ISO/IEC JTC 1/SC 22 N0833

Date: 2018-09-07

ISO/IEC TR 24772-2

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 5: Vulnerability descriptions for the programming language Ruby

Document type: International standard

Document subtype: if applicable

Document stage: (10) development stage

Document language: E

*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.

**Copyright notice**

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

*ISO copyright office*

*Case postale 56, CH-1211 Geneva 20*

*Tel. + 41 22 749 01 11*

*Fax + 41 22 749 09 47*

*E-mail copyright@iso.org*

*Web www.iso.org*

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

Contents

[Foreword 8](#_Toc519526879)

[Introduction 9](#_Toc519526880)

[1. Scope 10](#_Toc519526881)

[2. Normative references 10](#_Toc519526882)

[3. Terms and definitions, symbols and conventions 10](#_Toc519526883)

[3.1 Terms and definitions 10](#_Toc519526884)

[4 Language concepts 14](#_Toc519526885)

[5 General guidance for Ada 18](#_Toc519526886)

[5.1 Ada Language Design 18](#_Toc519526887)

[6 Specific Guidance for Ada 19](#_Toc519526888)

[6.1 General 19](#_Toc519526889)

[6.2 Type System [IHN] 19](#_Toc519526890)

[6.2.1 Applicability to language 19](#_Toc519526891)

[6.2.2 Guidance to language users 20](#_Toc519526892)

[6.3 Bit Representation [STR] 20](#_Toc519526893)

[6.3.1 Applicability to language 20](#_Toc519526894)

[6.3.2 Guidance to language users 20](#_Toc519526895)

[6.4 Floating-point Arithmetic [PLF] 21](#_Toc519526896)

[6.4.1 Applicability to language 21](#_Toc519526897)

[6.4.2 Guidance to language users 21](#_Toc519526898)

[6.5 Enumerator Issues [CCB] 21](#_Toc519526899)

[6.5.1 Applicability to language 21](#_Toc519526900)

[6.5.2 Guidance to language users 22](#_Toc519526901)

[6.6 Conversion Errors [FLC] 22](#_Toc519526902)

[6.6.1 Applicability to language 22](#_Toc519526903)

[6.6.2 Guidance to language users 22](#_Toc519526904)

[6.7 String Termination [CJM] 23](#_Toc519526905)

[6.8 Buffer Boundary Violation (Buffer Overflow) [HCB] 23](#_Toc519526906)

[6.9 Unchecked Array Indexing [XYZ] 23](#_Toc519526907)

[6.9.1 Applicability to language 23](#_Toc519526908)

[6.9.2 Guidance to language users 23](#_Toc519526909)

[6.10 Unchecked Array Copying [XYW] 23](#_Toc519526910)

[6.11 Pointer Type Conversions [HFC] 24](#_Toc519526911)

[6.11.1 Applicability to language 24](#_Toc519526912)

[6.11.2 Guidance to language users 24](#_Toc519526913)

[6.12 Pointer Arithmetic [RVG] 24](#_Toc519526914)

[6.13 Null Pointer Dereference [XYH] 24](#_Toc519526915)

[6.13.1 Applicability to the language 24](#_Toc519526916)

[6.13.2 Guidance to language users 24](#_Toc519526917)

[6.14 Dangling Reference to Heap [XYK] 25](#_Toc519526918)

[6.14.1 Applicability to language 25](#_Toc519526919)

[6.14.2 Guidance to language users 25](#_Toc519526920)

[6.15 Arithmetic Wrap-around Error [FIF] 25](#_Toc519526921)

[6.16 Using Shift Operations for Multiplication and Division [PIK] 25](#_Toc519526922)

[6.17 Choice of Clear Names [NAI] 26](#_Toc519526923)

[6.17.1 Applicability to language 26](#_Toc519526924)

[6.17.2 Guidance to language users 26](#_Toc519526925)

[6.18 Dead store [WXQ] 27](#_Toc519526926)

[6.18.1 Applicability to language 27](#_Toc519526927)

[6.18.2 Guidance to Language Users 27](#_Toc519526928)

[6.19 Unused Variable [YZS] 27](#_Toc519526929)

[6.19.1 Applicability to language 27](#_Toc519526930)

[6.19.2 Guidance to language users 27](#_Toc519526931)

[6.20 Identifier Name Reuse [YOW] 27](#_Toc519526932)

[6.20.1 Applicability to language 27](#_Toc519526933)

[6.20.2 Guidance to language users 28](#_Toc519526934)

[6.21 Namespace Issues [BJL] 28](#_Toc519526935)

[6.22 Initialization of Variables [LAV] 28](#_Toc519526936)

[6.22.1 Applicability to language 28](#_Toc519526937)

[6.22.2 Guidance to language users 29](#_Toc519526938)

[6.23 Operator Precedence/Order of Evaluation [JCW] 29](#_Toc519526939)

[6.23.1 Applicability to language 29](#_Toc519526940)

[6.23.2 Guidance to language users 29](#_Toc519526941)

[6.24 Side-effects and Order of Evaluation [SAM] 30](#_Toc519526942)

[6.24.1 Applicability to language 30](#_Toc519526943)

[6.24.2 Guidance to language users 30](#_Toc519526944)

[6.25 Likely Incorrect Expression [KOA] 30](#_Toc519526945)

[6.25.1 Applicability to language 30](#_Toc519526946)

[6.25.2 Guidance to language users 31](#_Toc519526947)

[6.26 Dead and Deactivated Code [XYQ] 31](#_Toc519526948)

[6.26.1 Applicability to language 31](#_Toc519526949)

[6.26.2 Guidance to language users 31](#_Toc519526950)

[6.27 Switch Statements and Static Analysis [CLL] 32](#_Toc519526951)

[6.27.1 Applicability to language 32](#_Toc519526952)

[6.27.2 Guidance to language users 32](#_Toc519526953)

[6.28 Demarcation of Control Flow [EOJ] 32](#_Toc519526954)

[6.29 Loop Control Variables [TEX] 33](#_Toc519526955)

[6.30 Off-by-one Error [XZH] 33](#_Toc519526956)

[6.30.1 Applicability to language 33](#_Toc519526957)

[6.30.2 Guidance to language users 33](#_Toc519526958)

[6.31 Structured Programming [EWD] 34](#_Toc519526959)

[6.31.1 Applicability to language 34](#_Toc519526960)

[6.31.2 Guidance to language users 34](#_Toc519526961)

[6.32 Passing Parameters and Return Values [CSJ] 34](#_Toc519526962)

[6.32.1 Applicability to language 34](#_Toc519526963)

[6.32.2 Guidance to language users 34](#_Toc519526964)

[6.33 Dangling References to Stack Frames [DCM] 34](#_Toc519526965)

[6.33.1 Applicability to language 34](#_Toc519526966)

[6.33.2 Guidance to language users 35](#_Toc519526967)

[6.34 Subprogram Signature Mismatch [OTR] 35](#_Toc519526968)

[6.34.1 Applicability to language 35](#_Toc519526969)

[6.34.2 Guidance to language users 36](#_Toc519526970)

[6.35 Recursion [GDL] 36](#_Toc519526971)

