ISO/IEC JTC1/SC22/WG5 N1929

Fortran Annex to TR 24772, Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

WG5 Vulnerabilities Subgroup

Annex Fortran
(Informative)
Vulnerability descriptions for language Fortran

1 Programming Language Fortran

Fortran.1 Identification of Standards

The International Standard Programming Language Fortran is defined by ISO/IEC JTC1 1539-1 (2010).

Fortran.2 General Terminology and Concepts

Fortran.2.1 About Fortran

The Fortran standard is written in terms of a *processor* which includes the language translator (that is, the compiler or interpreter, and supporting libraries), the operating system (affecting, for example, how files are stored or which files are available to a program), and the hardware (affecting, for example, the machine representation of numbers or the availability of a clock).

The Fortran standard specifies how the contents of files are interpreted. The standard does not place requirements on the size or complexity of a program that may cause a processor to fail.

A program conforms to the Fortran standard if it uses only forms specified by the standard, and does so with the interpretation given by the standard. Where the standard fails to give an interpretation to a form used by a program, the program is not standard-conforming. A procedure is standard-conforming if it may be included in an otherwise standard-conforming program in a way that is standard conforming.

The Fortran standard allows a processor to support features not defined by the standard, provided such features do not contradict the standard. Use of such features, called *extensions*, should be avoided. Processors must be able to detect and report the use of extensions.

Annex B.1 of the Fortran standard lists six features of Fortran 90 that have been deleted because they were redundant and considered largely unused. Although no longer part of the standard, they are supported by many compilers to allow old programs to continue to run. Annex B.2 lists ten features of Fortran 90 that are regarded as obsolescent because they are redundant - better methods are available in the current standard. The obsolescent features are described in the standard using a small font. The use of any deleted or obsolescent feature should be avoided. It should be replaced by a modern counterpart for greater clarity and reliability (by automated means if possible). Processors must be able to detect and report the use of these features.

The Fortran standard defines a set of intrinsic procedures, and allows a processor to extend this set with further procedures. A program that uses an intrinsic procedure not defined by the standard is not standard-conforming. Use of intrinsic procedures not defined by the standard should be avoided. Processors must be able to detect and report the use of intrinsics not defined by the standard.

The Fortran standard does not completely specify the effects of programs in some situations, but rather allows the processor to employ any of several alternatives. These alternatives are called *processor dependencies* and are summarized in Annex A of the standard. The programmer should not rely for program correctness on a particular alternative being chosen by a processor. In general, the representation of quantities, the results of operations, and the results of the calculations performed by intrinsic procedures are all processor-dependent approximations of their respective exact mathematical equivalent.

Although strenuous efforts have been made, and are ongoing, to ensure that the Fortran standard provides an interpretation for all Fortran programs, circumstances may arise where the standard fails to do so. If the standard fails to provide an interpretation for a program, the program is not standard-conforming.

Generally, processors are required to detect deviation from the standard during translation only, and not during execution of a program. It is the responsibility of the programmer to adhere to the Fortran standard. Many processors offer debugging aids to assist with this task. For example, most processors support options to report when, during execution, an array index is found to be out-of-bounds in an array reference.

Generally, the Fortran standard is written as specifying what a correct program produces as output, and not how such output is actually produced. That is, the standard specifies that a program executes *as if* certain actions occur in a certain order, but not that such actions actually occur. A means other than Fortran (for example, a debugger) might be able to detect such particulars, but not a standard-specified means (for example, a print statement).

The values of data objects that are guaranteed to be represented correctly are specified in terms of a bit model, an integer model, and a floating point model. Inquiry intrinsic procedures return values that describe the model rather than any particular hardware. The Fortran standard places minimal constraints on the representation of characters.

Interoperability of Fortran program units with program units written in other languages is defined in terms of a *companion processor*. A Fortran processor is its own companion processor, and may have other companion processors as well. The interoperation of Fortran program units is defined as if the companion processor is defined by the C programming language.

Fortran is an inherently parallel programming language, with program execution consisting of one or more replications, called *images*, of the program. The standard makes no requirements of how many images exist for any program, nor of the mechanism of inter-image communication. Inquiry intrinsic procedures are defined to allow a program to detect the number of images in use, and which replication a particular image represents. Within an image, many statements involving arrays are specifically designed to allow efficient vector instructions. Several constructs for iteration are specifically designed to allow parallel execution.

Fortran is the oldest international standard programming language with

the first Fortran compilers appearing over fifty years ago. During half a century of computing, computing technology has changed immensely and Fortran has evolved via several revisions of the standard. Also, during half a century of computing and in response to customer demand, some popular compilers supported extensions. There remains a substantial body of Fortran code that is written to previous versions of the standard or with extensions to previous versions, and before modern techniques of software development came into widespread use. The process of revising the standard has been done carefully with a goal of protecting applications programmers' investments in older codes. Very few features were deleted from older revisions of the standard; those that were deleted were little-used, or redundant with a superior alternative, or error-prone with a safer alternative. Modern compilers generally continue to support deleted features from older revisions of the Fortran standard, and even some extensions from older compilers, and do so with the intention of reproducing the original semantics. Also, there exist automatic means of replacing at least some archaic features with modern alternatives. Even with automatic assistance, there may be reluctance to change this existing software due to its having proven itself through usage on a wider variety of hardware than is in general use at present, or due to issues of regulation or certification. The decision to modernize trusted software is made cognizant of many factors, including the availability of resources to do so and the perceived benefits. This document does not attempt to specify one-size-fits-all criteria for modernizing trusted old code.

