ISO/IEC JTC 1/SC 22 N0703

Date: 2017-03-10

ISO/IEC TR 24772-8

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Stephen Michell 2016-3-7 11:18 AM
**Deleted:** N 0000

Stephen Michell 2016-3-7 11:18 AM
**Deleted:** 5-06-19

DRAFT DRAFT DRAFT

# Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Vulnerability descriptions for the programming language Fortran

*Élément introductif — Élément principal — Partie n: Titre de la partie*

**Warning**

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: International standard
Document subtype: if applicable
Document stage: (10) development stage
Document language: E

**Contents**

Page

## Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772-8, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

# Introduction

This Technical Report provides guidance for the programming language Fortran so that application developers considering Fortran or using Fortran will be better able to avoid the programming constructs that lead to vulnerabilities in software written in the Fortran language and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This technical can also be used in comparison with companion technical reports and with the language-independent report, TR 24772-1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This technical report part is intended to be used with TR 24772-1, which discusses programming language vulnerabilities in a language independent fashion.

It should be noted that this Technical Report is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such report can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

# Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages through language selection and use – Vulnerability descriptions for the programming language Fortran

## 1. Scope

This Technical Report specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.

Vulnerabilities described in this technical report document the way that the vulnerability described in the language-independent writeup (in Tr 24772-1) are manifested in Fortran.

## 2. Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

*ISO/IEC TR 24772-1 Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages, Part 1, General Guidance*
*ISO/IEC 1539-1:2010, Information technology -- Programming languages -- Fortran -- Part 1: Base language*
*ISO/IEC 1539-2:2000, Information technology – Programming languages – Fortran – Varying length character strings*
*ISO/IEC 1539-3:1999,* Information *technology -- Programming languages -- Fortran -- Part 3: Conditional compilation*
ISO 80000–2:2009, *Quantities and units — Part 2: Mathematical signs and symbols to be use in the natural sciences and technology*
ISO/IEC 2382–1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*

ISO IEC ???? 854-1987, Radix-Independent Floating-Point Arithmetic, IEEE, 1987

## 3. Terms and definitions, symbols and conventions

### 3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382–1, in TR 24772-1 and the following apply. Other terms are defined where they appear in *italic* type.

The precise statement of the following definitions can be found in 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 older versions of Fortran 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 data representation methods

***module***: a separate scope that contains definitions that can 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 one of a set of 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

## 4 Language concepts

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 specify the size or complexity of a program that might 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. A subprogram is standard-conforming if it can 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 are able to detect and report the use of extensions.

Annex B.1 of the Fortran standard lists six features of older versions of Fortran that have been deleted because they were redundant and considered largely unused. Although no longer part of the standard, they are supported by many processors to allow old programs to continue to run. Annex B.2 lists ten features of Fortran 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 are able to detect and report the use of these features.

The Fortran standard defines a set of intrinsic procedures and intrinsic modules, and allows a processor to extend this set with further procedures and modules. A program that uses an intrinsic procedure or module not defined by the standard is not standard-conforming. A program that uses an entity not defined by the standard from a module defined by the standard is not standard-conforming. Use of intrinsic procedures or modules not defined by the standard should be avoided. Use of entities not defined by the standard from intrinsic modules should be avoided. Processors are able to detect and report the use of intrinsic procedures 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.2 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 occasionally 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.

Processors are required to detect deviation from the standard so far as can be determined from syntax rules and constraints during translation only, and not during execution of a program. It is the responsibility of the program 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 subscript 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 intrinsic data objects are described 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 entities of type character and type logical.

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 might 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 asynchronously executing 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. Synchronization statements are defined to allow a program to synchronize its images. 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 processors 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 processors 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. Many modern processors generally continue to support deleted features from older revisions of the Fortran standard, and even some extensions from older processors, 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 might be reluctance to change 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 criteria for modernizing trusted old code.

## 5 General guidance for Fortran

In addition to the Top 10 generic programming rules from TR 24772-1 clause 5.4, additional rules from this section apply specifically to the C programming language. The recommendations of this section are restatements of recommendations from clause 6, but represent ones stated frequently, or that are considered as particularly noteworthy by the authors. Clause 6 of this document contains the full set of recommendations, as well as explanations of the problems that led to the recommendations made.

Every guidance provided in this section, and in the corresponding Part section, is supported material in Clause 6 of this document, as well as other important recommendations.

*What do we do with generic rules that do not apply to this Part?*

*What guidance do we give when the generic rule is highly qualified here?*

| Number | Recommended avoidance mechanism | References |
|---|---|---|
| 1 | Never use implicit typing. Always declare all variables. Use implicit none to enforce this. | |
| 2 | Use explicit conversion intrinsics for the conversion of values of intrinsic types, 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. | |
| 3 | Use a temporary variable with a large range to read a value from an untrusted source so that the value can be checked against the limits provided by the inquiry intrinsics for the type and kind of the variable to be used. Similarly, use a temporary variable with a large range to hold the value of an expression before assigning it to a variable of a type and kind that has a smaller numeric range to ensure that the value of the expression is within the allowed range for the variable. When assigning an expression of one type and kind to a variable of a type and kind that might have a smaller numeric range, check that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable. | |
| 4 | Use whole array assignment, operations, and bounds inquiry intrinsics where possible. | |
| 5 | Obtain array bounds from array inquiry intrinsics wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable array as procedure dummy arguments to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays. | |
| 6 | Use default initialization in the declarations of pointer components. | |
| 7 | Specify pure (or elemental) for procedures where possible for greater clarity of the programmer's intentions. | |
| 8 | Code a status variable for all statements that support one, and examine its 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.. | |
| 9 | Avoid the use of common and equivalence. Use modules instead of common to share data. Use allocatable data instead of equivalence. | |
| 10 | Supply an explicit interface to specify the external attribute for all external procedures invoked. | |

Stephen Michell 2017-3-7 12:13 PM
**Formatted:** Font:Not Bold, English

# 6 Specific Guidance for Fortran

## 6.1 General

This clause contains specific advice for Fortran about the possible presence of vulnerabilities as described in TR 24772-1, and provides specific guidance on how to avoid them in Fortran program code. This section mirrors TR 24772-1 clause 6 in that the vulnerability "Type System [IHN]" is found in 6.2 of TR 24772-1, and Fortran specific guidance is found in clause 6 and subclauses in this TR.

