompletely or inaccurately implemented, i.e. a value has been created and then 'forgotten about'.

~~In the programming languages C and C++ a *volatile* variable is always assumed to be "subsequently used", because storing to such variables may have side effects unknown to the implementation.~~

~~Dead stores by themselves are innocuous, but can combine with other vulnerabilities, such as index bounds errors or buffer overflows, to mask errors or provide hidden channels.~~

This vulnerability is ~~very similar~~related to Unused Variable [YZS]. ~~Indeed, a variable that is declared and initialized but never subsequently used may be regarded as either a dead store or an unused variable.~~

6.18.2 Cross reference

CWE:
563. Unused Variable
MISRA C++ 2008: 0-1-4 and 0-1-6
CERT C guidelines: MSC13-C
See also Unused Variable [YZS]

6.18.3 Mechanism of failure

A variable is assigned a value but this is never subsequently used. Such an assignment is then generally referred to as a dead store.

A dead store may be indicative of careless programming or of a design or coding error; as either the use of the value was forgotten (almost certainly an error) or the assignment was performed even though it was not needed (at best inefficient). Dead stores may also arise as the result of

mistyping the name of a variable, if the mistyped name matches the name of a variable in an enclosing scope.

There are legitimate uses for apparent dead stores. For example, the value of the variable might be intended to be read by another execution thread or an external device. In such cases, though, the variable should be marked as volatile. Common compiler optimization techniques will remove apparent dead stores if the variables are not marked as volatile, hence causing incorrect execution.

~~Dead stores may also arise as the result of mistyping the name of a variable, if the mistyped name matches the name of a variable in an enclosing scope.~~

A dead store ~~is~~ is also justifiable if, for example:

• ~~The variable is volatile and the assignment of a value triggers some external event.~~

• ~~The~~ The code has been automatically generated, where it is commonplace to find dead stores introduced to keep the generation process simple and uniform.

• The code is initializing a sparse data set, where all members are cleared, and then selected values assigned a value.

~~While a dead store is very unlikely of itself to be the cause of erroneous behaviour., their presence may also be an indication that compiler warnings are either suppressed or are being ignored by programmers.~~

6.18.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- ~~Dead stores are possible in any~~Any programming language that provides assignment. ~~(Pure functional languages do not have this issue.)~~

Formatted: Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

6.18.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

~~• Enable detection of dead stores in their compiler (if available). The default setting may be to suppress these warnings.~~

• Use static analysis to identify any dead stores in the program, and ensure that there is a justification for them.

• If variables are intended to be accessed by other execution threads or external devices, mark them as volatile.

• ~~Do not declare~~Avoid declaring variables of compatible types in nested scopes with similar names.

6.18.6 Implications for standardization

In future standardization activities, the following items should be considered:

• Languages should consider ~~requiring~~ providing ~~mandatory~~ optional ~~diagnostics~~ warnings for dead store.

6.19 Unused Variable [YZS]

6.19.1 Description of application vulnerability

~~A variable is declared but neither read nor written in the program, making it an unused variable~~An unused variable is one that is declared but neither read nor written in the program. This type of error suggests that the design has been incompletely or inaccurately implemented.

Unused variables by themselves are innocuous, ~~but can combine with other vulnerabilities such as index bounds errors or buffer overflows to mask errors or provide hidden channels~~but they may provide memory space that attackers could use in combination with other techniques.

This vulnerability is ~~very similar~~similar to Dead Store [WXQ] if the variable is initialized but never used. ~~Indeed, a variable that is declared and initialized but never subsequently used, may be regarded as either a dead store or an unused variable.~~

6.19.2 Cross reference

CWE:
563. Unused Variable
MISRA C++ 2008: 0-1-3
CERT C guidelines: MSC13-C
See also Dead Store [WXQ]

6.19.3 Mechanism of failure

A variable is declared, but never used. ~~It is likely that the variable is simply vestigial, but it is also possible that the unused variable points out a bug. This is likely to suggest that the design has been incompletely or inaccurately implemented~~The existence of an unused variable may indicate a design or coding error.

~~Whilst an unused variable is very unlikely of itself to be the cause of erroneous behaviour, as~~Because compilers routinely diagnose unused local variables, their presence ~~is often~~may be an indication that compiler warnings are either suppressed or are being ignored ~~by programmers~~. While unused variables are innocuous, they may provide available memory space to be used by attackers to exploit other vulnerabilities.

6.19.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

• ~~Unused variables (in the technical sense above) are possible only in languages~~Languages that provide variable declarations.

6.19.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

• Enable detection of unused variables in the compiler. ~~The default setting may be to suppress these warnings.~~

6.19.6 Implications for standardization

In future standardization activities, the following items should be considered:

• Languages should consider requiring mandatory diagnostics for unused variables.