Fortran.2.2 Fortran Definitions

The following definitions are taken from the Fortran standard.

argument association association between an effective argument and a dummy argument

assumed shape array a dummy argument array whose shape is assumed from the corresponding actual argument

assumed size array a dummy argument array whose size is assumed from the corresponding actual argument

deleted feature a feature that existed in Fortran 77 but has been removed from later versions of the standard

explicit interface an interface of a procedure that includes all the characteristics of the procedure and names for its dummy arguments

image one of a mutually cooperating set of instances of a Fortran program; each has its own execution state and set of data objects

implicit typing an archaic rule that declares a variable upon use according to the first letter of its name

kind type parameter a value that determines one of a set of processor-dependent set of representations

module a separate scope that contains definitions that are to be accessed from other scopes

obsolescent feature a feature that is not recommended because better methods exist in the current standard

processor combination of computing system and mechanism by which programs are transformed for use on that computing system

processor dependent not completely specified in the Fortran standard, having methods and semantics determined by the processor

pure procedure a procedure subject to constraints such that its execution has no side effects

type named category of data characterized by a set of values, a syntax for denoting these values, and a set of operations that interpret and manipulate the values

Fortran.3 Type System [IHN]

Fortran.3.1 Applicability to Fortran

The Fortran type system is a strong type system consisting of the type, the kind, and in most circumstances, the rank, of a data object. The kind of an entity is a parameterization of the type. Objects of the same type that differ in the value of their kind parameter(s) differ in representation, and therefore in the limits of the values they may represent. A processor must support two kinds of type real and must support a complex kind for every real kind that it supports. A processor must support at least one integer kind with a range of 10^{18} or greater.

The compatible types in Fortran are the numeric types: integer, real, and complex. No coercion exists between type logical and any other type, nor between type character and any other type. Among the numeric types, coercion may result in a loss of information or a wrong result. For example, if a double-precision real is assigned to a single-precision real, roundoff is likely; and if integer operation results in a value outside the supported range, the

result is wrong. Likewise, if an integer variable with a large value is assigned to an integer variable whose range do not include the value, the result is wrong.

An example of coercion in Fortran is (assuming `rkp` names a suitable real kind parameter):

```
real( kind= rkp) ::  a
integer ::  i
a = a + i
```

which is automatically treated as if it were:

```
a = a + real( i, kind= rkp)
```

Objects of derived types are considered to have the same kind when their type definitions are the same instance of text (which may be distributed among program units by module use). (Sequence types and bind(c) types represent a narrow exception to this rule but they are not generally used because sequence types cannot be extended and cannot interoperate with types defined by a companion processor, and bind(c) types are, in general, only used to interoperate with types defined by a companion processor.)

Derived types may also have kind parameters and these kind parameters may apply to the derived type's components. Default assignment of variables of the same derived type is component-wise. Default assignment may be overridden by an explicitly coded assignment procedure. Type changing assignments and conversion procedures must be explicitly coded by the programmer. Operators applied to derived types must be explicitly coded by the programmer. These explicitly coded procedures can contain whatever checks are needed.

In addition to the losses mentioned in Clause 6, assignment of a complex entity to a real entity results in the loss of the imaginary part.

Assigning from an object of extended type to one of base type may also incur loss of data.

Intrinsic procedures may be used in declarations to supply desired kind values. Kind values may also be obtained as named constants from the intrinsic module `iso_fortran_env`.

Fortran.3.2 Guidance to Fortran Users

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

- Use kind values based on the needed range for integer types via the `selected_int_kind` intrinsic procedure, and based on the range and precision needed for real and complex types via the `selected_real_kind` intrinsic procedure.
- Use explicit conversion intrinsics for conversions, even when the conversion is within one type and is only a change of kind. Doing so alerts the maintenance programmer to the fact of the conversion, and that it is intentional.
- Use inquiry intrinsic procedures to learn the limits of a variable's representation and thereby take care to avoid exceeding those limits.
- Use derived types to prevent unwanted conversions.
- Use compiler options when available to detect during execution when a significant loss of information occurs.
- Use compiler options when available to detect during execution when an integer value overflows.

Fortran.4 Bit Representations [STR]

Fortran.4.1 Applicability to Fortran

Fortran defines bit positions by a *bit model* described in 13.3 of the standard. Care must be taken to understand the mapping between an external definition of the bits (for example, a control register) and the bit model. The programmer can rely on the bit model regardless of endian, or other hardware peculiarities.

Fortran allows constants to be defined by binary, octal, or hexadecimal digits, collectively called *BOZ constants*. These values may be assigned to named constants thereby providing a name for a mask.

Fortran provides access to individual bits within a storage unit by bit manipulating intrinsic procedures. Of particular use, double-word shift procedures are provided to extract bit fields crossing storage unit boundaries.

The bit model does not provide an interpretation for negative integer values. There are distinct shift intrinsic procedures to include, or not include, the sign bit.

Fortran.4.2 Guidance to Fortran Users

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

- Use the intrinsic procedure `bit_size` to determine the size of the bit model supported by the kind of integer in use.
- Be aware that the Fortran standard uses the term "left-most" to refer to the highest-order bit, and the term "left" to mean towards (as in `shiftl`), or from (as in `maskl`), the highest-order bit.
- Be aware that the Fortran standard uses the term "right-most" to refer to the lowest-order bit, and the term "right" to mean towards (as in `shiftr`), or from (as in `maskr`), the lowest-order bit.
- Avoid bit constants made by adding integer powers of two in favor of those created by the bit intrinsic procedures or encoded by BOZ constants.
- Use bit intrinsic procedures to operate on individual bits and bit fields, especially those that occupy more than one storage unit. Choose shift intrinsic procedures cognizant of the need to affect the sign bit, or not.
- Create objects of derived type to hide use of bit intrinsic procedures within defined operators and to separate those objects subject to arithmetic operations from those objects subject to bit operations.

Fortran.5 Floating-point Arithmetic [PLF]

Fortran.5.1 Applicability to Fortran

Fortran supports floating-point data. Furthermore, most processors support the IEEE standard and facilities are provided for the programmer to detect the extent of conformance.

Different rounding modes may be in effect during translation and execution, and during execution of arithmetic operations and formatted input/output conversions.

Fortran provides intrinsic procedures to give values describing the limits of any representation method in use, to provide access to the components of a floating point quantity, and to set the components.