## 6.2 Type System [IHN]

### 6.2.1 Applicability to language

The Fortran type system is a strong type system consisting of the data type and type parameters. A type parameter is an integer value that specifies a parameterization of the type; a user-defined type need not have any type parameters. Objects of the same type that differ in the value of their type parameter(s) might differ in representation, and therefore in the limits of the values they can represent. For many purposes for which other languages use type, Fortran uses the type, type parameters, and rank of a data object. A conforming processor supports at least two kinds of type real and a complex kind corresponding to each supported real kind. Double precision real is required to provide more digits of decimal precision than default real. A conforming processor supports 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 might result in a loss of information or an undetected failure to conform to the standard. For example, if a double-precision real is assigned to a single-precision real, round-off is likely; and if an integer operation results in a value outside the supported range, the program is not conforming. This might not be detected. Likewise, assigning a value to an integer variable whose range does not include the value, renders the program not conforming.

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 type when their type definitions are the same instance of text (which can be made available to other program units by module use). Sequence types and bind(c) types represent a narrow exception to this rule. Sequence types are less commonly used because they are less convenient to use, cannot be extended, and cannot interoperate with types defined by a companion processor. Bind(c) types are, in general, only used to interoperate with types defined by a companion processor; they also cannot be extended.

A derived type can have type parameters and these parameters can be applied to the derived type's components. Default assignment of variables of the same derived type is component-wise. Default assignment can be

Stephen Michell 2016-3-7 11:20 AM
**Deleted:** <#>*[ See Template] [Thoughts welcomed as to what could be provided here. Possibly an opportunity for the language community to address issues that do not correlate to the guidance of section 6. For languages that provide non-mandatory tools, how those tools can be used to provide effective mitigation of vulnerabilities described in the following sections]*

Stephen Michell 2016-3-7 11:24 AM
**Formatted:** Font:Italic

overridden by an explicitly coded assignment procedure. For derived-type objects, type changing assignments and conversion procedures are required to be explicitly coded by the programmer. Other than default assignment, each operation on a derived type is defined by a procedure. These procedures can contain any necessary checks and coercions.

In addition to the losses mentioned in Clause 6 of ISO/IEC TR 24772, assignment of a complex entity to a noncomplex variable only assigns the real part.

Assignment of an object of extended type to one of base type only assigns the base type part.

Intrinsic functions can be used in constant expressions that compute desired kind type parameter values. Also, the intrinsic module `iso_fortran_env` supplies named constants suitable for kind type parameters.

### 6.2.2 Guidance to language users

- 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 of values of intrinsic types, 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 avoid implicit 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.

## 6.3 Bit Representation [STR]

### 6.3.1 Applicability to language

Fortran defines bit positions by a *bit model* described in Subclause 13.3 of the standard. Care should 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 can be assigned to named constants thereby providing a name for a mask.

Fortran provides access to individual bits within a storage unit by bit manipulation 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 interpret, or not interpret, the left-most bit as the sign bit.

### 6.3.2 Guidance to language users

- 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 favour 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.

## 6.4 Floating-point Arithmetic [PLF]

Stephen Michell 2016-3-7 11:26 AM

**Comment [1]:** Confirm that the FP issues updated in -1 at the June 2015 meeting are reflected here.

### 6.4.1 Applicability to language

Fortran supports floating-point data. Furthermore, most processors support parts of the IEEE 754 standard and facilities are provided for the programmer to detect the extent of conformance.

The rounding mode in effect during translation might differ from the rounding mode in effect during execution; the rounding mode could change during execution. A separate rounding mode is provided for input/output formatting conversions, this rounding mode could also change during execution.

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

### 6.4.2 Guidance to language users

- 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 proceeds as expected.
- Avoid creating 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. (This relies on a deleted feature.) A floating-point value can 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 the use of bit operations to get or to set 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, where 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, where the IEEE intrinsic modules are in use.

## 6.5 Enumerator Issues [CCB]

### 6.5.1 Applicability to language

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.

### 6.5.2 Guidance to language users

- 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 companion processor 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.

## 6.6 Numeric Conversion Errors [FLC]

### 6.6.1 Applicability to language

Fortran processors are required to support two kinds of type real and are required to support a complex kind for every real kind supported. Fortran processors are required to 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.

### 6.6.2 Guidance to language users

- Use the kind selection intrinsic procedures to select sizes of variables supporting the required operations and values.
- Use a temporary variable with a large range to read a value from an untrusted source so that the value can be checked against the limits provided by the inquiry intrinsics for the type and kind of the variable to be used.
- Use a temporary variable with a large range to hold the value of an expression before assigning it to a variable of a type and kind that has a smaller numeric range to ensure that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.
- When assigning an expression of one type and kind to a variable of a type and kind that might have a smaller numeric range, check that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.
- 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.

## 6.7 String Termination [CJM]

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

## 6.8 Buffer Boundary Violation (Buffer Overflow) [HCB]

A Fortran program might be affected by this vulnerability in two situations. The first is that an array subscript 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 subscript be separately within its bounds, not simply that the resulting offset be within the array as a whole.

Fortran does not mandate array subscript 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 where the left-hand side is not an allocatable object. However, Fortran does not mandate that array shapes be checked during 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 length mismatch occurs, the assignment has a blank-fill (if the value is too short) or truncate (if the value is too long) semantic. Otherwise, the variable defined is resized, if needed, to the correct size.

Most implementations include an optional facility for bounds checking. These are likely to 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, or because the local declaration of the bounds might be inconsistent with those of the actual argument. It is therefore preferable to use an assumed-shape array as a procedure dummy argument. The performance of operations involving assumed-shape arrays is improved by the use of the `contiguous` attribute.

Fortran provides a set of array bounds intrinsic inquiry procedures which can 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 can be reliably found.

### 6.8.2 Guidance to language users

- 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 intrinsic procedures wherever needed. Use explicit interfaces and

assumed-shape arrays or allocatable
- dummy arguments to ensure that array shape information is passed to all procedures where needed, and can be used to dimension local automatic arrays.
- Use allocatable arrays where array 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 storing outside the assigned variable.

