ISO/IEC JTC 1/SC 22/OWGV N 0143

New Vulnerability Descriptions Proposed by J3 (Fortran), contributed by Dan Nagle, 26 August 2008

(reformatted by JWM)

6.9+ Unused Subprogram Argument [YZS]

6.9+.0 Status and History

6.9+.1 Description of application vulnerability

A dummy argument to a subprogram is present on the argument list, but it is never used within the subprogram. As a variant, it may be assigned an initial value, but otherwise be unused. Depending upon the programming language and argument passing mechanism, a value may or may not have been changed in any caller of the subprogram. This type of error suggests that the design has been incompletely or inaccurately implemented.

6.9+.2 Cross Reference

6.9+.3 Mechanism of failure

A dummy argument is declared, but never used. When the subprogram signature is present during compilation of calling subprograms, there must be an associated actual argument. It is likely that the argument is simply vestigial, but it is also possible that the unused argument points to a bug. This is likely to suggest that the design has been incompletely or inaccurately implemented.

The dummy argument may be assigned an initial value by the subprogram. This value may or may not have an effect on the corresponding actual argument in the calling subprogram. If the called subprogram is called from many points of the program, it might not be repaired due to the difficulty of finding all lines where it has been called.

An unused dummy argument is indicative of sloppy programming or of a design change or of a coding bug: either the use of the value was forgotten (almost certainly a bug) or the assignment was done even though it was not needed (sloppiness).

An unused dummy argument is unlikely to be the cause of a vulnerability. However, since compilers diagnose unused dummy arguments routinely, their presence is often an indication that compiler warnings are either suppressed or are being ignored by programmers. Also, a new version of the called subprogram, perhaps innocently in a new version of a library, or

maliciously in a subverted shared library, may store a value in the calling subprogram causing erroneous execution.

6.9+.4 Applicable language characteristics

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

- Unused dummy arguments are possible in any language that provides declaration of dummy arguments
- A store to a dummy argument may affect the actual argument in any language that allows side-effects in called subprograms
- A store may be more likely to affect the actual argument in a language that does not require argument intents
- A store may be more likely to affect variables in the calling subprogram where separate compilation of subprograms is allowed

6.9+.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.
- Use argument intents to limit where the called subprogram can store to a variable in the calling subprogram
- Declare subprograms to be pure so no stores to dummy arguments are allowed

6.9+.6 Implications for standardization

- Languages should consider requiring a mandatory diagnostics for unused dummy arguments.
- Languages should provide for argument intents.
- Languages should provide for pure subprograms.

6.9+.7 Bibliography [None]

6.11+ Type Choice [JIO]

6.11+.0 Status and History

6.11.1 Description of application vulnerability

Types are often subdivided into subtypes that differ in terms of the capacity for holding values. Using a lesser capable subtype where a more capable subtype is needed can result in overflow or other loss of information resulting in unpredictable values.

6.11+.2 Cross reference

6.11+.3 Mechanism of failure

Types often have subtypes, differing only in capability. The operations, (some) literal values, and (most) valid contexts are the same. Many programmers think of types in terms of the storage used, not the mathematical properties of variables of the type. Selecting the correct one for a programming variable requires analysis of the role to be supported by variables of that type. This must be done by reference to the values to be stored, not to the storage used to do so.

For example, in C, the types long int, int and short int might (or might not) be three different types of integer. A programmer might have a belief that array indices may always be ints. But with larger memories, a long int may be required. Specific analysis of the problem at hand is the only way to know whether int is serviceable, or long int is necessary.

The fact of larger memories implies that more operands may be stored, which implies that more operations may be undertaken for a given calculation. The magnitude of the accumulated round-off errors may then imply that a more precise floating point number is required where previously a less precise floating point number sufficed. Since many floating point calculations scale as some power of the amount of input data, an explicit calculation must be done to estimate the reliability of the use of any particular precision.

For example, in Fortran, a default real may suffice for inverting a small (say, 3 x 3) matrix, but a double precision real may be necessary for a large (say, 1000 x 1000) system.

6.11+.4 Applicable language characteristics

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

- Languages supporting more than one subtype of the same type, with different capabilities.

6.11+.5 Avoiding the vulnerability or mitigating its effects

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

- Analyze the problem to be solved to learn the magnitudes and/or the precisions of the quantities needed as auxiliary variables, partial results and final results.

6.11+.6 Implications for standardization

- Provide a mechanism for selecting types with sufficient capability for the problem at hand.
- Provide a way to learn the limits of types actually selected.

6.11+.7 Bibliography

6.39+ Sentinel Values [DTK]

6.39+.0 Status and history

6.39+.1 Description of application vulnerability

When a queue of work items is maintained, each work item may be marked with a work category. When information beyond the work category is attempted to be encoded in the work category, to indicate, say, end of work items, a sentinel value may be used. This confuses two different categories of information. When the software is generalized, or used in a new application, the previously distinct sentinel value may coincide with a newly valid category, thereby causing unpredictable behavior.