Fortran.5.2 Guidance to Fortran Users

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

- Use procedures from a trusted library to perform calculations where floating-point accuracy is needed. Understand the use of the library procedures and test the diagnostic status values returned to ensure the calculation proceeded as expected.
- Do not create a logical value from a test for equality or inequality between two floating-point expressions. Use compiler options where available to detect such usage.
- Do not use floating-point variables as loop indices; use integer variables instead. A floating-point value may be computed from the integer loop variable as needed.
- Use intrinsic inquiry procedures to determine the limits of the representation in use when needed.
- Avoid getting or setting the parts of a floating point quantity. Use intrinsic procedures to provide the functionality when needed.
- Use the intrinsic module procedures to determine the limits of the processor's conformance to IEEE 754, and to determine the limits of the representation in use, when the IEEE intrinsic modules and the IEEE real kinds are in use.
- Use the intrinsic module procedures to detect and control the available rounding modes and exception flags, when the IEEE intrinsic modules are in use.

Fortran.6 Enumerator Issues [CCB]

Fortran.6.1 Applicability to Fortran

Fortran provides enumeration values for interoperation with C programs that use C enums. Their use is expected most often to occur when a C enum appears in the function prototype whose interoperation requires a Fortran interface.

The Fortran enumeration values are integer constants of the correct kind to interoperate with the corresponding C enum. The Fortran variables to be assigned the enumeration values are of type integer and the correct kind to interoperate with C variables of C type enum.

Fortran.6.2 Guidance to Fortran Users

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

- Use enumeration values in Fortran only when interoperating with C procedures that have enumerations as formal parameters and/or return enumeration values as function results.
- Ensure the interoperability of the C and Fortran definitions of every enum type used.
- Ensure that the correct companion processor has been identified, including any compiler options that affect enum definitions.
- Do not use variables assigned enumeration values in arithmetic operations, or to receive the results of arithmetic operations if subsequent use will be as an enumerator.

Fortran.7 Numeric Conversion Errors [FLC]

Fortran.7.1 Applicability to Fortran

Fortran processors must support two kinds of type real and must support a complex kind for every real kind supported. A processor must support at least one integer kind with a range of 10^{18} or greater and most processors support at least one integer kind with a smaller range.

Automatic conversion among these types is allowed.

Fortran.7.2 Guidance to Fortran Users

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

- Use the kind selection intrinsic procedures to select sizes of variables supporting the required operations and values.
- Ensure that values from untrusted sources are checked against the limits provided by the inquiry intrinsics for the type and kind of variable in use.
- Use the inquiry intrinsic procedures to supply the extreme values that a data object supports to ensure that program data is within limits.
- Use derived types and put checks in the applicable defined assignment procedures.
- Use static analysis to identify whether numeric conversion will lose information.
- Use compiler options when available to detect during execution when a significant loss of information occurs.
- Use compiler options when available to detect during execution when an integer value overflows.

Fortran.8 String Termination [CJM]

Fortran.8.1 Applicability to Fortran

This vulnerability is not applicable to Fortran since strings are not terminated by a special character.

Fortran.9 Buffer Boundary Violation [HCB]

Fortran.9.1 Applicability to Fortran

A Fortran program might be affected by this vulnerability in two situations. The first is that an array index could be outside its bounds, and the second is that a character substring index could be outside its length. The Fortran

standard requires that each array index be separately within its bounds, not simply that the resulting offset be within the array as a whole.

Fortran does not mandate array index checking to verify in-bounds array references, nor character substring index checking to verify in-bounds substring references.

The Fortran standard requires that array shapes conform for whole array assignments and operations. However, Fortran does not mandate that array shapes be checked with whole-array assignments and operations.

When a whole-array assignment occurs to define an allocatable array, the allocatable array is resized, if needed, to the correct size. When a whole character assignment occurs to define an allocatable character, the allocatable character is resized, if needed, to the correct size.

When a character assignment occurs to define a non-allocatable character entity and a size mismatch occurs, the assignment has a blank-fill (if the value is too short) or truncate (if the value is too long) semantic. If the character entity being defined is allocatable, it is resized if needed to be the correct size.

When an allocatable array is to be explicitly resized, the move_alloc intrinsic procedure may be used to efficiently obtain a larger amount of memory.

Most implementations include an optional facility for bounds checking. These may be incomplete for a dummy argument that is an explicit-shape or assumed-size array because of passing only the address of such an object, and/or the reliance on local declaration of the bounds. It is therefore preferable to use an assumed-shape array as a procedure argument. The performance of assumed-shape arrays was improved in Fortran 2008 with the introduction of the `contiguous` attribute.

Fortran provides a set of array bounds intrinsic inquiry procedures which may be used to obtain the bounds of arrays where such information is available. Fortran also provides character length intrinsic inquiry intrinsics so the length of character entities may be reliably found.

Fortran.9.2 Guidance to Fortran Users

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

- Enable bounds checking throughout development of a code. Disable bounds checking during production runs only for program units that are critical for performance.

- Obtain array bounds from array inquiry intrinsics whenever needed. Use explicit interfaces and assumed-shape arrays to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.
- Use allocatable arrays when arrays operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.
- Use allocatable character variables where assignment of strings of widely-varying sizes is expected so the left-hand side character variable is reallocated as needed.
- Use intrinsic assignment rather than explicit loops to assign data to statically-sized character variables so the truncate-or-blank-fill semantic protects against storage outside the assigned variable.

Fortran.10 Unchecked Array Indexing [XYZ]

Fortran.10.1 Applicability to Fortran

A Fortran program might be affected by this vulnerability in the situation an array index could be outside its bounds, The Fortran standard requires that each array index be separately within its bounds, not simply that the resulting offset be within the array as a whole.

Fortran does not mandate that array sizes be checked with whole-array assignment to a non-allocatable array.

When a whole-array assignment occurs to define an allocatable array, the allocatable array is resized, if needed, to the correct size. When a whole character assignment occurs to define an allocatable character, the allocatable character is resized, if needed.

When an allocatable array is to be explicitly resized, the move_alloc intrinsic procedure may be used to efficiently obtain a larger amount of memory.

Most processors include an optional facility for bounds checking. These may be incomplete for a dummy argument that is an explicit-shape or assumed-size array because of passing only the address of such an object, and/or the reliance on local declaration of the bounds. It is therefore preferable to use an assumed-shape array as a procedure argument. The performance of

assumed-shape arrays was improved in Fortran 2008 with the introduction of the `contiguous` attribute.