## 6.9 Unchecked Array Indexing [XYZ]

### 6.9.1 Applicability to language

A Fortran program might be affected by this vulnerability in the situation an array subscript could be outside its bounds. The Fortran standard requires that each array subscript 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 during 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.

Most processors include an optional facility for bounds checking. These are likely to 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, or because the local declaration of the bounds might be inconsistent with those of the actual argument. It is therefore preferable to use an assumed-shape array as a procedure argument. The performance of operations involving assumed-shape arrays is improved by the use of the `contiguous` attribute.

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

### 6.9.2 Guidance to language users

- 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 intrinsic procedures wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable arrays as procedure dummy arguments to ensure that array shape information is passed to all procedures where needed, and can be used to dimension local automatic arrays.
- Use allocatable arrays where 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 subscript

arithmetic.
- Arrays can be declared in modules which makes their bounds information available wherever the array is available.

## 6.10 Unchecked Array Copying [XYW]

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

Most implementations include an optional facility for bounds checking. These are likely to 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 or allocatable array as a procedure dummy argument. The performance of operations involving assumed-shape arrays is improved by the use of the `contiguous` attribute.

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

### 6.10.2 Guidance to language users

- 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 wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable array as procedure dummy arguments to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.
- Use allocatable arrays where arrays operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.

## 6.11 Pointer Type Conversions [HFC]

### 6.11.1 Applicability to language

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 can be associated only with an object of its type. A polymorphic pointer that is not unlimited polymorphic is declared with a type and can 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. An unlimited polymorphic pointer can also be assigned to a sequence type or bind(c) type pointer; this is unsafe, and cannot be checked during compilation.

When an unlimited polymorphic pointer has a target of a sequence type or an interoperable derived type, a type-breaking cast might 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 could be made if the pointer is processed by procedures written in a language other than Fortran.

### 6.11.2 Guidance to language users

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

## 6.12 Pointer Arithmetic [RVG]

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

## 6.13 Null Pointer Dereference [XYH]

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

A Fortran pointer by default is initially undefined and not nullified. A pointer is only nullified when it is done explicitly, either by pointer assigning the result of the `null` intrinsic procedure or by the `nullify` statement.

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.

### 6.13.2 Guidance to language users

Stephen Michell 2016-3-7 11:29 AM
**Formatted:** Font:Bold

Stephen Michell 2016-3-7 11:29 AM
**Formatted:** Heading 2

- Use compiler options where available to 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 referencing a target through the pointer if there is any possibility of it being disassociated.
- Associate pointers before referencing them.
- Use default initialization in the declarations of pointer components.
- Use initialization in the declarations of all pointers that have the save attribute.

## 6.14 Dangling Reference to Heap [XYK]

### 6.14.1 Applicability to language

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

### 6.14.2 Guidance to language users

- 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.

- Use compiler options where available to 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 if the pointer might have a longer lifetime than the target or the target attribute of the target. Check actual arguments that are argument associated with dummy arguments that are given the target attribute within the referenced procedure.
- Check for successful deallocation when deallocating a pointer by using the `stat=` specifier.

## 6.15 Arithmetic Wrap-around Error [FIF]

### 6.15.1 Applicability to language

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

### 6.15.2 Guidance to language users

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

## 6.16 Using Shift Operations for Multiplication and Division [PIK]

### 6.16.1 Applicability to language

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

### 6.16.2 Guidance to language users

- 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.

## 6.17 Choice of Clear Names [NAI]

### 6.17.1 Applicability to language

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

A name can include underscore characters, except in the initial position. The number of consecutive underscores is significant but might be difficult to see.

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

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

### 6.17.2 Guidance to language users

- Declare all variables and use `implicit none` to enforce this.
- Do not attempt to 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.

## 6.18 Dead store [WXQ]

### 6.18.1 Applicability to language

Fortran provides assignment so this is applicable.

### 6.18.2 Guidance to Language Users

- 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 processor.
- Do not use similar names in nested scopes.

## 6.19 Unused Variable [YZS]

### 6.19.1 Applicability to language

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

### 6.19.2 Guidance to language users

- Use a processor that can detect a variable that is declared but not used and enable the processor's option to do so at all times.
- Use processor options where available or a static analysis to detect variables to which a value is assigned but are not referenced.

## 6.20 Identifier Name Reuse [YOW]

### 6.20.1 Applicability to language

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

- Module procedures have a nested scope within their module host.
- Internal procedures have a nested scope within their (procedure) host.
- A block construct might have a nested scope within the host scope.
- An array constructor might have a nested scope.

The index variables of some constructs, such as or do `concurrent`, `forall`, or array constructor implied do loops, are local to the construct. A select name in an `associate` or `select type` construct is local to the construct.

### 6.20.2 Guidance to language users

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

## 6.21 Namespace Issues [BJL]

Stephen Michell 2017-3-7 12:23 PM
**Formatted:** Heading 2

Stephen Michell 2017-3-7 12:23 PM
**Deleted:**

### 6.21.1 Applicability to language

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 could arise.

Specifically, a variable that appears in the local scope but is not explicitly declared, might 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.

### 6.21.2 Guidance to language users

- Never use implicit typing. Always declare all variables. Use `implicit none` to enforce this.
- Use a global `private` statement in all modules to require explicit specification of the `public` attribute.
- Use an `only` clause on every `use` statement.
- Use renaming when needed to avoid name collisions.

## 6.22 Initialization of Variables [LAV]

### 6.22.1 Applicability to language

The value of a variable that has never been given a value is undefined. It is the programmer's responsibility to guard against use of uninitialized variables.

### 6.22.2 Guidance to language users

- 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.

## 6.23 Operator Precedence and Associativity [JCW]

Stephen Michell 2016-3-7 11:30 AM
**Deleted:** /Order of Evaluation

### 6.23.1 Applicability to language

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.`, and `.neqv.`. In addition, any monadic defined operator, the intrinsic operator `//`, and any dyadic defined operator have a position in this order, but these positions are not well known.

### 6.23.2 Guidance to language users

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

## 6.24 Side-effects and Order of Evaluation [SAM]

### 6.24.1 Applicability to language

Fortran functions are permitted to have side effects, unless the function is declared to have the `pure` attribute. 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 processor.

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.

### 6.24.2 Guidance to language users

- 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.

## 6.25 Likely Incorrect Expression [KOA]

### 6.25.1 Applicability to language

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 processors allow a dyadic operator immediately preceding a unary operator, which should be avoided. However, this can be detected by using processor 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.