[6.35.1 Applicability to language 36](#_Toc519526972)

[6.35.2 Guidance to language users 36](#_Toc519526973)

[6.36 Ignored Error Status and Unhandled Exceptions [OYB] 36](#_Toc519526974)

[6.36.1 Applicability to language 36](#_Toc519526975)

[6.36.2 Guidance to language users 36](#_Toc519526976)

[6.37 Type-breaking Reinterpretation of Data [AMV] 37](#_Toc519526977)

[6.37.1 Applicability to language 37](#_Toc519526978)

[6.37.2 Guidance to language users 37](#_Toc519526979)

[6.38 Deep vs. Shallow Copying [YAN] 37](#_Toc519526980)

[6.38.1 Applicability to language 37](#_Toc519526981)

[6.38.2 Guidance to language users 37](#_Toc519526982)

[6.39 Memory Leak and Heap Fragmentation [XYL] 38](#_Toc519526983)

[6.39.1 Applicability to language 38](#_Toc519526984)

[6.39.2 Guidance to language users 38](#_Toc519526985)

[6.40 Templates and Generics [SYM] 38](#_Toc519526986)

[6.41 Inheritance [RIP] 39](#_Toc519526987)

[6.41.1 Applicability to language 39](#_Toc519526988)

[6.41.2 Guidance to language users 39](#_Toc519526989)

[6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP] 39](#_Toc519526990)

[6.42.1 Applicability to language 39](#_Toc519526991)

[6.42.2 Guidance to Language Users 40](#_Toc519526992)

[6.43 Redispatching [PPH] 40](#_Toc519526993)

[6.43.1 Applicability to language 40](#_Toc519526994)

[6.43.2 Guidance to Language Users 40](#_Toc519526995)

[6.44 Polymorphic variables [BKK] 40](#_Toc519526996)

[6.44.1 Applicability to language 40](#_Toc519526997)

[6.44.2 Guidance to Language Users 41](#_Toc519526998)

[6.45 Extra Intrinsics [LRM] 41](#_Toc519526999)

[6.46 Argument Passing to Library Functions [TRJ] 41](#_Toc519527000)

[6.46.1 Applicability to language 41](#_Toc519527001)

[6.46.2 Guidance to language users 41](#_Toc519527002)

[6.47 Inter-language Calling [DJS] 42](#_Toc519527003)

[6.47.1 Applicability to Language 42](#_Toc519527004)

[6.47.2 Guidance to Language Users 42](#_Toc519527005)

[6.48 Dynamically-linked Code and Self-modifying Code [NYY] 42](#_Toc519527006)

[6.49 Library Signature [NSQ] 42](#_Toc519527007)

[6.49.1 Applicability to language 42](#_Toc519527008)

[6.49.2 Guidance to language users 43](#_Toc519527009)

[6.50 Unanticipated Exceptions from Library Routines [HJW] 43](#_Toc519527010)

[6.50.1 Applicability to language 43](#_Toc519527011)

[6.50.2 Guidance to language users 43](#_Toc519527012)

[6.51 Pre-Processor Directives [NMP] 43](#_Toc519527013)

[6.52 Suppression of Language-defined Run-time Checking [MXB] 44](#_Toc519527014)

[6.52.1 Applicability to Language 44](#_Toc519527015)

[6.52.2 Guidance to Language Users 44](#_Toc519527016)

[6.53 Provision of Inherently Unsafe Operations [SKL] 44](#_Toc519527017)

[6.53.1 Applicability to Language 44](#_Toc519527018)

[6.53.2 Guidance to language users 44](#_Toc519527019)

[6.54 Obscure Language Features [BRS] 44](#_Toc519527020)

[6.54.1 Applicability to language 44](#_Toc519527021)

[6.54.2 Guidance to language users 45](#_Toc519527022)

[6.55 Unspecified Behaviour [BQF] 45](#_Toc519527023)

[6.55.1 Applicability to language 45](#_Toc519527024)

[6.55.2 Guidance to language users 45](#_Toc519527025)

[6.56 Undefined Behaviour [EWF] 46](#_Toc519527026)

[6.56.1 Applicability to language 46](#_Toc519527027)

[6.56.2 Guidance to language users 46](#_Toc519527028)

[6.57 Implementation-Defined Behaviour [FAB] 47](#_Toc519527029)

[6.57.1 Applicability to language 47](#_Toc519527030)

[6.57.2 Guidance to language users 47](#_Toc519527031)

[6.58 Deprecated Language Features [MEM] 48](#_Toc519527032)

[6.58.1 Applicability to language 48](#_Toc519527033)

[6.58.2 Guidance to language users 48](#_Toc519527034)

[6.59 Concurrency – Activation [CGA] 48](#_Toc519527035)

[6.59.1 Applicability to language 48](#_Toc519527036)

[6.59.2 Guidance to language users 48](#_Toc519527037)

[6.60 Concurrency – Directed termination [CGT] 49](#_Toc519527038)

[6.60.1 Applicability to language 49](#_Toc519527039)

[6.60.2 Guidance to language users 49](#_Toc519527040)

[6.61 Concurrent Data Access [CGX] 49](#_Toc519527041)

[6.61.1 Applicability to language 49](#_Toc519527042)

[6.61.2 Guidance to language users 49](#_Toc519527043)

[6.62 Concurrency – Premature Termination [CGS] 50](#_Toc519527044)

[6.62.1 Applicability to language 50](#_Toc519527045)

[6.62.2 Guidance to language users 50](#_Toc519527046)

[6.63 Protocol Lock Errors [CGM] 50](#_Toc519527047)

[6.63.1 Applicability to language 50](#_Toc519527048)

[6.63.2 Guidance to language users 50](#_Toc519527049)

[6.64 Reliance on external format strings [SHL] 51](#_Toc519527050)

[7 Language specific vulnerabilities for Ada 51](#_Toc519527051)

[8 Implications for standardization 51](#_Toc519527052)

[Bibliography 52](#_Toc519527053)

[Index 54](#_Toc519527054)

# 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 24775, 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 Ada so that application developers considering Ruby or using Ruby will be better able to avoid the programming constructs that lead to vulnerabilities in software written in the Ruby 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 Report can also be used in comparison with companion technical reports and with the language-independent report, ISO/IEC TR 24772-1, *Information Technology – Programming Languages— Guidance to avoiding vulnerabilities in programming languages,* to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This Technical Report 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 – Part 5: Vulnerability descriptions for the programming language Ruby**

# 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 document TR 24772-1 are manifested in Ruby.

# 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 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 TR 24772-1, *Information Technology – Programming Languages— Guidance to avoiding vulnerabilities in programming languages*

[ISO/IEC TR 15942:2000](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=29575), *Guidance for the Use of the Ada programming language in high integrity systems*

[ISO/IEC TR 24718:2005](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38828), *Guide for the use of the Ada Ravenscar Profile in high integrity systems*

[IEEE 754-2008, Binary Floating Point Arithmetic](http://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf), IEEE, 2008

[ANSI/IEEE 854-1987, Radix-Independent Floating-Point Arithmetic](http://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf), 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.