Fortran provides a set of array bounds intrinsic inquiry procedures which may be used to obtain the bounds of arrays where such information is available.

Fortran.10.2 Guidance to Fortran Users

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

- Ensure that consistent bounds information about each array is available throughout a program.
- Enable bounds checking throughout development of a code. Disable bounds checking during production runs only for program units that are critical for performance.
- Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
- Obtain array bounds from array inquiry intrinsics whenever needed. Use explicit interfaces and assumed-shape arrays to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.
- Use allocatable arrays when arrays operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.
- Declare the lower bound of each array extent to fit the problem, thus minimizing the use of index arithmetic.

Fortran.11 Unchecked Array Copying [XYW]

Fortran.11.1 Applicability to Fortran

Fortran provides array assignment, so this vulnerability applies.

An array assignment with shape disagreement is prohibited, but the standard does not require the processor to check for this.

When a whole-array assignment occurs to define an allocatable array, the allocatable array is resized, if needed, to the correct size. When a whole character assignment occurs to define an allocatable character, the allocatable character is resized, if needed.

Most implementations include an optional facility for bounds checking. These may be incomplete for a dummy argument that is an explicit-shape or assumed-size array because of passing only the address of such an object, and/or the reliance on local declaration of the bounds. It is therefore preferable to use an assumed-shape array as a procedure argument. The performance of assumed-shape arrays was improved in Fortran 2008 with the introduction of the `contiguous` attribute.

Fortran provides a set of array bounds intrinsic inquiry procedures which may be used to obtain the bounds of arrays where such information is available.

Fortran.11.2 Guidance to Fortran Users

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

- Ensure that consistent bounds information about each array is available throughout a program.
- Enable bounds checking throughout development of a code. Disable bounds checking during production runs only for program units that are critical for performance.
- Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
- Obtain array bounds from array inquiry intrinsics whenever needed. Use explicit interfaces and assumed-shape arrays to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.
- Use allocatable arrays when arrays operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.

Fortran.12 Pointer Casting and Pointer Type Changes [HFC]

Fortran.12.1 Applicability to Fortran

This vulnerability is not applicable to Fortran in most circumstances. There is no mechanism for associating a data pointer with a procedure pointer. A non-polymorphic pointer is declared with a type and may be associated only with an object of its type. A polymorphic pointer that is not unlimited polymorphic is declared with a type and may be associated only with an object of its type or an extension of its type. An unlimited polymorphic pointer can be used to reference its target only by using a type with which the type of its target is compatible in a `select type` construct. These restrictions are enforced during compilation.

When an unlimited polymorphic pointer has a target of a sequence type or an interoperable derived type, a type-breaking cast may occur.

A pointer appearing as an argument to the intrinsic module procedure `c_f_pointer` effectively has its type changed to the intrinsic type `c_ptr`. Further casts may be made if the pointer is processed by procedures written in a language other than Fortran.

Fortran.12.2 Guidance to Fortran Users

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

- Avoid C interoperability features in programs that do not interoperate with other languages.
- Avoid use of sequence types.

Fortran.13 Pointer Arithmetic [RVG]

Fortran.13.1 Applicability to Fortran

This vulnerability is not applicable to Fortran. There is no mechanism for pointer arithmetic in Fortran.

Fortran.14 Null Pointer Dereference [XYH]

Fortran.14.1 Applicability to Fortran

A Fortran pointer must not be referenced when its status is disassociated.

The Fortran intrinsic procedure `associated` determines whether a pointer that is not undefined has a valid target, or whether it is associated with a particular target.

Some processors include an optional facility for pointer checking.

Fortran.14.2 Guidance to Fortran Users

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

- Enable pointer checking during development of a code throughout. Disable pointer checking during production runs only for program units that are critical for performance.
- Use the `associated` intrinsic procedure before dereferencing the pointer if there is any possibility of it being disassociated.

Fortran.15 Dangling Reference to Heap [XYK]

Fortran.15.1 Applicability to Fortran

This vulnerability is applicable to Fortran because it has pointers, and separate allocate and deallocate statements for them.

Fortran.15.2 Guidance to Fortran Users

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

- Use allocatable objects in preference to pointer objects whenever the facilities of allocatable objects are sufficient.
- Use compiler options where available to detect dangling references.
- Enable pointer checking throughout development of a code. Disable pointer checking during production runs only for program units that are critical for performance.

- Do not pointer-assign a pointer to a target whenever the pointer has a longer lifetime than the target.
- Check for successful deallocation when deallocating a pointer by using the `stat=` specifier.

Fortran.16 Arithmetic Wrap-around Error [FIF]

Fortran.16.1 Applicability to Fortran

This vulnerability is applicable to Fortran for integer values. Some compilers have an option to detect this vulnerability at run time.

Fortran.16.2 Guidance to Fortran Users

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

- Use the intrinsic `selected_int_kind` to select an integer kind value that will be adequate for all anticipated needs, including long into the future.
- Use compiler options when available to detect during execution when an integer value overflows.

Fortran.17 Using Shift Operations for Multiplication and Division [PIK]

Fortran.17.1 Applicability to Fortran

Fortran provides bit manipulation through intrinsic procedures that operate on integer variables. Specifically, both shifts that replicate the sign bit and shifts that do not are provided as intrinsic procedures with integer operands.

Fortran.17.2 Guidance to Fortran Users

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

- Separate integer variables into those on which bit operations are performed and those on which integer arithmetic is performed.

- Do not use shift intrinsics where integer multiplication or division is intended.

Fortran.18 Sign Extension Error [XZI]

Fortran.18.1 Applicability to Fortran

This vulnerability is not applicable to Fortran. Fortran does not have unsigned data types. The processor is responsible for converting a value correctly when a smaller size integer's value is assigned to a larger sized integer variable.

Fortran.19 Choice of Clear Names [NAI]

Fortran.19.1 Applicability to Fortran

Fortran is a single-case language; upper case and lower case are treated identically by the standard in names.

Names may include underscores. The number of consecutive underscores is significant but may be difficult to see.

When implicit typing is in effect, a misspelling of a name results in a new variable. Implicit typing may be disabled by use of the `implicit none` statement.

Fortran has no reserved names. Language keywords are permitted as names.