### 6.25.2 Guidance to language users

- 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 processor’s ability to detect such occurrences.

## 6.26 Dead and Deactivated Code [XYQ]

### 6.26.1 Applicability to language

There is no requirement in the Fortran standard for processors 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 converted to comments. Alternatively, a source code control package can be used to preserve the text of older versions of a program.

### 6.26.2 Guidance to language users

- Use a compiler, or other tool, that can detect dead or deactivated code.
- Use a coverage tool to check that the test suite causes every statement to be 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.

## 6.27 Switch Statements and Static Analysis [CLL]

### 6.27.1 Applicability to language

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

Fortran has a computed `go to` statement that allows control to flow from one alternative to another, and allows other unexpected flow of control.

### 6.27.2 Guidance to language users

- 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.
- Avoid the use of computed `go to` statements.

## 6.28 Demarcation of Control Flow [EOJ]

### 6.28.1 Applicability to language

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

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

### 6.28.2 Guidance to language users

- 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.

## 6.29 Loop Control Variables [TEX]

### 6.29.1 Applicability to language

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 processor 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.

### 6.29.2 Guidance to language users

- 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.
- Verify that where the iteration variable is an actual argument, it is associated with an `intent(in)` or a `value` dummy argument.

## 6.30 Off-by-one Error [XZH]

### 6.30.1 Applicability to language

Fortran is not very susceptible 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 might 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.

### 6.30.2 Guidance to language users

- 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, where doing so reduces the overall amount of explicit subscript arithmetic.

## 6.31 Structured Programming [EWD]

### 6.31.1 Applicability to language

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 processor 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.

### 6.31.2 Guidance to language users

- 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.

## 6.32 Passing Parameters and Return Values [CSJ]

### 6.32.1 Applicability to language

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

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, intrinsic procedures, and internal procedures have explicit interfaces. An external procedure has an explicit interface only when one is provided by a procedure declaration or interface body. Such an interface body could be generated automatically using a software tool. Explicit interfaces allow processors to check the type, kind, and rank of arguments and result variables of functions.

### 6.32.2 Guidance to language users

- 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.

## 6.33 Dangling References to Stack Frames [DCM]

### 6.33.1 Applicability to language

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.

### 6.33.2 Guidance to language users

- Do not pointer-assign a pointer to a target if the pointer association might have a longer lifetime than the target or the `target` attribute of the target.
- Use allocatable variables in preference to pointers wherever they provide sufficient functionality.

## 6.34 Subprogram Signature Mismatch [OTR]

### 6.34.1 Applicability to language

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

The Fortran standard requires that interfaces match, but does not require that the processor diagnoses mismatches. However, processors do check this when the interface is explicit. Some processors can check interfaces if inter-procedural analysis is requested.

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

### 6.34.2 Guidance to language users

- Use explicit interfaces, preferably by placing procedures inside a module or another procedure.
- Use a processor that checks all interfaces, especially if this can be checked during compilation with no execution overhead.
- Use a processor or other tool to create explicit interface bodies for external procedures.

## 6.35 Recursion [GDL]

### 6.35.1 Applicability to language

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

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

### 6.35.2 Guidance to language users

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

## 6.36 Ignored Error Status and Unhandled Exceptions [OYB]

### 6.36.1 Applicability to language

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 intrinsic modules, Fortran does not support exception handling.

### 6.36.2 Guidance to language users

- Code a status variable for all statements that support one, and examine its 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 might be returned by an intrinsic procedure or by a library procedure.

## 6.37 Type-breaking Reinterpretation of Data [AMV]

Stephen Michell 2017-3-7 12:29 PM
**Moved (insertion) [1]**

### 6.37.1 Applicability to language

Stephen Michell 2017-3-7 12:29 PM
**Deleted:** 8

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.

Stephen Michell 2017-3-7 12:29 PM
**Deleted:** 8

### 6.37.2 Guidance to language users

Stephen Michell 2017-3-7 12:29 PM
**Deleted:** 8

- 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 compiler options where available to detect violation of the rules for common and equivalence.

## 6.38 Deep vs. Shallow Copying [YAN]

### 6.38.1 Applicability to language

TBD

Stephen Michell 2017-3-7 12:30 PM
**Formatted:** Normal

Stephen Michell 2017-3-7 12:30 PM
**Formatted:** English (US)

### 6.38.2 Guidance to language users

TBD

Stephen Michell 2017-3-7 12:30 PM
**Formatted:** English (US)

Stephen Michell 2017-3-7 12:30 PM
**Formatted:** Normal

Stephen Michell 2017-3-7 12:29 PM
**Moved up [1]:** 6.38 Type-breaking Reinterpretation of Data [AMV]

Stephen Michell 2017-3-7 12:31 PM
**Deleted:** 6.37 Fault Tolerance and Failure Strategies [REW] ... [1]

- 
- 
- 
- 

## 6.39 Type-breaking Reinterpretation of Data

TBD

Stephen Michell 2016-3-7 11:37 AM
**Formatted:** Normal

Stephen Michell 2016-3-7 11:37 AM
**Deleted:** 39

## 6.39 Memory Leaks and Heap Fragmentation [XYL]

Stephen Michell 2016-3-7 11:37 AM
**Deleted:** 39

### 6.39.1 Applicability to language

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.

### 6.39.2 Guidance to language users

Stephen Michell 2016-3-7 11:37 AM
**Deleted:** 39

- Use allocatable data items rather than pointer data items whenever possible.
- Use final routines to free memory resources allocated to a data item of derived type.
- Use a tool during testing to detect memory leaks.

## 6.40 Templates and Generics [SYM]

Stephen Michell 2016-3-7 11:38 AM
**Deleted:** 0

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

## 6.41 Inheritance [RIP]

Stephen Michell 2016-3-7 11:38 AM
**Deleted:** 1

### 6.41.1 Applicability to language

Stephen Michell 2016-3-7 11:38 AM
**Deleted:** 1

Fortran supports inheritance so this vulnerability applies.