3.1.1

**block**a procedure which is passed to a method invocation

3.1.2

**class**an object which defines the behaviour of a set of other objects called its instances. class variable: A variable whose value is shared by all the instances of a class.

3.1.3

**constant**  
a variable which is defined in a class or a module and is accessible both inside and outside the class or module. Note: The value of a constant is ordinarily expected to remain unchanged during the execution of a program, but IPA Ruby Standardization Draft does not force it.

3.1.4

**exception**   
an object which represents an exceptional event.

3.1.5

**global variable**  
a variable which is accessible everywhere in a program.

3.1.6

**implementation-defined**   
possibly differing between implementations, but defined for every implementation.

3.1.7

**instance method**   
a method which can be invoked on all the instances of a class.

3.1.8

**instance variable**   
a variable that exists in a set of variable bindings which every object has.

3.1.9

**local variable**   
a variable which is accessible only in a certain scope introduced by a program construct such as a method definition, a block, a class definition, a module definition, a singleton class definition, or the top level of a program.

3.1.10

**method**   
a procedure which, when invoked on an object, performs a set of computations on the object.  
method visibility: An attribute of a method which determines the conditions under which a method invocation is allowed.

3.1.11

**module**   
an object which provides features to be included into a class or another module.

3.1.12

**objec**t   
a computational entity which has states and behaviour. The behaviour of an object is a set of methods which can be invoked on the object.

3.1.13

**singleton class**   
an object which can modify the behaviour of its associated object. singleton method: An instance method of a singleton class.

# 4 Language concepts

# 5 General guidance for Ruby

## 5.1 Ruby Language Design

**5.2 Top Avoidance Mechanisms**

The recommendations of this subclause are restatements of recommendations from clause 6 that have been identified as the most frequent or noteworthy recommendations from clause 6. Table 5.1 identifies the most relevant avoidance mechanisms to be used to prevent vulnerabilities in Ruby.

In addition to the generic programming rules from TR 24772-1 clause 5.4, additional rules from this subclause apply specifically to the Ruby programming language. Clause 6 of this document provides guidance to mitigate against known vulnerabilities in Ruby.

|  |  |  |
| --- | --- | --- |
| **Index** | **Avoidance Mechanism** | **Reference** |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |
| 8 |  |  |
| 9 |  |  |
| 10 |  |  |
| 11 |  |  |
| 12 |  |  |
| 13 |  |  |
| 14 |  |  |

#### Table 5-1 Most relevant avoidance mechanisms to be used to prevent vulnerabilities

# 6 Specific Guidance for Ruby

## 6.1 General

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

## 6.2 Type System [IHN]

### 6.2.1 Applicability to language

Ruby employs a dynamic type system usually referred to as “duck typing”. In this system the class or type of an object is less important than the interface, or methods, it defines. Two different classes may respond to the same methods, which mean instances of each class will handle the same method call. Usually an object is not implicitly changed into another type.

Automatic conversion occurs for some built-in types in certain situations. For example with the addition of a float and an integer, the integer will be converted automatically to a float. In the examples below, the result of an operation is indicated by a Ruby comment starting with =>.

a=2  
b = 2.0  
a + b #=> 4.0

Another instance of automatic conversion is when an integer becomes too large to fit within a machine word. On a 32-bit machine Ruby Fixnums have the range -230 to 230-1. When an integer becomes such that it no longer fits within said range it is converted to a Bignum. Bignums are arbitrary length integers bounded only by memory limitations.

Explicit conversion methods exist in Ruby to convert between types. The integer class contains the methods to\_s and to\_f which return the integer represented as a string object and float object, respectively.

10.to\_s #=> “10” 10.to\_f #=> 10.0

Strings likewise support conversion to integer and float objects.

“5”.to\_i #=> 5 “5”.to\_f #=> 5.0

Duck typing grants programmers of Ruby great flexibility. Strict typing is not imposed by the language, but if a programmer chooses, he or she can write programs such that methods mandate the class of the objects on which they operate. This is discouraged in Ruby. If an object is called with a method it does not know, an exception will be raised.

### 6.2.2 Guidance to language users

## 6.3 Bit Representation [STR]

### 6.3.1 Applicability to language

Ruby abstracts internal storage of integers. Users do not need to concern themselves about the size (in bits) of an integer. Since integers grow as needed the user does not need to worry about overflow. Ruby provides a mechanism to inspect specific bits of an integer through the [] method. For example to read the 10th bit of a number:

number = 42 number[10] #=> 0 number = 1024 number[10] #=> 1

Note that the bits returned are not required to correspond to the internal representation of the number, just that it returns a consistent representation of the number in that implementation.

Ruby supports a variety of bitwise operators. These include ~ (not), & (and), | (or), ^ (exclusive or), << (shift left), and >> (shift right). Each of these operators works with integers of any size.

Ruby offers a pack method for the Array class (Array#pack) which produces a binary sequence dictated by the user supplied template. In this way members of an array can be converted to different bit representations. For instance an option for numbers is to store them in one of three ways: native-endian, big-endian, and little-endian. In this way bit sequences can be constructed for a particular interaction or purpose. There is a similar unpack method which will extract data given a template and bit sequence.

### 6.3.2 Guidance to language users

* For values created within Ruby the user need not concern themselves with the internal representation of data. In most situations using specific binary representations makes code harder to read and understand.
* Network packets that go on the wire are one case where bit representation is important. In situations like this be sure to use the Array#pack to produce network endian data15.
* Binary files are another situation where bit representation matters. The file format description should indicate big-endian or little-endian preference.

## 6.4 Floating-point Arithmetic [PLF]

### 6.4.1 Applicability to language

Ruby supports the use of floating-point arithmetic with the Float class. The precision of floats in Ruby is implementation defined, however if the underlying system supports IEC 60559, the representation of floats shall be the 64-bit double format as specified in IEC 60559, 3.2.2.

Floating-point numbers are usually approximations of real numbers and as such some precision is lost. This is problematic when performing repeated operations. For example adding small values to numbers sometimes results in accumulation errors. Testing numbers for equality is sometimes unreliable as well. For this reason floating-point numbers should not be used to terminate loops.

### 6.4.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.4.5 of TR 24772-1..

## 6.5 Enumerator Issues [CCB]

### 6.5.1 Applicability to language

Ruby does not provide enumerations. Instead provides a facility for named symbols. These symbols are unique representations with no value associated. In Ruby, symbols are lightweight objects which need not be defined ahead of time. For example,

travel(:north)  
is a valid use of the symbol :north. (Ruby’s literal syntax for symbols is a colon followed by a word.) There is no

danger of accidentally getting to the “value” of an enumeration. So this:

travel(:north + :south)

is not allowed. Symbols do not support addition, or any method which alters the symbol.

Sometimes it is helpful to have values associated with enumerations. In Ruby this can be accomplished by using a hash. For example,

traffic\_light = {

:green => “go”

:yellow => “caution”

:red => “stop”}

traffic\_light[:yellow]

In this way values can be associated with the symbols. Members of a hash are accessed using the same bracket syntax as members of arrays. Note only integers can be used in array indexing, thus non-standard use of a symbol as an array index will raise an exception. .

