6.22 Initialization [LAV]

6.22.1 Description of application vulnerability

Reading a variable that has not been assigned a value appropriate to its type can cause unpredictable execution in the block that uses the value of that variable, and has the potential to export bad values to callers, or to cause out-of-bounds memory accesses.

Uninitialized variable usage is frequently not detected until after testing and often when the code in question is delivered and in use, because happenstance will provide variables with adequate values (such as default data settings or accidental left-over values) until some other change exposes the defect.

Variables that are declared during module construction (by a class constructor, instantiation, or elaboration) may have alternate paths that can read values before they are set. This can happen in straight sequential code but is more prevalent when concurrency or co-routines are present, with the same impacts described above.

Another vulnerability occurs when compound objects are initialized incompletely or incorrectly, as can happen when objects are incrementally built, fields are added under maintenance, or data structures are initialized as an aggregate. When possible and supported by the language, aggregate initialization is preferable to field-by-field initialization statements, and named association is preferable to positional, as it facilitates human review and is less susceptible to error injection under maintenance. Aggregate initialization can be problematic for multidimensional arrays if the braces and initializer lists are not ordered properly. A structure being initialized as an aggregate can result in elements being initialized to unintended values if the data is not listed carefully. For classes, the declaration and initialization may occur in separate modules. In such cases it must be possible to show that every field that needs an initial value receives that value, and to document ones that do not require initial values.

6.22.2 Cross reference

CWE:
457. Use of Uninitialized Variable
JSF AV Rules: 71, 142, 143, 144, 145 and 147
MISRA C 2012: 9.1, 9.2, and 9.3
MISRA C++ 2008: 8-5-1
CERT C guidelines: DCL14-C and EXP33-C
Ada Quality and Style Guide: 5.9.6

6.22.3 Mechanism of failure

Uninitialized objects may have invalid values, valid but wrong values, or valid and dangerous values. Wrong values could cause unbounded branches in conditionals or unbounded loop executions, or could simply cause wrong calculations and results.

Stephen Michell 2017-6-19 4:13 PM
Formatted: Right

Wagoner, Larry D. 2017-5-24 3:11 PM
Deleted: of variables

Wagoner, Larry D. 2017-5-24 3:25 PM
Deleted: or

Wagoner, Larry D. 2017-5-24 3:49 PM
Deleted: .

Wagoner, Larry D. 2017-5-24 3:48 PM
Deleted: whole-structure

Wagoner, Larry D. 2017-5-24 3:59 PM
Formatted: Font:11 pt, Not Bold

There is a special case of pointers or access types. When such a type contains null values, a bound violation and hardware exception can result. When such a type contains plausible but meaningless values, random data reads and writes can collect erroneous data or can destroy data that is in use by another part of the program; when such a type is an access to a subprogram with a plausible (but wrong) value, then either a bad instruction trap may occur or a transfer to an unknown code fragment can occur. All of these scenarios can result in undefined behaviour.

Uninitialized or incorrectly initialized variables are difficult to identify and use for attackers, but can be arbitrarily dangerous in safety situations.

The general problem of showing that all program objects are initialized is intractable.

Wagoner, Larry D. 2017-5-24 4:01 PM
Deleted: ;

6.22.4 Applicable language characteristics

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

- Languages that permit variables to be read before they are assigned.

6.22.5 Avoiding the vulnerability or mitigating its effects

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

- Carefully structure programs to show that all variables are set before first read on every path throughout each subprogram.
- When an object is visible from multiple modules, identify a module that must set the value before reads can occur from any other module that can access the object, and ensure that this module is executed first.
- When concurrency, interrupts and co-routines are present, identify where early initialization occurs and show that the correct order is set via program structure, not by timing, OS precedence, or chance.
- Initialize each object at elaboration time, or immediately after subprogram execution commences and before any branches.
- If the subprogram must commence with conditional statements, show that every variable declared and not initialized earlier is initialized on each branch.
- Ensure that the initial object value is a sensible value for the logic of the program. The so-called "junk initialization" (such as, for example, setting every variable to zero) prevents the use of tools to detect otherwise uninitialized variables.
- Define or reserve fields or portions of the object to only be set when fully initialized. Consider, however, that this approach has the effect of setting the variable to possibly mistaken values while defeating the use of static analysis to find the uninitialized variables.
- Use static analysis tools to show that all objects are set before use. As the general problem is intractable, keep initialization algorithms simple so that they can be analyzed.

- When declaring and initializing the object together, if the language does not require the compiler to statically verify that the declarative structure and the initialization structure match, use static analysis tools to help detect any mismatches.
- When setting compound objects, if the language provides a capability for aggregate initialization, use that in preference to a sequence of initializations as this facilitates coverage analysis; otherwise use tools that perform such coverage analysis and document the initialization. Do not perform partial initializations unless there is no choice, and document any deviations from full initialization.
- Verify aggregate initializations to ensure that initializations are performed as expected since complex data structures can assign values different than expected.
- Where default assignments of multiple components are performed, explicit declaration of the component names and/or ranges helps static analysis and identification of component changes during maintenance.
- Use named assignments in preference to positional assignment where the language has named assignments that can be used to build reviewable assignment structures that can be analyzed by the language processor for completeness. Use comments and secondary tools to help show correct assignment where the language only supports positional assignment notation.

Wagoner, Larry D. 2017-5-24 4:03 PM
Deleted: mechanisms to set all components together

Wagoner, Larry D. 2017-5-24 4:03 PM
Deleted: those

6.22.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered:

- Some languages have ways to determine if modules and regions are elaborated and initialized and to raise exceptions if this does not occur. Languages that do not, could consider adding such capabilities.
- Languages could consider setting aside fields in all objects to identify if initialization has occurred, especially for security and safety domains.
- Languages that do not support whole-object initialization, could consider adding this capability.