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

### 6.41.2 Guidance to language users

Stephen Michell 2016-3-7 11:38 AM
**Deleted:** 1

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

## 6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP]

### 6.42.1 Applicability to language

TBD

### 6.42.2 Guidance to language users

TBD

## 6.43 Redispatching [PPH]

### 6.43.1 Applicability to language

TBD

### 6.43. 2 Guidance to language users

TBD

## 6.44 Polymorphic Variables

### 6.44.1 Applicability to language

TBD

### 6.44. 2 Guidance to language users

TBD

## 6.45 Extra Intrinsics [LRM]

### 6.45.1 Applicability to language

Fortran permits a processor to supply extra intrinsic procedures.

The processor that provides extra intrinsic procedures might be standard-conforming; the program that uses one is not.

### 6.45.2 Guidance to language users

- 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.

Stephen Michell 2016-3-7 11:39 AM
**Formatted:** Heading 2

Stephen Michell 2016-3-7 11:41 AM
**Formatted:** Font:11 pt

Stephen Michell 2017-3-7 12:34 PM
**Formatted:** Normal

Stephen Michell 2016-3-7 11:42 AM
**Formatted:** Normal

Stephen Michell 2016-3-7 11:40 AM
**Formatted:** Heading 3

Stephen Michell 2016-3-7 11:41 AM
**Deleted:** 2

Stephen Michell 2016-3-7 11:42 AM
**Deleted:** 2

Stephen Michell 2016-3-7 11:42 AM
**Deleted:** 2

### 6.46 Argument Passing to Library Functions [TRJ]

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 3

#### 6.46.1 Applicability to language

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 3

Fortran allows use of libraries so this vulnerability applies.

#### 6.46.2 Guidance to language users

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 3

- Use libraries from reputable sources with reliable documentation and understand the documentation to appreciate the range of acceptable input.
- Verify arguments to library procedures when their validity is in doubt.
- Use condition constructs such as `if` and `where` to prevent invocation of a library procedure with invalid arguments.
- Provide explicit interfaces for library procedures. If the library provides a module containing interface bodies, use the module.

### 6.47 Inter-language Calling [DJS]

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 4

#### 6.47.1 Applicability to Language

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 4

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

#### 6.47.2 Guidance to Language Users

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 4

- Correctly identify the companion processor, including any options affecting its types.
- 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.

### 6.48 Dynamically-linked Code and Self-modifying Code [NYY]

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 5

#### 6.48.1 Applicability to language

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 5

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.

#### 6.48.2 Guidance to language users

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 5

- Use compiler options to effect a static link.

### 6.49 Library Signature [NSQ]

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 46

#### 6.49.1 Applicability to language

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 46

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

#### 6.49.2 Guidance to language users

Stephen Michell 2016-3-7 11:43 AM
**Deleted:** 46

- Use explicit interfaces for the library code if they are available. Avoid libraries that do not provide explicit

interfaces.

- Carefully construct explicit interfaces for the library procedures where library modules are not provided.
- Prefer libraries that provide procedures as module procedures rather than as external procedures.

## 6.50 Unanticipated Exceptions from Library Routines [HJW]

Stephen Michell 2016-3-7 11:44 AM
**Deleted:** 48

### 6.50.1 Applicability to language

Stephen Michell 2016-3-7 11:44 AM
**Deleted:** 48

Fortran allows the use of libraries so this vulnerability applies.

### 6.50.2 Guidance to language users

Stephen Michell 2016-3-7 11:44 AM
**Deleted:** 47

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

## 6.51 Pre-Processor Directives [NMP]

Stephen Michell 2016-3-7 11:44 AM
**Deleted:** 48

### 6.51.1 Applicability to language

Stephen Michell 2016-3-7 11:44 AM
**Deleted:** 48

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 processors also supply a Fortran-aware version of `cpp`. Unless a Fortran-aware version of `cpp` is used, unexpected results, not always easily detected, can occur.

Other pre-processors might or might 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.

### 6.51.2 Guidance to language users

Stephen Michell 2016-3-7 11:45 AM
**Deleted:** 48

- 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 wherever possible.

## 6.52 Suppression of Language-defined Run-time Checking [MXB]

Stephen Michell 2016-3-7 11:45 AM
**Deleted:** 49

### 6.52.1 Applicability to Language

Stephen Michell 2016-3-7 11:45 AM
**Deleted:** 49

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

### 6.52.2 Guidance to Language Users

Stephen Michell 2016-3-7 11:45 AM
**Deleted:** 49

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

## 6.53 Provision of Inherently Unsafe Operations [SKL]

### 6.53.1 Applicability to Language

The types of actual arguments and corresponding dummy arguments are required to agree, but few processors 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 can 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. A processor might not be able to detect this.

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 processors will check them.

### 6.53.2 Guidance to language users

- Provide an explicit interface 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.

## 6.54 Obscure Language Features [BRS]

### 6.54.1 Applicability to language

Any use of deleted and obsolescent features, see 6.58 Deprecated Language Features, might produce semantic results not in accord with the modern programmer's expectations. They might be beyond the knowledge of modern code reviewers.

Variables can be storage associated through the use of common and equivalence. Defining the value of one alters the value of the other. They might 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 processor diagnostic.

### 6.54.2 Guidance to language users

- Use the processor to detect and identify obsolescent or deleted features and replace them by better methods.
- Avoid the use of common and equivalence.

Stephen Michell 2016-3-7 11:45 AM
**Deleted:** 0

Stephen Michell 2016-3-7 11:45 AM
**Deleted:** 0

Stephen Michell 2016-3-7 11:45 AM
**Deleted:** 0

Stephen Michell 2016-3-7 11:45 AM
**Deleted:** 1

Stephen Michell 2016-3-7 11:46 AM
**Deleted:** 1

Stephen Michell 2017-3-9 2:50 PM
**Deleted:** see

Stephen Michell 2017-3-9 2:50 PM
**Deleted:** 5

Stephen Michell 2017-3-9 2:50 PM
**Deleted:** ( *Error! Reference source not found.)*

Stephen Michell 2016-3-7 11:46 AM
**Deleted:** 1

- Specify the save attribute when supplying an initial value.
- Use implicit none to require explicit declarations.

### 6.55 Unspecified Behaviour [BQF]

Stephen Michell 2016-3-7 11:46 AM
**Deleted:** 2

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

### 6.56 Undefined Behaviour [EWF]

Stephen Michell 2016-3-7 11:46 AM
**Deleted:** 3

#### 6.56.1 Applicability to language

Stephen Michell 2016-3-7 11:46 AM
**Deleted:** 3

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 of these might be invoked instead of an intended external procedure with the same name.