### 6.5.2 Guidance to language users

* Use symbols for enumerators rather than named constants.
* Do not define named constants to represent enumerators.

## 6.6 Conversion Errors [FLC]

### 6.6.1 Applicability to language

Integers in the Ruby language are of unbounded length (the actual limit is dependent on the machine’s memory). When an integer exceeds the word size for the machine there is no rollover and no errors occur. Instead Ruby converts the integer from one type to another. When possible, integers in Ruby are stored in a Fixnum object. Fixnum is a class which has limited integer range, yet is able to store the number efficiently in one machine word. Typically on a 32-bit machine the range is usually -230 to 230-1. These ranges are implementation defined.

Once calculations exceed this range, integers are stored in a Bignum object. Bignum class allows any length (memory providing) integer. This all takes place without the user’s explicit instruction. The result of any Bignum calculation may be returned as an integer if the value can be represented as an integer.

Ruby converts integers to floating point with the user’s explicit intent. Loss of precision can occur converting from a large magnitude integer to a floating point number. This does not generate an error.

*This used to be Numeric Conversion errors. Address all conversion errors.*

### 6.6.2 Guidance to language users

* Be aware that use of Bignums can have performance and storage implications.

## 6.7 String Termination [CJM]

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

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

This vulnerability is not applicable to Ruby since array indexing is checked.

## 6.9 Unchecked Array Indexing [XYZ]

This vulnerability is not applicable to Ruby since array indexing is checked

## 6.10 Unchecked Array Copying [XYW]

his vulnerability is not applicable to Ruby since arrays grow.

## 6.11 Pointer Type Conversions [HFC]

This vulnerability is not applicable to Ruby since users cannot manipulate pointers.

## 6.12 Pointer Arithmetic [RVG]

This vulnerability is not applicable to Ruby since users cannot manipulate pointers.

## 6.13 Null Pointer Dereference [XYH]

This vulnerability is not applicable to Ruby since users cannot create or dereference null pointers.

## 6.14 Dangling Reference to Heap [XYK]

This vulnerability is not applicable to Ruby since users cannot explicitly allocate and explicitly deallocate memory.

## 6.15 Arithmetic Wrap-around Error [FIF]

This vulnerability is not applicable to Ruby since integers are unbounded.

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

This vulnerability is not applicable to Ruby since logic shifts on integers will not modify the sign bit or lose significant bits if the size of the value grows

## 6.17 Choice of Clear Names [NAI]

### 6.17.1 Applicability to language

Ruby is susceptible to errors resulting from similar looking names. Ruby provides scoping of local variables. However, this can be confusing. Local variables cannot be accessed from another method, but local variables can be accessed from a block. Ruby features variable prefixes for non-local variables. The dollar sign signifies a global variable. A single “@” symbol signifies a variable scoped to the current object. A double at symbol signifies a class wide variable, accessible from any instance of said class.

### 6.17.2 Guidance to language users

* Use names that are clear and visually unambiguous.
* Be consistent in choosing names.

## 6.18 Dead store [WXQ]

### 6.18.1 Applicability to language

Ruby is susceptible to errors of accidental assignments resulting from typos of variable names. Since variables do not need to declared before use such an assignment may go unnoticed. Such behaviour is indicative of programmer error

### 6.18.2 Guidance to Language Users

* Check that each assignment is made to the intended variable identifier.
* Use static analysis tools, as they become available, to mechanically identify dead stores in the program

## 6.19 Unused Variable [YZS]

### 6.19.1 Applicability to language

Ruby is susceptible to this vulnerability. Ruby does not permit the declaration of variables, but “declares” parameters, which might never be read or written, hence providing storage space useful to an attacker.

### 6.19.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.19.5 of TR 24772-1.
* Enable detection of unused variables in the processor.

## 6.20 Identifier Name Reuse [YOW]

### 6.20.1 Applicability to language

Ruby employs various levels of scope which allow users to name variables in different scopes with the same name. This can cause confusion in situations where the user is unaware of the scoping rules, especially in the use of blocks.

Modules provide a way to group methods and variables without the need for a class. To use these module and method names must be completely specified. For example:

Base64::encode(text)

However modules can be included, thus putting the contents of the module within the current scope. So:

include Base64

encode(text)

can cause clashes with names already in scope. When this occurs the current scope takes precedence, but the user may not realize this resulting in unknown errors.

### 6.20.2 Guidance to language users

* Follow the guidance of TR 24772-1 subclause 6.20.5???
* Ensure that a definition does not occur in a scope where a different definition is accessible.
* Know what a module defines before including. If any definitions conflict, do not include the module, instead use the fully qualified name to refer to any definitions in the module.

## 6.21 Namespace Issues [BJL]

### 6.21.1 Applicability to language

This is indeed an issue for Ruby. The interpreter will resolve names to the most recent definition as the one to use, possibly redefining a variable. Scoping provides some means of protection, but there are some cases where confusion arises. A method definition cannot access local variables defined outside of its scope, yet a block can access these variables. For example:

x = 50

def power(y)

puts x\*\*y end

power(2) #=> NameError: undefined local variable or method ‘x’

But the following can access the x variable as defined:

x = 50

def execute\_block(y)

yield y end

execute\_block(2) {|y| x\*\*y} #=> 2500

### 6.21.2 Guidance to language users

* Follow the guidance of ???
* Avoid unnecessary includes.
* Do not access variables outside of a block without justification.

## 6.22 Initialization of Variables [LAV]

his vulnerability is not applicable to Ruby since variables cannot be read before they are assigned.

## 6.23 Operator Precedence/Order of Evaluation [JCW]

### 6.23.1 Applicability to language

1. Ruby provides a rich set of operators containing over fifty operators and twenty levels of precedence. Confusion arises especially with operators which mean something similar, but are for different purposes. For example,
2. x = flag\_a or flag\_b

The Ruby language understands this as equivalent to:

(x = flag\_a) or flag\_b

The above assigns the value of flag\_a to x. If flag\_a evaluates to false, then the value of the entire expression is flag\_b. The intent of the programmer was most likely assign true to x if either flag\_a or flag\_b are true:

x = flag\_a || flag\_b

### 6.23.2 Guidance to language users

Follow the mitigation mechanisms of subclause 6.23.5 of TR 24772-1.

* Follow the mitigation mechanisms of TR 24772-1 subclause 6.23.5. ?????
* Use parenthesis around operators which are known to cause confusion and errors.
* Break complex expressions into simpler ones, storing sub-expressions in variables as needed.

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

### 6.24.1 Applicability to language

1. n Ruby method invocations can change the state of the receiver (object whose method is invoked). This occurs not just for input and output for which side-effects are unavoidable, but also for routine operations such as mutating strings, modifying arrays, or defining methods. Ruby has adopted a naming convention which indicates destructive methods (those which modify the receiver) instead of creating a new object which is a modified copy. For example,

array=[1,2,3] #=>[1,2,3] array.slice(1..2) #=> [2, 3] array #=> [1, 2, 3] array.slice!(1..2) #=> [2, 3] array #=> [1]

The method name with the exclamation signifies the object itself will be modified, whereas the other method does not modify it. Sometimes though the method is understood by the user to modify the object or cause side- effects. For example,

array = [1, 2, 3]

array.concat([4, 5, 6])

array #=> [1, 2, 3, 4, 5, 6]

These behaviours are documented and with little effort the user will be able recognize which methods cause side- effects and what those effects are.