Fortran.19.2 Guidance to Fortran Users

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

- Declare all variables and use `implicit none` to enforce this.
- Do not distinguish names by case only.
- Do not use consecutive underscores in a name.
- Do not use keywords as names when there is any possibility of confusion.

Fortran.20 Dead Store [WXQ]

Fortran.20.1 Applicability to Fortran

Fortran provides assignment so this is applicable.

Fortran.20.2 Guidance to Fortran Users

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

- Use a compiler, or other analysis tool, that provides a warning for this.
- Use the volatile attribute where a variable is assigned a value to communicate with a device or process unknown to the compiler.
- Do not use similar names in nested scopes.

Fortran.21 Unused Variable [YZS]

Fortran.21.1 Applicability to Fortran

Fortran has separate declaration and use of variables and does not require that all variables declared be used, so this vulnerability applies.

Fortran.21.2 Guidance to Fortran Users

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

- Use a compiler that can detect a variable that is declared but not used and enable the compiler's option to do so at all times.

Fortran.22 Identifier Name Reuse [YOW]

Fortran.22.1 Applicability to Fortran

Fortran has several situations where nested scopes occur. These include:

- Module procedures are nested within their module host.
- Internal procedures are nested within their (procedure) host.

- A `block` construct might contain a declarative section allowing nested scope to declare new names within the host scope.
- An implied `do` loop may contain a nested scope.

Nested procedures access their host's variables via *host association*.

The index variables of some constructs, such as or `do concurrent`, `forall`, or array constructor implied do loops, are local to the construct.

Fortran.22.2 Guidance to Fortran Users

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

- Do not reuse a name within a nested scope.
- Clearly comment the distinction between similarly-named variables, whenever they occur in nested scopes.

Fortran.23 Namespace Issues [BJL]

Fortran.23.1 Applicability to Fortran

Fortran does not have namespaces. However, when implicit typing is used within a scope, and a module is accessed via use association without an only list, a similar issue may arise.

Specifically, a variable appearing but not explicitly declared may have a name that is the same as a name that was added to the module after the module was first used. This can cause the declaration, meaning, and the scope of the affected variable to change.

Fortran.23.2 Guidance to Fortran Users

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

- Never use implicit typing. Always declare all variables.
- Use a global `private` statement in all modules to require explicit export of specific names.

- Use an `only` clause on every `use` statement.
- Use renaming when needed to avoid name collisions.

Fortran.24 Initialization of Variables [LAV]

Fortran.24.1 Applicability to Fortran

The value of a variable that has never had a value stored in it is undefined. It is the programmer's responsibility to guard against use of uninitialized variables.

Fortran.24.2 Guidance to Fortran Users

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

- Favour explicit initialization for objects of intrinsic type and default initialization for objects of derived type. When providing default initialization, provide default values for all components.
- Use type value constructors to provide values for all components.
- Use compiler options, where available, to find instances of use of uninitialized variables.
- Use other tools, for example, a debugger or flow analyzer, to detect instances of the use of uninitialized variables.

Fortran.25 Operator Precedence/Order of Evaluation [JCW]

Fortran.25.1 Applicability to Fortran

Fortran specifies an order of precedence for operators. The order for the intrinsic operators is well known except among the logical operators `.not.` `.and.` `.or.` `.eqv.` `.neqv.` . These operators have the same precedence order for defined operations. In addition, any monadic defined operator, the defined operator `//`, and any dyadic defined operator have a position in this order, but these positions are not well known.

Fortran.25.2 Guidance to Fortran Users

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

- Use parentheses and partial-result variables within expressions to avoid any reliance on a precedence that is not well known.

Fortran.26 Side-effects and Order of Evaluation [SAM]

Fortran.26.1 Applicability to Fortran

Fortran functions are permitted to have side effects. Within some expressions, the order of invocation of functions is not specified. The standard explicitly requires that evaluating any part of an expression does not change the value of any other part of the expression, but there is no requirement for this to be diagnosed by the compiler.

Further, the Fortran standard allows a processor to ignore any part of an expression that is not needed to compute the value of the expression. Processors vary as to how aggressively they take advantage of this permission.

Fortran.26.2 Guidance to Fortran Users

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

- Replace any function with a side effect by a subroutine so that its place in the sequence of computation is certain.
- Assign function values to temporary variables and use the temporary variables in the original expression.
- Declare a function as pure whenever possible.

Fortran.27 Likely Incorrect Expression [KOA]

Fortran.27.1 Applicability to Fortran

While Fortran is not as susceptible to this issue as some languages (largely because assignment = is not an operator), nevertheless, some situations exist where a single character, present or absent, could change the meaning of an

expression. For example, assignment could be confused with pointer assignment when the name on the left-hand side has the pointer attribute and the name on the right-hand side has the target attribute.

Some compilers allow a dyadic operator immediately preceding a unary operator, which should be avoided. However, this can be detected by using compiler options to detect violations of the standard.

Fortran is not susceptible to the "dangling else" version of this problem because each construct has a unique end-of-construct statement.

Fortran.27.2 Guidance to Fortran Users

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

- Use an automatic tool to simplify expressions.
- Check for assignment versus pointer assignment carefully when assigning to names having the pointer attribute.
- Use dummy argument intents to assist the compiler's ability to detect such occurrences.

Fortran.28 Dead and Deactivated Code [XYQ]

Fortran.28.1 Applicability to Fortran

There is no requirement in the Fortran standard for compilers to detect code that cannot be executed. It is entirely the task of the programmer to remove such code.

The developer should justify each case of statements not being executed.

If desirable to preserve older code for documentation (for example, of an older numerical method), the code should be commented out. Alternatively, a source code control package can be used to preserve the text of older versions of a program.

Fortran.28.2 Guidance to Fortran Users

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

- Use a compiler, or other tool, that can detect dead or deactivated code.
- Use a coverage tool to check that every statement is executed.
- Use an editor or other tool that can transform a block of code to comments to do so with dead or deactivated code.
- Use a version control tool to maintain older versions of code when needed to preserve development history of a computational algorithm.