#### 6.56.2 Guidance to language users

Stephen Michell 2016-3-7 11:46 AM
**Deleted:** 3

- 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.
- Supply an explicit interface to specify the `external` attribute for all external procedures invoked.
- Avoid use of non-standard intrinsic procedures.
- Specific the `intrinsic` attribute for all non-standard intrinsic procedures.

### 6.57 Implementation-Defined Behaviour [FAB]

Stephen Michell 2016-3-7 11:46 AM
**Deleted:** 4

#### 6.57.1 Applicability to language

Stephen Michell 2016-3-7 11:46 AM
**Deleted:** 4

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.

#### 6.57.2 Guidance to language users

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 4

- 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.
- Supply an explicit interface to specify the `external` attribute for all external procedures invoked.
- Avoid use of non-standard intrinsic procedures.
- Specific the `intrinsic` attribute for all non-standard intrinsic procedures.

## 6.58 Deprecated Language Features [MEM]

### 6.58.1 Applicability to language

Because they are still used in some programs, many processors 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 might 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.

### 6.58.2 Guidance to language users

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

## 6.59 Concurrency – Activation [CGA]

TBD

### 6.59.1 Applicability to language

TBD

### 6.59.2 Guidance to language users

TBD

## 6.60 Concurrency – Directed termination [CGT]

**TBD**

### 6.60.1 Applicability to language

TBD

### 6.60.2 Guidance to language users

## 6.61 Concurrent Data Access [CGX]

### 6.61.1 Applicability to language

TBD

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 5

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 5

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 5

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 56

Stephen Michell 2017-3-7 12:41 PM
**Formatted:** Normal

Stephen Michell 2017-3-7 12:41 PM
**Deleted:**

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 56

Stephen Michell 2017-3-7 12:41 PM
**Formatted:** Normal

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 56

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 57

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 57

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 57

Stephen Michell 2016-3-7 11:47 AM
**Deleted:** 58

Stephen Michell 2017-3-9 2:58 PM
**Moved down [2]:** TBD

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 58

Stephen Michell 2017-3-9 2:58 PM
**Moved (insertion) [2]**

### 6.61.2 Guidance to language users

TBD

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 58

### 6.62 Concurrency – Premature Termination [CGS]

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 59

### 6.62.1 Applicability to language

Stephen Michell 2017-3-9 2:58 PM
**Deleted:** TBD

TBD

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 59

### 6.62.2 Guidance to language users

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 59

TBD

### 6.63 Protocol Lock Errors [CGM]

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 60

### 6.63.1 Applicability to language

Stephen Michell 2017-3-9 2:58 PM
**Deleted:** TBD

TBD

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 0

### 6.63.2 Guidance to language users

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 0

TBD

### 6.64 Uncontrolled Format String [SHL]

Stephen Michell 2016-3-7 11:48 AM
**Deleted:** 1

TBD

## 7 Language specific vulnerabilities for Fortran

## 8 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 or the `target` attribute 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.

# Bibliography


[1] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2004

[2] ISO/IEC TR 10000-1, *Information technology — Framework and taxonomy of International Standardized Profiles — Part 1: General principles and documentation framework*

[3] ISO 10241 (all parts), *International terminology standards*

[7] ISO/IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

[9] ISO/IEC 8652:1995, *Information technology — Programming languages* — Ada

[11] R. Seacord, *The CERT C Secure Coding Standard*. Boston,MA: Addison-Westley, 2008.

[14] ISO/IEC TR 15942:2000, *Information technology — Programming languages — Guide for the use of the Ada programming language in high integrity systems*

[17] ISO/IEC TR 24718: 2005, *Information technology — Programming languages — Guide for the use of the Ada Ravenscar Profile in high integrity systems*

[19] ISO/IEC 15291:1999, *Information technology — Programming languages — Ada Semantic Interface Specification (ASIS)*

[20] Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B).December 1992.

[21] IEC 61508: Parts 1-7, Functional safety: safety-related systems. 1998. (Part 3 is concerned with software).

[22] ISO/IEC 15408: 1999 Information technology. Security techniques. Evaluation criteria for IT security.

[23] J Barnes, High Integrity Software - the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

1. Lecture Notes on Computer Science 5020, "Ada 2012 Rationale: The Language, the Standard Libraries," John Barnes, Springer, 2012. ???????

[25] Steve Christy, *Vulnerability Type Distributions in CVE*, V1.0, 2006/10/04

[29] Lions, J. L. ARIANE 5 Flight 501 Failure Report. Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[33] The Common Weakness Enumeration (CWE) Initiative, MITRE Corporation, (http://cwe.mitre.org/)

[34] Goldberg, David, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, vol 23, issue 1 (March 1991), ISSN 0360-0300, pp 5-48.

[35] IEEE Standards Committee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008. Institute of Electrical and Electronics Engineers, New York, 2008.

[36] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008

[37] Bo Einarsson, ed. Accuracy and Reliability in Scientific Computing, SIAM, July 2005 http://www.nsc.liu.se/wg25/book

[38] GAO Report, Patriot *Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, B-247094, Feb. 4, 1992, http://archive.gao.gov/t2pbat6/145960.pdf

[39] Robert Skeel, *Roundoff Error Cripples Patriot Missile*, SIAM News, Volume 25, Number 4, July 1992, page 11, http://www.siam.org/siamnews/general/patriot.htm

[41] Holzmann, Garard J., Computer, vol. 39, no. 6, pp 95-97, Jun., 2006, *The Power of 10: Rules for Developing Safety-Critical Code*

[42] P. V. Bhansali, A systematic approach to identifying a safe subset for safety-critical software, ACM SIGSOFT Software Engineering Notes, v.28 n.4, July 2003

[43] Ada 95 Quality and Style Guide, SPC-91061-CMC, version 02.01.01. Herndon, Virginia: Software Productivity Consortium, 1992. Available from: http://www.adaic.org/docs/95style/95style.pdf

[44] Ghassan, A., & Alkadi, I. (2003). Application of a Revised DIT Metric to Redesign an OO Design. *Journal of Object Technology* , 127-134.