6.39+.2 Cross reference

6.39+.3 Mechanism of failure

A "magic number" or a "special value" of a computational quantity might be used to signal some action on the part of the program other than processing data. If the use of the software is later generalized, the once-special value can become indistinguishable from valid data. When interpreted as the signal, the program can behave unexpectedly by a user who expects the valid data to be processed.

An example is provided: Suppose a program analyzes radar data, taking data every degree of azimuth from 0 to 359. A packet of data is sent to an analysis module for processing, updating displays, recording, and so on. Since all degree values are non-negative, a degree "value" of -1 is used to signal to stop processing, compute summary data, close files, and so on. When the software driving this software is reconfigured to record data from -180 degrees to 179 degrees, the analysis will halt when -1 degrees is reached. If the magic value is changed to, say, -999, the software is still at risk of failing when future enhancements (say, counting accumulated degrees on complete revolutions) bring -999 into the range of valid data.

Sentinel values should not be used. Rather, the software should be designed to use a separate predicate function to return the needed extra signals. In the above example, a shut_down() function, returning a boolean value, removes the possibility that subsequent generalization of the software will fail mysteriously.

The fact that the software is trusted may lead to less emphasis on it during testing of the more general ensemble of software.

This is not to be taken to indicate that language-standard-defined conventions, or other wide-spread conventions or usages, must be avoided, but rather that application-specific, and therefore likely surprising, sentinel values are to be avoided.

6.39.4 Applicable language characteristics

Most programming languages allow this type of fault.

6.39.5 Avoiding the vulnerability or mitigating its effects

Design software so that distinct logical predicates are signaled by distinct mechanisms. Embedding signals in a data channel should be shunned. Do not allow the possibility that metadata is confused with data.

Use an enumeration type to convey category information. Do not rely upon large ranges of integers, with outliers having special meanings.

6.39.6 Implications for standardization

Most programming languages have a boolean or logical type to convey the needed predicate.

6.39.7 Bibliography

6.41+ Extra Intrinsics [PUS]

6.41+.0 Status and history

6.41+.1 Description of application vulnerability

Most languages define intrinsic procedures, which are easily available, or always "simply available", to any translation unit. If a translator extends the set of intrinsics beyond those defined by the standard, and the standard specifies that intrinsics are selected before procedures of the same signature defined by the application, a different procedure may be unexpectedly used when switching between translators.

6.41+.2 Cross reference

6.41+.3 Mechanism of failure

Most standard programming languages define a set of intrinsic procedures which may be used in any application. Some language standards allow a translator to extend this set of intrinsic procedures. Some language standards specify that intrinsic procedures are selected ahead of an application procedure of the same signature. This may cause a different procedure to be used when switching between translators.

For example, most languages provide a routine to calculate the square root of a number, usually named sqrt(). If a translator also provided, as an extension, a cube root routine, say named cbrt(), that extension may override an application defined procedure of the same signature. If the two different cbrt() routines chose different branch cuts when applied to complex arguments, the application could unpredictably go wrong.

If the language standard specifies that application defined procedures are selected ahead of intrinsic procedures of the same signature, the use of the wrong procedure may mask a linking error.

6.41+.4 Applicable language characteristics

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

Any language where translators may extend the set of intrinsic procedures and where intrinsic procedures are selected ahead of application defined (or external library defined) procedures of the same signature.

6.41+.5 Avoiding the vulnerability or mitigating its effects

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

- Use whatever language features are available to mark a procedure as language defined or application defined.
- Be aware of the documentation for every translator in use and avoid using procedure signatures matching those defined by the translator as extending the standard set.

6.41+.6 Implications for standardization

Language standards should:

- clearly state whether translators may extend the set of intrinsic procedures or not
- clearly state what the precedence is for resolving collisions
- clearly provide ways to mark a procedure signature as being the intrinsic or an application provided procedure
- provide that a warning is issued when an application procedure matches the signature of an intrinsic procedure

6.41+.7 Bibliography

6.41+ Library Signature [NSQ]

6.41+.0 Status and history

6.41+.1 Description of application vulnerability

Some older libraries were coded before the value of subprogram signatures was recognized, and added to language standards. If the library is large, the effort of adding those signatures by hand may be tedious and error-prone.

6.41+.2 Cross reference

6.41+.3 Mechanism of failure

When an older software library lacks the language specified signatures, due to its being prepared prior to the requirement that signatures be used, the signature must be created. If this is done manually, it may be tedious and error-prone.

Automated methods may exist. Or, translators may have options to create the signatures as they compile the older library. However, neither of these remedies might be required by the language standard.

6.41+.4 Applicable language characteristics

Languages where older versions of the language did not specify that subprogram signatures be supplied for all subprogram references.

6.41+.5 Avoiding the vulnerability or mitigating its effects

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

- Use translator options to create the needed signatures when compiling the library
- Use tools to create the signatures
- Avoid using translator options to reference library procedures without proper signatures
- Try to find a later version of the library that has the signatures

6.41+.6 Implications for standardization

Language standards should:

- Require translators to create signatures when needed when translating older libraries that lack them
- Provide correct linkage even in the absence of correctly specified procedure signatures. (Note that this may be very difficult where the original source code is unavailable.)

6.41+.7 Bibliography