The order of evaluation in Ruby is left-to-right. Order of evaluation and order of precedence are different. Precedence allows the familiar order of operations for expressions. For example,

a+b\*c  
a is evaluated, followed by b and c, then the value of b and the value of c are multiplied and added to the value

of a. This is a subtle point which matters only if a, b, or c cause side effects. The following illustrates this:

def a; print “A”; 1; end

def b; print “B”; 2; end

def c; print “C”; 3; end  
a + b \* c #=> 7, and “ABC” is printed to standard output

### 6.24.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.24.5 of TR 24772-1. ???
* Read method documentation to be aware of side-effects.
* Do not depend on side-effects of a term in the expression itself.

## 6.25 Likely Incorrect Expression [KOA]

### 6.25.1 Applicability to language

1. Ruby has operators which are typographically similar, yet which have different meanings. The assignment operator and comparison operators are examples of these. Both are expressions and can be used in conditional expressions.
2. if a = 3 then #...
3. if a == 3 then #...

The first example assigns the value 3 to the variable a. 3 evaluates to true and the conditional is executed. The second checks that the variable a is equal to the value 3 and executes the conditional if true.

Another instance is the use of assignments in Boolean expressions. For instance,

a = x or b = y

This expression assigns the value x to a. If x is false then the value of y will be assigned to b. This should be avoided as the second assignment will not always occur. This could possibly be the intention of the programmer, but a more clear way to write the code which accomplishes that is:

a=x  
b = y if a

There is no confusion here as the second assignment clearly has an if-modifier. This is common and well understood in the Ruby language.

### Guidance to language users

* Follow the mitigation mechanisms of subclause 6.25.5 of TR 24772-1.?????
* Avoid assignments in conditions.
* Do not perform assignments within Boolean expressions

## 6.26 Dead and Deactivated Code [XYQ]

### 6.26.1 Applicability to language

Ruby allows the usual sources of dead code (described in 6.28) that are common to most conventional programming languages. .

### 6.26.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.26.5 of TR 24772-1.

## 6.27 Switch Statements and Static Analysis [CLL]

### 6.27.1 Applicability to language

Ruby provides a case statement. This construct is similar to C’s switch statement with a few important differences. Cases do not “flow through” from one to the next. Only one case will be executed. An else case can be provided, but is not required. If no cases match then the value of the case statement is nil.

### 6.27.2 Guidance to language users

* Include an else clause, unless the intention of cases not covered is to return the value nil.
* Multiple expressions (separated by commas) may be served by the same when clause.

## 6.28 Demarcation of Control Flow [EOJ]

This vulnerability is not applicable to Ruby since control constructs require an explicit termination symbol.

## 6.29 Loop Control Variables [TEX]

### 6.27.1 Applicability to language

Ruby allows the modification of loop control variables from within the body of the loop.

### 6.27.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.26.5 of TR 24772-1. ???
* Do not modify loop control variables inside the loop body

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

### 6.30.1 Applicability to language

Like any programming language which supplies equality operators and array indexing, Ruby is vulnerable to off- by-one-errors. These errors occur when the developer creates an incorrect test for a number range or does not index arrays starting at zero.

Some looping constructs of the language alleviate the problem, but not all of them. For example this code

for i in 1..5

print i

end#=> 12345

In addition to this is the usual confusion associated between <, <=, >, and >= in a test Also unique to Ruby is the confusion of these particular loop constructs:

5.times {|x| p x}

and

Each loop executes the code block five times. However the values passed to the block differ. With 5.times the loop starts with the value 0 and the last value passed to the block is 4. However in the case of 1.upto(5), it starts by passing 1, and ends by passing 5.

### Guidance to language users

* Follow the mitigation mechanisms of subclause 6.30.5 of TR 24772-1. ???
* Use careful programming practice when programming border cases.
* Use static analysis tools to detect off-by-one errors as they become available.
* Instead of writing a loop to iterate all the elements of a container use the each method supplied by the object’s class.

## 6.31 Structured Programming [EWD]

### 6.31.1 Applicability to language

Ruby makes structured programming easy for the user. Its object-oriented nature encourages at least a minimum amount of structure. However, it is still possible to write unstructured code. One feature which allows this is the break statement. The statement ends the execution of the current innermost loop. Excessive use of this may be confusing to others as it is not standard practice.

### Guidance to language users

* Follow the mitigation mechanisms of subclause 6.31.5 of TR 24772-1.
* While there are some cases where it might be necessary to use relatively unstructured programming methods, they should generally be avoided. The following ways help avoid the above named failures of structured programming:
* Instead of using multiple return statements, have a single return statement which returns a variable that has been assigned the desired return value.
* In most cases a break statement can be avoided by using another looping construct. These are abundant in Ruby.
* Use classes and modules to partition functionality.

## 6.32 Passing Parameters and Return Values [CSJ]

### 6.32.1 Applicability to language

Ruby uses call by reference. Each variable is a named reference to an object. Return values in Ruby are merely the object of the last expression, or a return statement. Note that Ruby allows multiple return values by way of array. The following is valid:

return angle, velocity#=> [angle, velocity]

or less verbosely:

[angle, velocity] #as the last line of the method

While pass by reference is a low over-head way of passing parameters, sometimes confusion can arise for programmers. If an object is modified by a method, then the possibility exists that the original object was modified. This may not the intended consequence. For example,

def pig\_latin(word)

word = word[1..-1] << word[0] if !word[/^[aeiouy]/]

word << “ay”

end

The above method modifies the original object if it is that string starts with a vowel. The effect is the value outside the scope of the method is modified. The following revised method avoids this by calling the dup method on the object word:

def pig\_latin\_revised(word)

word = word[/^[aeiouy]/] ? word.dup : word[1..-1] << word[0]

word << “ay”

end

### 6.32.2 Guidance to language users

* Follow avoidance advice in subclause 6.32.5 of TR 24772-1.
* Methods which modify their parameters should have the exclamation mark suffix. This is a standard Ruby idiom alerting users to the behaviour of the method.
* Make local copies of parameters inside methods if they are not intended to be modified.

## 6.33 Dangling References to Stack Frames [DCM]

1. This vulnerability is not applicable to Ruby since users cannot create dangling references.

## 6.34 Subprogram Signature Mismatch [OTR]

### 6.34.1 Applicability to language

Subprogram signatures in Ruby only consist of an arity count and name. A mismatch in the number of parameters will thus be caught before a call is executed. The type of each parameter is not enforced by the interpreter. This is considered a desirable feature of the language, in that an object that responds to the same methods can imitate an object of another type. If an object does not respond to a method an error will be thrown. Also if the implementer chooses they can query the object to test its available methods and choose how to proceed.