[45] Subramanian, S., Tsai, W.-T., & Rayadurgam, S. (1998). Design Constraint Violation Detection in Safety-Critical Systems. The 3rd IEEE International Symposium on High-Assurance Systems Engineering , 109 - 116.

[46] Lundqvist, K and Asplund, L., "*A Formal Model of a Run-Time Kernel for Ravenscar*", The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999

# Index


Ada, 13, 59, 63, 73, 76
AMV – Type-breaking Reinterpretation of Data, 72
*API*
  Application Programming Interface, 16
APL, 48
Apple
  OS X, 120
*application vulnerabilities*, 9
Application Vulnerabilities
  Adherence to Least Privilege [XYN], 113
  Authentication Logic Error [XZO], 135
  Cross-site Scripting [XYT], 125
  Discrepancy Information Leak [XZL], 129
  Distinguished Values in Data Types [KLK], 112
  Download of Code Without Integrity Check [DLB], 137
  Executing or Loading Untrusted Code [XYS], 116
  Hard-coded Password [XYP], 136
  Improper Restriction of Excessive Authentication Attempts [WPL], 140
  Improperly Verified Signature [XZR], 128
  Inclusion of Functionality from Untrusted Control Sphere [DHU], 139
  Incorrect Authorization [BJE], 138
  Injection [RST], 122
  Insufficiently Protected Credentials [XYM], 133
  Memory Locking [XZX], 117
  Missing or Inconsistent Access Control [XZN], 134
  Missing Required Cryptographic Step [XZS], 133
  Path Traversal [EWR], 130
  Privilege Sandbox Issues [XYO], 114
  Resource Exhaustion [XZP], 118
  Resource Names [HTS], 120
  Sensitive Information Uncleared Before Use [XZK], 130
  Unquoted Search Path or Element [XZQ], 127
  Unrestricted File Upload [CBF], 119
  Unspecified Functionality [BVQ], 111
  URL Redirection to Untrusted Site ('Open Redirect') [PYQ], 140
  Use of a One-Way Hash without a Salt [MVX], 141
application vulnerability, 5
Ariane 5, 21

bitwise operators, 48
BJE – Incorrect Authorization, 138
BJL – Namespace Issues, 43
*black-list*, 120, 124
BQF – Unspecified Behaviour, 92, 94, 95
`break`, 60
BRS – Obscure Language Features, 91
buffer boundary violation, 23
buffer overflow, 23, 26
buffer underwrite, 23
BVQ – Unspecified Functionality, 111

C, 22, 48, 50, 51, 58, 60, 63, 73
C++, 48, 51, 58, 63, 73, 76, 86
C11, 192
*call by copy*, 61
*call by name*, 61
*call by reference*, 61
*call by result*, 61
*call by value*, 61
*call by value-result*, 61
CBF – Unrestricted File Upload, 119
CCB – Enumerator Issues, 18
CGA – Concurrency – Activation, 98
CGM – Protocol Lock Errors, 105
CGS – Concurrency – Premature Termination, 103
CGT - Concurrency – Directed termination, 100
CGX – Concurrent Data Access, 101
CGY – Inadequately Secure Communication of Shared Resources, 107
CJM – String Termination, 22
CLL – Switch Statements and Static Analysis, 54
concurrency, 2
`continue`, 60
cryptologic, 71, 128
CSJ – Passing Parameters and Return Values, 61, 82

dangling reference, 31
DCM – Dangling References to Stack Frames, 63
Deactivated code, 53
Dead code, 53
*deadlock*, 106
DHU – Inclusion of Functionality from Untrusted Control Sphere, 139
Diffie-Hellman-style, 136
digital signature, 84
DJS – Inter-language Calling, 81
DLB – Download of Code Without Integrity Check, 137
*DoS*
  Denial of Service, 118
dynamically linked, 83

EFS – Use of unchecked data from an uncontrolled or tainted source, 109
encryption, 128, 133
endian
big, 15
little, 15
endianness, 14
Enumerations, 18
EOJ – Demarcation of Control Flow, 56
EWD – Structured Programming, 60
*EWF – Undefined Behaviour*, 92, 94, 95
*EWR – Path Traversal*, 124, 130
exception handler, 86

*FAB – Implementation-defined Behaviour*, 92, 94, 95
FIF – Arithmetic Wrap-around Error, 34, 35
FLC – Numeric Conversion Errors, 20
Fortran, 73

GDL – Recursion, 67
generics, 76
GIF, 120
`goto`, 60

HCB – Buffer Boundary Violation (Buffer Overflow), 23, 82
HFC – Pointer Casting and Pointer Type Changes, 28
HJW – Unanticipated Exceptions from Library Routines, 86
*HTML*
Hyper Text Markup Language, 124
HTS – Resource Names, 120
*HTTP*
Hypertext Transfer Protocol, 127

IEC 60559, 16
IEEE 754, 16
IHN –Type System, 12
inheritance, 78
IP address, 119

Java, 18, 50, 52, 76
JavaScript, 125, 126, 127
JCW – Operator Precedence/Order of Evaluation, 47

KLK – Distinguished Values in Data Types, 112
KOA – Likely Incorrect Expression, 50