Fortran.29 Switch Statements and Static Analysis [CLL]

Fortran.29.1 Applicability to Fortran

Fortran has a `select case` construct, but control never flows from one alternative to another.

Fortran.29.2 Guidance to Fortran Users

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

- Cover cases that are expected never to occur with a `case default` clause to ensure that unexpected cases are detected and processed, perhaps emitting an error message.

Fortran.30 Demarcation of Control Flow [EOJ]

Fortran.30.1 Applicability to Fortran

Modern Fortran supports block constructs for choice and iteration, which have separate end statements for `do`, `select`, and `if` constructs. Furthermore, these constructs may be named which reduces visual confusion when blocks are nested.

There are archaic forms of loops and choices that should be avoided.

Fortran.30.2 Guidance to Fortran Users

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

- Use the block form of the `do`-loop, together with `cycle` and `exit` statements, rather than the non-block `do`-loop.
- Use the `if` construct or `select case` construct whenever possible, rather than statements that rely on labels, that is, the arithmetic `if` and `go to` statements.
- Use names on block constructs to provide matching of initial statement and end statement for each construct.

Fortran.31 Loop Control Variables [TEX]

Fortran.31.1 Applicability to Fortran

A Fortran enumerated `do` loop has the trip increment and trip count established when the `do` statement is executed. These do not change during the execution of the loop.

The program is prohibited from changing the value of an iteration variable during execution of the loop. The compiler is usually able to detect violation of this rule, but there are situations where this is difficult or requires use of a processor option; for example, an iteration variable might be changed by a procedure that is referenced within the loop.

Fortran.31.2 Guidance to Fortran Users

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

- Ensure that the value of the iteration variable is not changed other than by the loop control mechanism during the execution of a `do` loop.

Fortran.32 Off-by-one Error [XZH]

Fortran.32.1 Applicability to Fortran

Fortran is not very liable to this vulnerability because it permits explicit declarations of upper and lower bounds of arrays, which allows bounds that are relevant to the application to be used. For example, latitude can be declared with bounds -90 to 90, while longitude can be declared with bounds -180 to 180. Thus, user-written arithmetic on subscripts can be minimized.

This vulnerability is applicable to a mixed-language program containing both Fortran and C, since arrays in C always have the lower bound 0, and it may reduce the overall amount of explicit subscript arithmetic to declare the Fortran arrays with lower bounds of zero when they would otherwise be given different lower bounds.

Fortran.32.2 Guidance to Fortran Users

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

- Declare array bounds to fit the natural bounds of the problem.
- Declare interoperable arrays with the lower bound 0 so that the subscript values correspond between languages, when doing so reduces the overall amount of explicit subscript arithmetic.

Fortran.33 Structured Programming [EWD]

Fortran.33.1 Applicability to Fortran

As the first language to be formally standardized, Fortran has older constructs that allow an unstructured programming style to be employed.

These features have been superseded by better methods. The Fortran standard continues to support these archaic forms to allow older programs to function. Some of them are obsolescent, which means that the compiler is required to be able to detect and report their usage.

Automatic tools are the preferred method of refactoring unstructured code. Only where automatic tools are unable to do so should refactoring be done manually.

Refactoring efforts should always be thoroughly checked by testing of the new code.

Fortran.33.2 Guidance to Fortran Users

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

- Use a tool to automatically refactor unstructured code.

- Replace unstructured code manually with modern structured alternatives only where automatic tools are unable to do so.
- Use the compiler or other tool to detect archaic usage.

Fortran.34 Passing Parameters and Return Values [CSJ]

Fortran.34.1 Applicability to Fortran

Fortran does not specify the argument passing mechanism, but rather specifies the rules of *argument association*. These rules are generally implemented by either pass-by-reference or by copy-in/copy-out.

More restrictive rules apply to coarrays and to arrays with the contiguous attribute. Rules for procedures declared to have a C binding follow the rules of C.

Module procedures and internal procedures have explicit interfaces. An external procedure may have an explicit interface when an interface block is in scope to declare one. An interface block may be provided automatically or manually. Explicit interfaces allow compilers to check the type, kind, and rank of arguments and return values of functions.

Fortran.34.2 Guidance to Fortran Users

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

- Specify explicit interfaces by placing procedures in modules where the procedure is to be used in more than one scope, or by using internal procedures where the procedure is to be used in one scope only.
- Specify argument intents to allow further checking of argument usage.
- Specify pure (or elemental) for procedures where possible for greater clarity of the programmer's intentions.
- Use a compiler or other tool to automatically create explicit interfaces for external procedures.

Fortran.35 Dangling References to Stack Frames [DCM]

Fortran.35.1 Applicability to Fortran

A Fortran pointer is vulnerable to this issue when a local target does not have the `save` attribute and the pointer has a lifetime longer than the target. However, the intended functionality is often available with allocatables, which do not suffer from this vulnerability. The Fortran standard explicitly states that the lifetime of an allocatable function result extends to its use in the expression that invoked the call.

Fortran.35.2 Guidance to Fortran Users

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

- Do not pointer-assign a pointer to a target whenever the pointer has a longer lifetime than the target.
- Use allocatable variables in preference to pointers whenever they provide sufficient functionality.

Fortran.36 Subprogram Signature Mismatch [OTR]

Fortran.36.1 Applicability to Fortran

The Fortran term denoting a procedure's signature is its interface.

The Fortran standard requires that interfaces match, but does not require that the compiler diagnoses mismatches. However, compilers do check this when the interface is explicit.

Explicit interfaces are provided automatically when procedures are placed in modules or are internal procedures within other procedures.

Fortran.36.2 Guidance to Fortran Users

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

- Use explicit interfaces, preferably by placing procedures inside a module or another procedure.

- Use a compiler that checks all interfaces, especially since this may be checked during compilation with no execution overhead.
- Use a compiler or other tool to create explicit interface blocks for external procedures.

Fortran.37 Recursion [GDL]

Fortran.37.1 Applicability to Fortran

Fortran supports recursion, so this vulnerability applies. Possibly recursive procedures must be marked with the `recursive` attribute, thereby leaving some documentation of the programmer's intentions.

Recursive calculations may appear attractive in some situations due to their close resemblance to the most compact mathematical formula of the quantity to be computed.