### Guidance to language users

* Follow the mitigation mechanisms of subclause 6.34.5 of TR 24772-1.
* The Ruby interpreter will provide error messages for instances of methods called with an inappropriate number of arguments. All other aspects of consistency should be checked thoroughly in accordance with the general guidance found in 6.36.5.

## 6.35 Recursion [GDL]

### 6.35.1 Applicability to language

Ruby permits recursion, hence is subject to the problems described in 6.37 of TR 24772-1.

### 6.35.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.35.5 of TR 24772-1.

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

### 6.36.1 Applicability to language

Ruby provides the class Exception which is used to communicate between raise methods (methods which throw an exception) and rescue statements. Exception objects carry information about the exception including its type, possibly a descriptive string, and optional trace back.

Given this information the programmer can deal with exception appropriately within rescue statements. In some cases this might be program termination, while in other cases an error may be par for the course.

### Guidance to language users

* Follow the mitigation mechanisms of subclause 6.36.5 of TR 24772-1.
* Extend Ruby’s exception handling for your specific application.
* Use the language’s built-in mechanisms (rescue, retry) for dealing with errors.

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

This vulnerability is not applicable to Ruby since every data has a single interpretation.

## 6.38 Deep vs. Shallow Copying [YAN]

### 6.38.1 Applicability to language

The vulnerability described in subclause 6.38 of TR 24772-1 applies to Ada. It can be mitigated somewhat by language constructs that allow the creation of abstractions and the addition of user-defined copying operations, such that inadvertent aliasing problems can be contained within the abstraction. The default semantics of assignment create a shallow copy, when applied to the root of a graph structure.

### 6.38.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.38.5 of TR 24772-1.
* Use controlled types and appropriate redefinitions of the Initialize, Adjust, and Finalize operation to create deep copies when needed.
* Use a pre-existing Container type for trees.

## 6.39 Memory Leak and Heap Fragmentation [XYL]

This vulnerability is no applicable to Ruby since users cannot explicitly allocate memory. .

## 6.40 Templates and Generics [SYM]

This vulnerability is not applicable to Ruby since it does not include templates or generics.

## 6.41 Inheritance [RIP]

### 6.41.1 Applicability to language

Ruby allows classes to inherit from one parent class. In addition to this modules can be included in a class. The class inherits the module’s instance methods, class variables, and constants. Including modules can silently redefine methods or variables. Caution should be exercised when including modules for this reason. At most a class will have one direct superclass.

### Guidance to language users

* Follow the mitigation mechanisms of subclause 6.41.5 of TR 24772-1.
* Provide documentation of encapsulated data, and how each method affects that data.
* Inherit only from trusted sources, and, whenever possible check the version of the superclass during initialization.
* Provide a method that provides versioning information for each class

## 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 [BKK]

### 6.44.1 Applicability to language

TBD

## 6.44.2 Guidance to Language Users

TBD

## 6.45 Extra Intrinsics [LRM]

This vulnerability is not applicable to Ruby since the most recent definition of a method is selected for use.

## 6.46 Argument Passing to Library Functions [TRJ]

### 6.46.1 Applicability to language

T The original Ruby interpreter is written in the C programming language. Because of this many libraries for Ruby have been written to interface with the Ruby and C. The library designer should make the library validate any input before its use.

### Guidance to language users

* Follow the mitigation mechanisms of subclause 6.46.5 of TR 24772-1.???
* Develop wrappers around library functions that check the parameters before calling the function.
* Use only libraries known to validate parameter values.

## 6.47 Inter-language Calling [DJS]

### 6.47.1 Applicability to Language

Ruby is susceptible to this vulnerability when used in a multi-lingual environment.

There is not a standard definition for interactions with other languages. The Ruby language does not mandate calling or return conventions for example. Two conforming Ruby processors may differ enough that binary libraries designed for one may not work in the other.

### Guidance to Language Users

* Follow the mitigation mechanisms of subclause 6.47.5 of TR 24772-1. ???
* Implementations may provide a framework for inter-language calling. Be familiar with the data layout and calling mechanism of said framework.
* Use knowledge of all languages used to form names acceptable in all languages involved.
* Ensure the language in which error checking occurs is the one that handles the error.
* Consider handling any exceptions that might be raised in Ada code before returning to a routine from a foreign language, to prevent possible stack corruption if the foreign language cannot handle exceptions raised in Ada code.

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

### 6.48.1 Applicability to Language

Dynamically-linked code might be a different version at runtime than what was tested during development. This may lead to unpredictable results. Self-modifying code can be written in Ruby.

### Guidance to Language Users

* Verify dynamically linked code being used is the same as that which was tested.
* o not write self-modifying code.

## 6.49 Library Signature [NSQ]

### 6.49.1 Applicability to language

Ruby implementations which interface with libraries must have correct signatures for functions. Creating correct signatures for a large library is cumbersome and should be avoided by using tools.

### 6.49.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.49.5 of TR 24772-1.
* Use tools to create signatures.
* Avoid using libraries without proper signatures

## 6.50 Unanticipated Exceptions from Library Routines [HJW]

### 6.50.1 Applicability to language

Ruby interfaces with libraries which could encounter unanticipated exceptions. In some situations, largely dependent on the interpreter implementation, exceptions can cause unpredictable and possibly fatal results.

### 6.50.2 Guidance to language users

* se library routines which specify all possible exceptions.
* Use libraries which generate Ruby exceptions that can be rescued. .

## 6.51 Pre-Processor Directives [NMP]

This vulnerability is not applicable to Ruby since Ruby lacks a pre-processor.

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

This vulnerability does not apply to Ruby since suppression of language defined run-time checks is not allowed. They are an integral part of the Ruby language.

## 6.53 Provision of Inherently Unsafe Operations [SKL]

This vulnerability does not apply to Ruby. It provides a means for "type-casting" which is safe by honoring classes with the same interface as the same type. If differences are exposed an exception will be raised by the processor. However, unlike statically typed languages, type safety in a Ruby program cannot be checked at compile-time. Therefore, maintenance of type safety in a Ruby program critically depends upon the correct coding of exception handlers to catch and treat type exceptions. Hence, Ruby programmers must pay particular attention to the class of vulnerabilities described in TR 24772-1 clause 6.38 and 6.36 [OYB]

## 6.54 Obscure Language Features [BRS]

This vulnerability is not applicable to Ruby.

## 6.55 Unspecified Behaviour [BQF]

### 6.55.1 Applicability to language

Unspecified behaviour occurs where the proposed Ruby standard does not mandate a particular behaviour. Unspecified behaviour in Ruby is abundant. In the proposed standard there are 136 instances of the phrase

“unspecified behaviour.” Examples of unspecified behaviour are:

* Calling Numeric#coerce(numeric) with the value NaN.
* Calling Integer#&(other) if other is not an instance of the class Integer. This also applies to

Integer#|, Integer#^, Integer#<<, and Integer#>>

* Calling String#\*(num) if other is not an instance of the class Integer

### 6.55.2 Guidance to language users

* Do not rely on unspecified behaviour because the behaviour can change at each instance.
* Code that makes assumptions about the unspecified behaviour should be replaced to make it less reliant on a particular installation and more portable.
* Document instances of use of unspecified behaviour.

