TECHNICAL REPORT

ISO/IEC TR 24772-2

First edition
2020-04

Programming languages — Guidance to avoiding vulnerabilities in programming languages —

Part 2: Ada

Langages de programmation — Conduite pour éviter les vulnérabilités dans les langages de programmation — Partie 2: Ada

Reference number
ISO/IEC TR 24772-2:2020(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

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

This first edition cancels and replaces ISO/IEC TR 24772:2013, which has been split into several parts.

This document is intended to be used with ISO/IEC TR 24772-1, which discusses programming language vulnerabilities in a language independent fashion.

A list of all parts in the ISO/IEC 24772 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

This document provides guidance for the programming language Ada so that application developers considering Ada or using Ada can better avoid the programming constructs that lead to vulnerabilities in software written in the Ada 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 can lead to vulnerabilities in their software. This document can also be used in comparison with companion documents and with the language-independent ISO/IEC TR 24772-1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

It should be noted that this document 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.

Programming languages — Guidance to avoiding vulnerabilities in programming languages —

Part 2: Ada

1 Scope

This document 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 document is applicable to the software developed, reviewed or maintained for any application.

Vulnerabilities described in this document present the way that the vulnerability described in ISO/IEC TR 24772-1 are manifested in Ada.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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 2382, *Information technology — Vocabulary*

ISO/IEC 8652, *Information technology — Programming languages — Ada*

ISO/IEC TR 24772-1, *Programming languages — Guidance to avoiding vulnerabilities in programming languages — Part 1: Language-independent guidance*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO/IEC 8652, ISO/IEC TR 24772-1 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1
abnormal representation
representation of an object that is incomplete or that does not represent any valid value of the object's subtype

3.2
access-to-object
pointer to an object

3.3
access-to-subprogram
pointer to a subprogram (function or procedure)

3.4
access type
type for objects that designate (point to) objects or subprograms

Note 1 to entry: This is often called a *pointer* (3.38) type in other languages.

3.5
access value
value of an *access type* (3.4) that is either null or designates another object or subprogram

3.6
allocator
construct that allocates storage from the heap or from a *storage pool* (3.45)

3.7
aspect specification
mechanism used to specify assertions about the behaviour of subprograms, types and objects as well as operational and representational *attributes* (3.9) of various kinds of entities

3.8
atomic
characteristic of an object that guarantees that every access to an object is an indivisible access to the entity in memory instead of possibly partial, repeated manipulation of a local or register copy

3.9
attribute
characteristic of a declaration that can be queried by special syntax to return a value corresponding to the requested attribute

3.10
bit ordering
implementation defined (3.32) value that is either `High_Order_First` or `Low_Order_First` that permits the specification or query of the way that bits are represented in memory within a single memory unit

3.11
bounded error
error that does not need to be detected either prior to or during run time, but if not detected falls within a bounded range of possible effects

3.12
case statement
statement that provides multiple paths of execution dependent on the value of the selecting expression, but which have only one of the alternative sequences selected

Note 1 to entry: A selecting expression is understood to be an expression that is part of a case statement or *case expression* (3.13) and that determines which choice is taken in executing the case statement or evaluating the case expression; it is of *discrete type* (3.19).

3.13
case expression
expression that provides multiple paths of execution dependent on the value of the selecting expression, but which have only one of the alternative dependent expressions evaluated

Note 1 to entry: A selecting expression is understood to be an expression that is part of a *case statement* (3.12) or case expression and that determines which choice is taken in executing the case statement or evaluating the case expression; it is of *discrete type* (3.19).

3.14
case choice
alternative defined in the *case statement* (3.12) or *case expression* (3.13) which are required to be of the same type as the type of the selecting expression in the case statement or case expression, and by which all possible values of the selecting expression are required to be covered

Note 1 to entry: A selecting expression is understood to be an expression that is part of a case statement or case expression and that determines which choice is taken in executing the case statement or evaluating the case expression; it is of *discrete type* (3.19).

3.15
configuration pragma
directive to the compiler that is used to select *partition* (3.37)-wide or system-wide options and that applies to all compilation units appearing in the compilation or all future compilation units compiled into the same environment

Note 1 to entry: A compilation unit is understood to be the smallest Ada syntactic construct that can be submitted to the compiler, and that is usually held in a single compilation file.