*language vulnerabilities*, 9
*Language Vulnerabilities*
Argument Passing to Library Functions [TRJ], 80
Arithmetic Wrap-around Error [FIF], 34
Bit Representations [STR], 14
Buffer Boundary Violation (Buffer Overflow) [HCB], 23
Choice of Clear Names [NAI], 37
Concurrency – Activation [CGA], 98
Concurrency – Directed termination [CGT], 100
Concurrency – Premature Termination [CGS], 103
Concurrent Data Access [CGX], 101
Dangling Reference to Heap [XYK], 31
Dangling References to Stack Frames [DCM], 63
Dead and Deactivated Code [XYQ], 52
Dead Store [WXQ], 39
Demarcation of Control Flow [EOJ], 56
Deprecated Language Features [MEM], 97
Dynamically-linked Code and Self-modifying Code [NYY], 83
Enumerator Issues [CCB], 18
Extra Intrinsics [LRM], 79
*Floating-point Arithmetic [PLF]*, xvii, 16
Identifier Name Reuse [YOW], 41
Ignored Error Status and Unhandled Exceptions [OYB], 68
Implementation-defined Behaviour [FAB], 95
Inadequately Secure Communication of Shared Resources [CGY], 107
Inheritance [RIP], 78
Initialization of Variables [LAV], 45
Inter-language Calling [DJS], 81
Library Signature [NSQ], 84
Likely Incorrect Expression [KOA], 50
Loop Control Variables [TEX], 57
Memory Leak [XYL], 74
Namespace Issues [BJL], 43
Null Pointer Dereference [XYH], 30
Numeric Conversion Errors [FLC], 20
Obscure Language Features [BRS], 91
Off-by-one Error [XZH], 58
Operator Precedence/Order of Evaluation [JCW], 47
Passing Parameters and Return Values [CSJ], 61, 82
Pointer Arithmetic [RVG], 29
Pointer Casting and Pointer Type Changes [HFC], 28
Pre-processor Directives [NMP], 87
Protocol Lock Errors [CGM], 105
Provision of Inherently Unsafe Operations [SKL], 90
Recursion [GDL], 67
Side-effects and Order of Evaluation [SAM], 49
Sign Extension Error [XZI], 36
String Termination [CJM], 22
Structured Programming [EWD], 60
Subprogram Signature Mismatch [OTR], 65
Suppression of Language-defined Run-time Checking [MXB], 89

Switch Statements and Static Analysis [CLL], 54
Templates and Generics [SYM], 76
Termination Strategy [REU], 70
Type System [IHN], 12
Type-breaking Reinterpretation of Data [AMV], 72
Unanticipated Exceptions from Library Routines [HJW], 86
Unchecked Array Copying [XYW], 27
Unchecked Array Indexing [XYZ], 25
Uncontrolled Fromat String [SHL], 110
Undefined Behaviour [EWF], 94
Unspecified Behaviour [BFQ], 92
Unused Variable [YZS], 40
Use of unchecked data from an uncontrolled or tainted source [EFS], 109
Using Shift Operations for Multiplication and Division [PIK], 35
language vulnerability, 5
LAV – Initialization of Variables, 45
LHS (left-hand side), 241
Linux, 120
*livelock*, 106
`longjmp`, 60
LRM – Extra Intrinsics, 79

MAC address, 119
macof, 118
MEM – Deprecated Language Features, 97
memory disclosure, 130
Microsoft
Win16, 121
Windows, 117
Windows XP, 120
*MIME*
Multipurpose Internet Mail Extensions, 124
MISRA C, 29
MISRA C++, 87
`mlock()`, 117
MVX – Use of a One-Way Hash without a Salt, 141
MXB – Suppression of Language-defined Run-time Checking, 89

NAI – Choice of Clear Names, 37
*name type equivalence*, 12
NMP – Pre-Processor Directives, 87
NSQ – Library Signature, 84
*NTFS*
New Technology File System, 120
`NULL`, 31, 58
`NULL pointer`, 31
null-pointer, 30

NYY – Dynamically-linked Code and Self-modifying Code, 83

OTR – Subprogram Signature Mismatch, 65, 82
OYB – Ignored Error Status and Unhandled Exceptions, 68, 163

Pascal, 82
PHP, 124
*PIK – Using Shift Operations for Multiplication and Division*, 34, 35, 197
*PLF – Floating-point Arithmetic*, xvii, 16
POSIX, 99
`pragmas`, 75, 96
predictable execution, 4, 8
PYQ – URL Redirection to Untrusted Site ('Open Redirect'), 140

real numbers, 16
Real-Time Java, 105
resource exhaustion, 118
REU – Termination Strategy, 70
*RIP – Inheritance*, xvii, 78
`rsize_t`, 22
RST – Injection, 109, 122
*runtime-constraint handler*, 191
RVG – Pointer Arithmetic, 29

safety hazard, 4
safety-critical software, 5
SAM – Side-effects and Order of Evaluation, 49
security vulnerability, 5
SeImpersonatePrivilege, 115
`setjmp`, 60
SHL – Uncontrolled Format String, 110
`size_t`, 22
SKL – Provision of Inherently Unsafe Operations, 90
software quality, 4
*software vulnerabilities*, 9
*SQL*
Structured Query Language, 112
STR – Bit Representations, 14
`strcpy`, 23
`strncpy`, 23
*structure type equivalence*, 12
`switch`, 54
SYM – Templates and Generics, 76
symlink, 131

*tail-recursion*, 68
templates, 76, 77
TEX – Loop Control Variables, 57
**thread**, 2

TRJ – Argument Passing to Library Functions, 80
*type casts*, 20
*type coercion*, 20
*type safe*, 12
*type secure*, 12
*type system*, 12

UNC
Uniform Naming Convention, 131
Universal Naming Convention, 131
`Unchecked_Conversion`, 73
UNIX, 83, 114, 120, 131
unspecified functionality, 111
*Unspecified functionality*, 111
*URI*
Uniform Resource Identifier, 127
URL
Uniform Resource Locator, 127

`VirtualLock()`, 117

*white-list*, 120, 124, 127
Windows, 99
WPL – Improper Restriction of Excessive Authentication Attempts, 140
WXQ – Dead Store, 39, 40, 41

XSS
Cross-site scripting, 125
XYH – Null Pointer Deference, 30
XYK – Dangling Reference to Heap, 31
XYL – Memory Leak, 74
*XYM – Insufficiently Protected Credentials*, 9, 133
XYN –Adherence to Least Privilege, 113
XYO – Privilege Sandbox Issues, 114
XYP – Hard-coded Password, 136
XYQ – Dead and Deactivated Code, 52
XYS – Executing or Loading Untrusted Code, 116
XYT – Cross-site Scripting, 125
XYW – Unchecked Array Copying, 27
XYZ – Unchecked Array Indexing, 25, 28
XZH – Off-by-one Error, 58
XZI – Sign Extension Error, 36
XZK – Senitive Information Uncleared Before Use, 130
XZL – Discrepancy Information Leak, 129
XZN – Missing or Inconsistent Access Control, 134
XZO – Authentication Logic Error, 135
XZP – Resource Exhaustion, 118
XZQ – Unquoted Search Path or Element, 127
XZR – Improperly Verified Signature, 128
XZS – Missing Required Cryptographic Step, 133
XZX – Memory Locking, 117

YOW – Identifier Name Reuse, 41, 44
*YZS – Unused Variable*, 39, 40