## 6.56 Undefined Behaviour [EWF]

### 6.56.1 Applicability to language

Undefined behaviour in Ruby is cover by subclause 6.55 [BQF] and subclause 6.57 [FAB].

### 6.56.2 Guidance to language users

* Avoid using features of the language which are not specified to an exact behaviour

## 6.57 Implementation-Defined Behaviour [FAB]

### 6.57.1 Applicability to language

The proposed Ruby standard defines implementation-defined behaviour as: possibly differing between implementations, but defined for every implementation.

The proposed Ruby standard has documented 98 instances of implementation defined behaviour. Examples of implementation defined behaviour are:

* Whether a singleton class can have class variables or not.
* The direct superclass of Object.
* The visibility of Module#class\_variable\_get.
* Kernel.p(\* args) return value..

### 6.57.2 Guidance to language users

* The abundant nature of implementation-defined behaviour makes it difficult to avoid. As much as possible users should avoid implementation defined behaviour.
* Determine which implementation-defined implementations are shared between implementations. These are safer to use than behaviour which is different for every.

## 6.58 Deprecated Language Features [MEM]

### 6.58.1 Applicability to language

TBD

### 6.58.2 Guidance to language users

TBD

## 6.59 Concurrency – Activation [CGA]

## 6.59.1 Applicability to language

TBD

### 6.59.2 Guidance to language users

* TBD.

## 6.60 Concurrency – Directed termination [CGT]

## 6.60.1 Applicability to language

TBD

### 6.60.2 Guidance to language users

TBD

## 6.61 Concurrent Data Access [CGX]

## 6.61.1 Applicability to language

TBD

### 6.61.2 Guidance to language users

TBD

### 6.62 Concurrency – Premature Termination [CGS]

## 6.62.1 Applicability to language

TBD.

## 6.62.2 Guidance to language users

TBD

## 6.63 Protocol Lock Errors [CGM]

## 6.63.1 Applicability to language

TBD

### 6.63.2 Guidance to language users

TBD

## 6.64 Reliance on external format strings [SHL]

TBD

## 7 Language specific vulnerabilities for Ada

## 8 Implications for standardization

Future standardization efforts should consider the following items to address vulnerability issues identified earlier in this Annex:

* **Pragma** Restrictions could be extended to statically constrain dubious uses of control structures (see 6.31 Structured Programming [EWD]).
* When appropriate, language-defined checks should be added to reduce the possibility of multiple outcomes from a single construct, such as by disallowing side-effects in cases where the order of evaluation could affect the result, similar to those specified for use of “**in out**” or “**out**” parameters of functions (see [6.24 Side-effects and Order of Evaluation [SAM]](#_6.24_Side-effects_and) and 6.55 Unspecified Behaviour [BQF]).
* When appropriate, language-defined checks should be added to reduce the possibility of erroneous execution, such as by disallowing unsynchronized access to shared variables (see 6.56 Undefined Behaviour [EWF]).
* Language standards should specify relatively tight boundaries on implementation-defined behaviour whenever possible, and the standard should highlight what levels represent a portable minimum capability on which programmers may rely. For languages like Ada that allow user declaration of numeric types, the number of predefined numeric types should be minimized (for example, strongly discourage or disallow declarations of Byte\_Integer, Very\_Long\_Integer, and similar, in **package** Standard) (see 6.57 Implementation-Defined Behaviour [FAB]).
* Ada could define a **pragma** Restrictions identifier No\_Hiding that forbids the use of a declaration that result in a local homograph (see 6.20 Identifier Name Reuse [YOW]).
* Ada could add the ability to declare in the specification of a function that it is pure, that is, it has no side effects (see 6.24 Side-effects and Order of Evaluation [SAM]).
* **Pragma** Restrictions could be extended to restrict the use of 'Address attribute to library level static objects (see 6.33 Dangling References to Stack Frames [DCM]).
* Future standardization of Ada should consider implementing a language-provided reference counting storage management mechanism for dynamic objects (see [6.38 Deep vs. Shallow Copying [YAN]](#_6.38_Deep_vs.)).

# Bibliography

[1] AQSG, Ada Quality and Style Guide, Guidelines for Professional Programmers. Available from: https://en.wikibooks.org/wiki/Ada\_Style\_Guide.

[2] Barnes, John, *High Integrity Software - the SPARK Approach to Safety and Security*. Addison-Wesley. 2002.

[3] Barnes, John, Lecture Notes on Computer Science 8338, Ada 2012 Rationale: The Language—The Standard Libraries, Springer, 2013.

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

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

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

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

[8] 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>

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

[10] 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.

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

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

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

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

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

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

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

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

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

[20] ISO/IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic (3 parts)*

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

[22] Lions, J. L. [*ARIANE 5 Flight 501 Failure Report*](http://en.wikisource.org/wiki/Ariane_501_Inquiry_Board_report). Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[23] 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

[24] RTCA SC167 DO178-B/EUROCAE ED-12B, *Software Considerations in Airborne Systems and Equipment Certification*. Issued in the USA by the Requirements and Technical Concepts for Aviation) and in Europe by the European Organization for Civil Aviation Electronics. December 1992.

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

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

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

[28] 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.

# Index

Abnormal representation, 10

**abort**, 31, 46, 49, 50

Access object, 11

Access type, 11

Access value, 11

Access-to-subprogram, 11

Allocator, 11

AMV – Type-breaking Reinterpretation of Data, 37

Aspect specification, 11

Atomic, 11, 14, 46, 49

Attribute, 11

'Access, 34

'Address, 34, 35, 51

'Alignment, 15

'Component\_Size, 15

'Exponent, 21

'First, 33, 47

'Image, 32

'Last, 33, 47

'Length, 33

'Range, 33

'Size, 15

'Unchecked\_Access, 17, 35, 44

'Valid, 42

‘Access, 24, 35

‘Callable, 49, 50

‘Terminated, 49, 50

‘Valid, 20, 29

’Valid, 29

Bit ordering, 11, 12

BJL – Namespace Issues, 28

Bounded Error, 11

BQF – Unspecified Behaviour, 45

BRS – Obscure Language Features, 44

Case choices, 11

Case expression, 11

Case statement, 11, 22, 32

CCB – Enumerator Issues, 21

CGA – Concurrency – Activation, 48

CGM – Protocol Lock Errors, 50

CGS – Concurrency – Premature Termination, 50

CGT – Concurrency – Directed termination, 49

CGX – Concurrent Data Access, 49

CJM – String Termination, 23

CLL – Switch Statements and Static Analysis, 32

Compilation unit, 11

Configuration pragma, 11, 16

Controlled type, 12

CSJ – Passing Parameters and Return Values, 34

DCM – Dangling References to Stack Frames, 34

Dead store, 12

Default expression, 12

Discrete type, 12

Discriminant, 12, 47

DJS – Inter-language Calling, 42

Endianness, 12

Enumeration Representation Clause, 12

Enumeration type, 12, 14

EOJ – Demarcation of Control Flow, 32

Erroneous execution, 12

EWD – Structured Programming, 34

EWF – Undefined Behaviour, 46

Exception, 12, 14, 15, 16, 19, 20, 22, 23, 28, 29, 33, 36, 41, 42, 43, 45, 47, 48, 50