3.16
controlled type
type descended from the language-defined type controlled or limited_controlled which is a specialized type in Ada where the declarer can tightly control the initialization, assignment, and finalization of objects of the type

3.17
dead store
assignment to a variable that is not used in subsequent instructions

3.18
default expression
expression of the formal object type that is used to initialize the formal object if an actual object is not provided

3.19
discrete type
integer type or *enumeration type* (3.23)

3.20
discriminant
parameter for a composite type that is used at elaboration of each object of the type to configure the object

3.21
endianness
byte ordering

3.22
enumeration representation clause
clause used to specify the internal codes for enumeration literals

3.23
enumeration type
discrete type (3.19) defined by an enumeration of its values, which are named by *identifiers* (3.31) or character literals, including the types character and boolean

3.24
erroneous execution
unpredictable result of an execution arising from an error that is not bounded by the language, but that does not need to be detected by the implementation either prior to or during run time

3.25
exception
mechanism to detect an exceptional situation and to initiate processing dedicated to recover from the exceptional situation

Note 1 to entry: Exceptions are raised explicitly by user code or implicitly by language-defined checks.

3.26
expanded name
name that is disambiguated from other identical names by prepending the name with the name of the enclosing scope

Note 1 to entry: For example, the name of an entity E within a package (or any other named enclosing entity) P is expanded or disambiguated by using the alternate name P.E instead of the simple name E.

3.27
fixed-point type
real-valued type with a specified error bound (called the "delta" of the type) that provide arithmetic operations carried out with fixed precision rather than the relative precision of floating-point types

3.28
generic formal subprogram
parameter to a generic package used to specify a subprogram or operator

3.29
hiding
process where a declaration can be hidden, either from direct visibility, or from all visibility, within certain parts of its scope

3.30
homograph
property of two declarations such that they have the same name, and do not overload each other according to the rules of the language

3.31
identifier
simplest form of a name

3.32
implementation defined
defined by a set of possible effects of a construct where the implementation may choose to implement any effect in the set of possible effects

3.33
modular type
integer type with values in the range 0.. modulus − 1 with wrap-around semantics for arithmetic operations, bit-wise "and" and "or" operations, and when defined in package Interfaces, arithmetic and logical shift operations

3.34
obsolescent feature
language feature that has been declared to be obsolescent or deprecated and documented in ISO/IEC 8652:2012, Annex J

3.35
operational and representation attribute
value of certain implementation-dependent characteristics obtained by querying the applicable *attributes* (3.9) and possibly specified by the user

3.36
overriding indicator
indicator that specifies the intent that an operation does or does not override ancestor operations by the same name, and used by the compiler to verify that the operation does (or does not) override an ancestor operation

3.37
partition
part of a program that consists of a set of library units such that each partition executes in a separate address space, possibly on a separate computer, and can execute concurrently with and communicate with other partitions

3.38
pointer
access object or *access value* (3.5)

3.39
pragma
directive to the compiler

3.40
range check
run-time check that ensures the result of an operation is contained within the range of allowable values for a given type or subtype, such as the check done on the operand of a type conversion

3.41
record representation clause
mechanism to specify the layout of components within records, that is, their order, position, and size

3.42
scalar type
any one of numeric, Boolean, enumeration, character and *access types* (3.4)

3.43
static expression
expression with statically known operands that are computed with exact precision by the compiler

3.44
storage place attribute
integer *attribute* (3.9) that specify, for a component of a record, the component position and size within the record

Note 1 to entry: The storage place attributes are: `Position`, `First_Bit` and `Last_Bit`.

3.45
storage pool
named location in an Ada program where all objects of a single *access type* (3.4) are allocated

3.46
storage subpool
separately reclaimable subdivision of a storage pool that is identified by a subpool handle

3.47
subtype declaration
construct that allows programmers to declare a named entity that defines a possibly restricted subset of values of an existing type or subtype, typically by imposing a constraint, such as specifying a smaller range of values

3.48
task
separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks from the same program

3.49
unused variable
variable that is declared but neither read nor written to in the program

3.50
volatile
characteristic of an object that guarantees that updates to the object are always seen in the same order by all *tasks* (3.48)

Note 1 to entry: All *atomic* (3.8) objects are volatile.

