ISO/IEC JTC 1/SC 22/WG 23 N 0272

Possible new vulnerability descriptions from splitting XYR into two descriptions

Date	2010-08-31
Contributed by	Clive Pygott
Original file name	XYR.doc
Notes	*Action item #14-09*

6.10 ~~Unused Variable~~Dead Store [XYR]

6.10.1 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., ~~making it a dead store. As a variant, a variable is declared but neither read nor written to in the program, making it an unused variable.~~ This ~~type of error~~ may suggest~~s~~ that the design has been incompletely or inaccurately implemented, i.e. a value has been created and then 'forgotten about'.

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

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

6.10.2 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 [???]

6.10.3 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.~~

A variable is assigned a value but this ~~value~~ is never subsequently used ~~thereafter~~. ~~The~~ Such an assignment is then generally referred to as a dead store. ~~Note that this may be acceptable if the variable is a volatile variable, for which the assignment of a value triggers some external event.~~

Comment [C H P1]: Where is this "generally referred to as a dead store"? This isn't a term in the BCS glossary

A dead store ~~is~~ 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 ~~(unless there is a justification for it)~~).

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 justifiable if, for example:

- the variable is volatile and the assignment of a value triggers some external event
- 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 initialising a sparse data set, where all members are cleared, then selected values assigned. a value.

An unused variable or aWhilst a dead store is very unlikely of itself to be the cause of a vulnerabilityerroneous behaviour. However,. since compilers diagnose unused variables routinely and dead stores occasionally, their presence is often may also be an indication that compiler warnings are either suppressed or are being ignored by programmers. This observation does not hold for automatically generated code, where it is commonplace to find unused variables and dead stores, introduced to keep the generation process simple and uniform.

6.10.4 Applicable language characteristics

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

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

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

6.10.5 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 and 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
Do not declare variables of compatible types in nested scopes with similar names

6.10.6 Implications for standardization

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

Languages should consider requiring mandatory diagnostics for unused variablesdead store.

6.10 Unused Variable [~~XYR~~???]

6.10.1 Description of application vulnerability

A ~~variable's value is assigned but never used, making it a dead store. As a variant, a~~ variable is declared but neither read nor written ~~to~~ in the program, making it an unused variable. This type of error suggests that the design has been incompletely or inaccurately implemented.

Unused variables by themselves are innocuous, but can ~~be~~ combine~~d~~ with other vulnerabilities such as index bounds errors ~~and~~ or buffer overflows ~~and may~~to mask errors or provide hidden channels.

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

6.10.2 Cross reference

CWE:
563. Unused Variable
MISRA C++ 2008: 0-1-4 and 0-1-6
CERT C guidelines: MSC13-C

See also Dead Store [XYR]

6.10.3 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.

~~A variable is assigned a value but this value is never used thereafter. The assignment is then generally referred to as a dead store. Note that this may be acceptable if the variable is a volatile variable, for which the assignment of a value triggers some external event.~~

~~A dead store is indicative of careless programming or of a design or coding error; either the use of the value was forgotten (almost certainly an error) or the assignment was performed even though it was not needed (unless there is a justification for it).~~

~~An~~Whilst an unused variable ~~or a dead store~~ is very unlikely of itself to be the cause of erroneous behaviour~~a vulnerability. However, since~~, as compilers routinely diagnose unused variables~~, routinely and dead stores occasionally,~~ their presence is often an indication that compiler warnings are either suppressed or are being ignored by programmers. ~~This observation does not hold for automatically generated code, where it is commonplace to find unused variables and dead stores, introduced to keep the generation process simple and uniform.~~

6.10.4 Applicable language characteristics

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

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

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

6.10.5 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 ~~and dead stores~~ in the compiler. The default setting may be to suppress these warnings.

6.10.6 Implications for standardization

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

Languages should consider requiring mandatory diagnostics for unused variables.