Constraint\_Error, 14, 23, 25, 32, 47

Program\_Error, 14, 15, 45

Storage\_Error, 14, 36

Tasking\_Error, 14, 48

Exception Information, 47

Expanded name, 12

Explicit conversions, 15, 19

FAB – Implementation-Defined Behaviour, 47

FIF – Arithmetic Wrap-around Error, 25

Fixed-point types, 12

FLC – Numeric Conversion Errors, 22

GDL – Recursion, 36

Generic formal subprogram, 12

HCB – Buffer Boundary Violation (Buffer Overflow), 23

HFC – Pointer Type Conversions, 24

Hiding, 12, 14, 51

hidden from all visibility, 14

hidden from direct visibility, 14

HJW – Unanticipated Exceptions from Library Routines, 43

Homograph, 12

Idempotent behaviour, 13

Identifier, 12

Identifier length, 26

IHN–Type System, 19

Implementation defined, 13, 14

Implicit conversions, 15, 19

International character sets, 26

JCW – Operator Precedence/Order of Evaluation, 29

Junk initialization, 29

KOA – Likely Incorrect Expression, 30

Language concepts, 14, 23, 24, 25, 32, 33, 38, 40, 51

Language Vulnerabilities

Argument Passing to Library Functions [TRJ], 39, 40, 41

Arithmetic Wrap-around Error [FIF], 25

Bit Representation [STR], 20

Buffer Boundary Violation (Buffer Overflow) [HCB], 23

Choice of Clear Names [NAI], 26

Concurrency – Activation [CGA], 48

Concurrency – Directed termination [CGT], 49

Concurrency – Premature Termination [CGS], 50

Concurrent Data Access [CGX], 49

Dangling Reference to Heap [XYK], 25

Dangling References to Stack Frames [DCM], 34

Dead and Deactivated Code [XYQ], 31

Dead store [WXQ], 27

Demarcation of Control Flow [EOJ], 32

Deprecated Language Features [MEM], 48

Dynamically-linked Code and Self-modifying Code [NYY], 42

Enumerator Issues [CCB], 21

Extra Intrinsics [LRM], 41

Floating-point Arithmetic [PLF], 21

Identifier Name Reuse [YOW], 27

Ignored Error Status and Unhandled Exceptions [OYB], 36

Implementation-Defined Behaviour [FAB], 47

Inheritance [RIP], 39

Initialization of Variables [LAV], 28

Inter-language Calling [DJS], 42

Library Signature [NSQ], 42

Likely Incorrect Expression [KOA], 30

Loop Control Variables [TEX], 33

Memory Leak [XYL], 38

Namespace Issues [BJL], 28

Numeric Conversion Errors [FLC], 22

Obscure Language Features [BRS], 44

Off-by-one Error [XZH], 33

Operator Precedence/Order of Evaluation [JCW], 29

Passing Parameters and Return Values [CSJ], 34

Pointer Arithmetic [RVG], 24

Pointer Type Conversions [HFC], 24

Protocol Lock Errors [CGM], 50

Provision of Inherently Unsafe Operations [SKL], 44

Recursion [GDL], 36

Side-effects and Order of Evaluation [SAM], 30

String Termination [CJM], 23

Structured Programming [EWD], 34

Subprogram Signature Mismatch [OTR], 35

Suppression of Language-defined Run-time Checking [MXB], 44

Switch Statements and Static Analysis [CLL], 32

Templates and Generics [SYM], 38

Type System [IHN], 19

Type-breaking Reinterpretation of Data [AMV], 37

Unanticipated Exceptions from Library Routines [HJW], 43

Unchecked Array Indexing [XYZ], 23

Undefined Behaviour [EWF], 46

Unspecified Behaviour [BQF], 45

Unused Variable [YZS], 27

Using Shift Operations for Multiplication and Division [PIK], 25

Language Vulnerability

Unchecked Array Copying [XYW], 23

LAV – Initialization of Variables, 28

LRM – Extra Intrinsics, 41

MEM – Deprecated Language Features, 48

Mixed casing, 26

Modular type, 13

MXB – Suppression of Language-defined Run-time Checking, 44

NAI – Choice of Clear Names, 26

NSQ – Library Signature, 42

NYY – Dynamically-linked Code and Self-modifying Code, 42

Obsolescent features, 13

Operational and Representation Attributes, 13, 15

OTR – Subprogram Signature Mismatch, 35

Overriding indicators, 13

OYB – Ignored Error Status and Unhandled Exceptions, 36

Partition, 13

PIK – Using Shift Operations for Multiplication and Division, 25

PLF – Floating-point Arithmetic, 21

Pointer, 13, 28

Polymorphic Variable, 15

Postconditions, 41

Pragma, 13, 44

Configuration pragma, 11

pragma Atomic, 15, 49

pragma Atomic\_Components, 15, 49

pragma Convention, 15, 36, 42

pragma Default\_Storage\_Pool, 17

pragma Detect\_Blocking, 15

pragma Discard\_Names, 15

pragma Export, 16, 36, 42

pragma Import, 16, 36, 37, 42

pragma Normalize\_Scalars, 16, 29

pragma Pack, 16

pragma Restrictions, 16, 17, 44, 45, 48, 51

pragma Suppress, 16, 17, 23, 44, 46

pragma Unchecked Union, 16

pragma Volatile, 16, 49

pragma Volatile\_Components, 16, 49

Preconditions, 41, 42

Program verification, 41

Range check, 13

Record Representation Clauses, 13

RIP – Inheritance, 39

RVG – Pointer Arithmetic, 24

SAM – Side-effects and Order of Evaluation, 30

Scalar type, 13

Separate Compilation, 16

Singular/plural forms, 26

SKL – Provision of Inherently Unsafe Operations, 44

Static expressions, 13

Storage Place Attributes, 13

Storage pool, 11, 13, 14, 16, 17, 38

Storage subpool, 14, 16, 38

STR – Bit Representation, 20

Subtype declaration, 14

SYM – Templates and Generics, 38

Symbols and conventions, 10

Task, 14, 50

Terms and definitions, 10

TEX – Loop Control Variables, 33

TRJ – Argument Passing to Library Functions, 39, 40, 41

Type conversion, 13, 14, 24

Type invariants, 41, 42

Unchecked conversions, 15, 20

Unchecked\_Conversion, 15, 17, 20, 37, 44, 46

Underscores and periods, 26

Unsafe Programming, 17, 22, 23, 24, 25, 32, 33, 38, 40, 44, 51

Unused variable, 14

User-defined floating-point types, 17

User-defined scalar types, 17

Volatile, 14, 27, 49

WXQ – Dead store, 27

XYK – Dangling Reference to Heap, 25

XYL – Memory Leak, 38

XYQ – Dead and Deactivated Code, 31

XYW – Unchecked Array Copying, 23

XYZ – Unchecked Array Indexing, 23

XZH – Off-by-one Error, 33

YOW – Identifier Name Reuse, 27

YZS – Unused Variable, 27