4 Language concepts

4.1 Enumeration type

The defining identifiers and defining character literals of an enumeration type are required to be distinct. The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.

4.2 Exception

There is a set of predefined exceptions in Ada in package standard: `Constraint_Error`, `Program_Error`, `Storage_Error` and `Tasking_Error`. One of them is raised when certain language-defined checks fail. User code can define and raise exceptions explicitly.

4.3 Hiding

Where hidden from all visibility, a declaration is not visible at all (neither using a `direct_name` nor a `selector_name`). Where hidden from direct visibility, only direct visibility is lost. Visibility using an expanded name is still possible.

4.4 Implementation defined

Implementations are necessary to document their behaviour in implementation-defined situations.

4.5 Type conversions

Ada uses a strong type system based on name equivalence rules. It distinguishes types, which embody statically checkable equivalence rules, and subtypes, which associate dynamic properties with types, for example, index ranges for array subtypes or value ranges for numeric subtypes. Subtypes are not types and their values are implicitly convertible to all other subtypes of the same type. All subtype and type-conversions ensure by static or dynamic checks that the converted value is within the value range of the target type or subtype. If a static check fails, then the program is rejected by the compiler. If a dynamic check fails, then an exception `Constraint_Error` is raised.

To effect a transition of a value from one type to another, three kinds of conversions can be applied in Ada.

a) Implicit conversions: there are few situations in Ada that allow for implicit conversions. An example is the assignment of a value of a type to a polymorphic variable of an encompassing class. In all cases where implicit conversions are permitted, neither static nor dynamic type safety or application type semantics (see below) are endangered by the conversion.

b) Explicit conversions: various explicit conversions between related types are allowed in Ada. All such conversions ensure by static or dynamic rules that the converted value is a valid value of the target type. Violations of subtype properties cause an exception to be raised by the conversion.

c) Unchecked conversions: Conversions that are obtained by instantiating the generic subprogram `Unchecked_Conversion` are unsafe and enable all vulnerabilities mentioned in 6.3 as the result of a breach in a strong type system. `Unchecked_Conversion` is occasionally needed to interface with type-less data structures, for example, hardware registers.

A guiding principle in Ada is that, with the exception of using instances of `Unchecked_Conversion`, no undefined semantics can arise from conversions and the converted value is a valid value of the target type.

4.6 Operational and Representation Attributes

Some attributes can be specified by the user. For example:

— `X'Alignment`: allows the alignment of objects on a storage unit boundary at an integral multiple of a specified value;

— `X'Size`: denotes the size in bits of the representation of the object;

— `X'Component_Size`: denotes the size in bits of components of the array type X.

4.7 User defined types

Ada allows the usual user-defined types such as records, classes (called tagged records), or access types. In addition, Ada allows for user-defined scalar types which permit specification of value ranges, value constraints, and for floating point and fixed-point types, precision. More advanced typing capabilities allow the user to specify types for communicating concurrently executing entities (tasks) and for synchronized data structures (protected objects).

The typing rules of the language prevent intermixing of objects and values of distinct types.

4.8 Pragma compiler directives

NOTE Each of these pragmas specifies that the similarly named aspect of the type, object, or component denoted by its argument is either True (for parameterless pragmas) or is the value of the pragma parameter.

4.8.1 **Pragma** `Atomic`

Specifies that all reads and updates of an object are indivisible.

4.8.2 **Pragma** `Atomic_Components`

Specifies that all reads and updates of an element of an array are indivisible.

4.8.3 **Pragma** `Convention`

Specifies that an Ada entity should use the conventions of another language.

4.8.4 **Pragma** `Detect_Blocking`

A configuration pragma that specifies that all potentially blocking operations within a protected operation need to be detected, resulting in the `Program_Error` exception being raised.

4.8.5 **Pragma** `Discard_Names`

Specifies that storage used at run-time for the names of certain entities, particularly exceptions and enumeration literals, can be reduced by removing name information from the executable image.