Fortran.37.2 Guidance to Fortran Users

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

- Prefer iteration to recursion when evaluating mathematical quantities, unless it can be proved that the depth of recursion can never be large.

Fortran.38 Ignored Error Status and Unhandled Exceptions [OYB]

Fortran.38.1 Applicability to Fortran

Many Fortran statements and some intrinsic procedures return a status value. In most circumstances, status error values returned from statements that are not received by the invoking program result in the error termination of the program. Some programmers, however, in order to "keep going" accept the status value but do not examine it. This results in a program crash without an explanation when subsequent steps in the program rely upon the previous statements having completed successfully.

Fortran consistently uses a scheme of status values where zero indicates success, a positive value indicates an error, and a negative value indicates some other information.

Other than via the IEEE modules, Fortran does not support exception handling.

Fortran.38.2 Guidance to Fortran Users

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

- Code a status value for all statements that support one, and examine the value prior to continuing execution. For faults that cause termination, provide a message to users of the program, perhaps with the help of the error message generated by the statement whose execution generated the error.
- Appropriately treat all status values that may be returned by library procedures.

Fortran.39 Termination Strategy [REU]

Fortran.39.1 Applicability to Fortran

Fortran distinguishes between normal termination (`stop`) and error termination (`error stop`). For a normal termination there are three stages, initiation, synchronization, and completion. This allows images that are still executing to access data on images that have finished and are waiting for synchronization. Error termination on one image causes error termination on the other images.

Therefore, there are three options available to a Fortran program. First, it can detect an error locally and handle it; second, it can detect an error and halt one image; and third, it can detect an error and signal all images to halt.

Fortran.39.2 Guidance to Fortran Users

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

- Decide upon a strategy for handling errors, and consistently use it across all portions of the program.
- Use `stop` or `error stop` as appropriate.

Fortran.40 Type-breaking Reinterpretation of Data [AMV]

Fortran.40.1 Applicability to Fortran

Storage association via `common` or `equivalence` statements, or via the `transfer` intrinsic procedure can cause a type-breaking reinterpretation of data. Type-breaking reinterpretation via `common` and `equivalence` is not standard-conforming.

Fortran.40.2 Guidance to Fortran Users

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

- Do not use `common` to share data. Use modules instead.
- Do not use `equivalence` to save storage space. Use allocatable data instead.
- Avoid use of the `transfer` intrinsic unless its use is unavoidable, and then document the use carefully.
- Use a processor option, if available, to detect violation of the rules for `common` and `equivalence`.

Fortran.41 Memory Leak [XYL]

Fortran.41.1 Applicability to Fortran

The misuse of pointers in Fortran can cause a memory leak. However, the intended functionality is often available with allocatables, which do not suffer from this vulnerability.

Fortran.41.2 Guidance to Fortran Users

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

- Use allocatable data items rather than pointer data items whenever possible.
- Use `final` routines to free memory resources allocated to a data item.

Fortran.42 Templates and Generics [SYM]

Fortran.42.1 Applicability to Fortran

Fortran does not support templates or generics, so this vulnerability does not apply.

Fortran.43 Inheritance [RIP]

Fortran.43.1 Applicability to Fortran

Fortran supports inheritance so this vulnerability applies.

Fortran supports single inheritance only, so the complexities associated with multiple inheritance do not apply.

Fortran.43.2 Guidance to Fortran Users

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

- Declare a type-bound procedure to be `non_overridable` when necessary to ensure that it is not overridden.
- Provide a component to store the version control identifier of the derived type, together with an accessor routine.

Fortran.44 Extra Intrinsics [LRM]

Fortran.44.1 Applicability to Fortran

Fortran permits a processor to supply extra intrinsics.

The processor that provides extra intrinsics is standard-conforming, the program that uses one is not.

Fortran.44.2 Guidance to Fortran Users

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

- Specify that an intrinsic or external procedure has the `intrinsic` or `external` attribute, respectively, in the scope where the reference occurs.

- Use compiler options to detect use of non-standard intrinsic procedures.

Fortran.45 Argument Passing to Library Functions [TRJ]

Fortran.45.1 Applicability to Fortran

Fortran allows use of libraries so this vulnerability applies.

Fortran.45.2 Guidance to Fortran Users

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

- Use libraries from reputable sources with reliable documentation and understand the documentation to appreciate the range of acceptable input.
- Verify arguments to intrinsic procedures when their validity is in doubt.
- Use condition constructs such as `if` and `where` to prevent invocation of an intrinsic procedure with invalid arguments.

Fortran.46 Inter-language Calling [DJS]

Fortran.46.1 Applicability to Fortran

Fortran supports interoperating with functions and data that may be specified by means of the C programming language. The facilities limit the interactions and thereby limit the extent of this vulnerability.

Fortran.46.2 Guidance to Fortran Users

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

- Use the `iso_c_binding` module, and use the correct constants therein to specify the type kind values needed.
- Use the `value` attribute as needed for dummy arguments.

Fortran.47 Dynamically-linked Code and Self-modifying Code [NYY]

Fortran.47.1 Applicability to Fortran

The Fortran standard does not discuss the means of program translation, so any use or misuse of dynamically linked libraries is processor dependent.

Fortran does not permit self-modifying code.

Fortran.47.2 Guidance to Fortran Users

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

- Use compiler options to effect a static link.

Fortran.48 Library Signature [NSQ]

Fortran.48.1 Applicability to Fortran

Fortran allows the use of libraries, so this vulnerability applies.

Fortran.48.2 Guidance to Fortran Users

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

- Use interface blocks for the library code if they are available. Avoid libraries that do not provide a module to supply explicit interfaces.
- Carefully construct interface blocks for the library procedures where library modules are not provided.

Fortran.49 Unanticipated Exceptions from Library Routines [HJW]

Fortran.49.1 Applicability to Fortran

Fortran allows the use of libraries so this vulnerability applies.

Fortran.49.2 Guidance to Fortran Users

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

- Check any return flags present and, if an error is indicated, take appropriate actions when calling a library procedure.

Fortran.50 Pre-processor Directives [NMP]

Fortran.50.1 Applicability to Fortran

The Fortran standard does not include pre-processing, so this vulnerability does not apply to standard programs. However, some Fortran programmers employ the C pre-processor cpp, or other pre-processors.

The C pre-processor, as defined by the C language, is unaware of several Fortran source code properties. Some suppliers of Fortran compilers also supply a Fortran-aware version of cpp. Unless a Fortran-aware version of cpp is used, unexpected results, not always easily detected, may occur.

Other pre-processors may or may not be aware of Fortran source code properties. Not all pre-processors have a Fortran-aware mode that could be used to reduce the probability of erroneous results.

Fortran.50.2 Guidance to Fortran Users

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

- Avoid use of the C pre-processor cpp.
- Avoid pre-processors generally. Where deemed necessary, a Fortran mode should be set.
- Use processor-specific modules in place of pre-processing whenever possible.

Fortran.51 Suppression of Language-defined Run-time Checking [MXB]

Fortran.51.1 Applicability to Fortran

The Fortran standard has many requirements that cannot be checked at compile time. While many compilers provide options for run-time checking, the standard does not require that any such checks be provided.

Fortran.51.2 Guidance to Fortran Users

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

- Use all run-time checks that are available during development.
- Use all run-time checks that are available during production running, except where critical for performance.
- Use several compilers during development to check as many conditions as possible.

Fortran.52 Provision of Inherently Unsafe Operations [SKL]

Fortran.52.1 Applicability to Fortran

The types of actual arguments and corresponding dummy arguments are required to agree, but few compilers check this unless the procedure has an explicit interface.

The intrinsic function `transfer` provides the facility to transform an object of one type to an object of another type that has the same physical representation.

A variable of one type may be storage associated through the use of `common` and `equivalence` with a variable of another type. Defining the value of one causes the value of the other to become undefined.

There are facilities for invoking C functions from Fortran and Fortran procedures from C. While there are rules about type agreement for the arguments, it is unlikely that implementations will check them.

Fortran.52.2 Guidance to Fortran Users

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

- Provide an interface block for each external procedure or replace the procedure by an internal or module procedure.
- Avoid the use of the intrinsic function `transfer`.
- Avoid the use of `common` and `equivalence`.
- Use the compiler or other automatic tool for checking the types of the arguments in calls between Fortran and C, make use of them during development and in production running except where performance would be severely affected.

Fortran.53 Obscure Language Features [BRS]

Fortran.53.1 Applicability to Fortran

Any use of deleted and obsolescent features, see MEM, might produce semantic results not in accord with the modern programmer's expectations. They might be beyond the knowledge of modern code reviewers.

Variables may be storage associated through the use of `common` and `equivalence`. Defining the value of one alters the value of the other. They may be of different types, in which case defining the value of one causes the value of the other to become undefined.

Supplying an initial value for a local variable implies that it has the `save` attribute, which might be unexpected by the developer.

If implicit typing is used, a simple spelling error might unexpectedly introduce a new name. The intended effect on the given variable will be lost without any compiler diagnostic.

Fortran.53.2 Guidance to Fortran Users

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

- Use the processor to detect and identify obsolescent or deleted features and replace them by better methods.

- Avoid the use of `common` and `equivalence`.
- Specify the `save` attribute when supplying an initial value.
- Use `implicit none` to require explicit declarations.

Fortran.54 Unspecified Behaviour [BQF]

Fortran.54.1 Applicability to Fortran

This vulnerability is described by Implementation-defined Behaviour [FAB].

Fortran.55 Undefined Behaviour [EWF]

Fortran.55.1 Applicability to Fortran

A Fortran processor is unconstrained unless the program uses only those forms and relations specified by the Fortran standard, and gives them the meaning described therein.

The behaviour of non-standard code can change between processors.

A processor is permitted to provide additional intrinsic procedures. One on these might be invoked instead of an intended external procedure with the same name.

Fortran.55.2 Guidance to Fortran Users

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

- Use processor options to detect and report use of non-standard features.
- Obtain diagnostics from more than one source, for example, use code checking tools.
- Specify the `external` attribute for all external procedures invoked.

Fortran.56 Implementation-defined Behaviour [FAB]

Fortran.56.1 Applicability to Fortran

Implementation-defined behaviour is known within the Fortran standard as processor-dependent. Annex A.2 of ISO/IEC 1539-1 (2010) contains a list of processor dependencies.

Different processors might process processor dependencies differently. Relying on one behaviour is not guaranteed by the Fortran standard.

Reliance on one behaviour where the standard explicitly allows several is not portable. The behaviour is liable to change between different processors.

Fortran.56.2 Guidance to Fortran Users

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

- Do not rely on the behaviour of processor dependencies. See Annex A.2 of ISO/IEC 1539-1 (2010) for a complete list.

Fortran.57 Deprecated Language Features [MEM]

Fortran.57.1 Applicability to Fortran

Because they are still used in some programs, many compilers support features of previous revisions of the Fortran standard that were deleted in later versions of the Fortran standard. These are listed in Annex B.1 of the Fortran standard. In addition, there are features of earlier revisions of Fortran that are still in the standard but are redundant and may be replaced by better methods. They are described in small font in the standard and are summarized in Annex B.2. Any use of these deleted and obsolescent features might produce semantic results not in accord with the modern programmer's expectations. They might be beyond the knowledge of modern code reviewers.

Fortran.57.2 Guidance to Fortran Users

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

- Use the processor to detect and identify obsolescent or deleted features and replace them by better methods.

Fortran.58 Implications for Standardization

Fortran.58.1 Implications for Standardization

Future standardization efforts should consider:

- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of integer overflows during program execution.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of out-of-bounds subscripts and array-shape mismatches in assignment statements during program execution.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of invalid pointer references during program execution.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of an invalid use of character constants as format specifiers.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of tests for equality between two objects of type real or complex.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of pointer assignment of a pointer whose lifetime is known to be longer than the lifetime of the target.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of the reuse of a name within a nested scope.
- Providing a means to specify explicitly a limited set of entities to be accessed by host association.

- Identifying, deprecating, and replacing features whose use is problematic where there is a safer and clearer alternative in the modern revisions of the language or in current practice in other languages.