

ISO/IEC JTC 1/SC 22 N **0000**

Date: 2016-07-01

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 – Vulnerability descriptions for the programming language Ada

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

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	7
Introduction.....	8
1. Scope	9
2. Normative references.....	9
3. Terms and definitions, symbols and conventions	9
3.1 Terms and definitions.....	9
4 Language concepts	12
5 General guidance for Ada	15
6 Specific Guidance for Ada.....	16
6.1 General	16
6.2 Type System [IHN].....	16
6.2.1 Applicability to language	16
6.2.2 Guidance to language users.....	16
6.3 Bit Representation [STR]	16
6.3.1 Applicability to language	16
6.3.2 Guidance to language users.....	16
6.4 Floating-point Arithmetic [PLF].....	17
6.4.1 Applicability to language	17
6.4.2 Guidance to language users.....	17
6.5 Enumerator Issues [CCB]	17
6.5.1 Applicability to language	17
6.5.2 Guidance to language users.....	18
6.6 Numeric Conversion Errors [FLC]	18
6.6.1 Applicability to language	18
6.6.2 Guidance to language users.....	18
6.7 String Termination [CJM].....	19
6.8 Buffer Boundary Violation (Buffer Overflow) [HCB]	19
6.9 Unchecked Array Indexing [XYZ].....	19
6.9.1 Applicability to language	19
6.9.2 Guidance to language users.....	19
6.10 Unchecked Array Copying [XYW]	19
6.11 Pointer Type Conversions [HFC].....	19
6.11.1 Applicability to language	19
6.11.2 Guidance to language users.....	20
6.12 Pointer Arithmetic [RVG].....	20
6.13 Null Pointer Dereference [XYH]	20
6.13.1 Applicability to the language	20
6.13.2 Guidance to language users.....	20
6.14 Dangling Reference to Heap [XYK]	20

6.14.1 Applicability to language	20
6.14.2 Guidance to language users	20
6.15 Arithmetic Wrap-around Error [FIF]	21
6.16 Using Shift Operations for Multiplication and Division [PIK]	21
6.17 Choice of Clear Names [NAI]	21
6.17.1 Applicability to language	21
6.17.2 Guidance to language users	22
6.18 Dead store [WXQ]	22
6.18.1 Applicability to language	22
6.18.2 Guidance to Language Users	22
6.19 Unused Variable [YZS]	22
6.19.1 Applicability to language	22
6.19.2 Guidance to language users	22
6.20 Identifier Name Reuse [YOW]	22
6.20.1 Applicability to language	22
6.20.2 Guidance to language users	23
6.21 Namespace Issues [BJL]	23
6.22 Initialization of Variables [LAV]	23
6.22.1 Applicability to language	23
6.22.2 Guidance to language users	24
6.23 Operator Precedence/Order of Evaluation [JCW]	24
6.23.1 Applicability to language	24
6.23.2 Guidance to language users	24
6.24 Side-effects and Order of Evaluation [SAM]	24
6.24.1 Applicability to language	24
6.24.2 Guidance to language users	25
6.25 Likely Incorrect Expression [KOA]	25
6.25.1 Applicability to language	25
6.25.2 Guidance to language users	26
6.26 Dead and Deactivated Code [XYQ]	26
6.26.1 Applicability to language	26
6.26.2 Guidance to language users	26
6.27 Switch Statements and Static Analysis [CLL]	26
6.27.1 Applicability to language	26
6.27.2 Guidance to language users	27
6.28 Demarcation of Control Flow [EOJ]	27
6.29 Loop Control Variables [TEX]	27
6.30 Off-by-one Error [XZH]	27
6.30.1 Applicability to language	27
6.30.2 Guidance to language users	28
6.31 Structured Programming [EWD]	28
6.31.1 Applicability to language	28
6.31.2 Guidance to language users	28
6.32 Passing Parameters and Return Values [CSJ]	28
6.32.1 Applicability to language	28
6.32.2 Guidance to language users	28
6.33 Dangling References to Stack Frames [DCM]	28

6.33.1 Applicability to language	28
6.33.2 Guidance to language users	29
6.34 Subprogram Signature Mismatch [OTR]	29
6.34.1 Applicability to language	29
6.34.2 Guidance to language users	30
6.35 Recursion [GDL]	30
6.35.1 Applicability to language	30
6.35.2 Guidance to language users	30
6.36 Ignored Error Status and Unhandled Exceptions [OYB]	30
6.36.1 Applicability to language	30
6.36.2 Guidance to language users	30
6.37 Fault Tolerance and Failure Strategies [REW]	31
6.37.1 Applicability to language	31
6.37.2 Guidance to language users	31
6.38 Type-breaking Reinterpretation of Data [AMV]	31
6.38.1 Applicability to language	31
6.38.2 Guidance to language users	31
6.39 Memory Leak [XYL]	32
6.39.1 Applicability to language	32
6.39.2 Guidance to language users	32
6.40 Templates and Generics [SYM]	32
6.41 Inheritance [RIP]	33
6.41.1 Applicability to language	33
6.41.2 Guidance to language users	33
6.42 Extra Intrinsic [LRM]	33
6.43 Argument Passing to Library Functions [TRJ]	33
6.43.1 Applicability to language	33
6.43.2 Guidance to language users	33
6.44 Inter-language Calling [DJS]	34
6.44.1 Applicability to Language	34
6.44.2 Guidance to Language Users	34
6.45 Dynamically-linked Code and Self-modifying Code [NYY]	34
6.46 Library Signature [NSQ]	34
6.46.1 Applicability to language	34
6.46.2 Guidance to language users	34
6.47 Unanticipated Exceptions from Library Routines [HJW]	34
6.47.1 Applicability to language	34
6.47.2 Guidance to language users	35
6.48 Pre-Processor Directives [NMP]	35
6.49 Suppression of Language-defined Run-time Checking [MXB]	35
6.49.1 Applicability to Language	35
6.49.2 Guidance to Language Users	35
6.50 Provision of Inherently Unsafe Operations [SKL]	35
6.50.1 Applicability to Language	35
6.51 Obscure Language Features [BRS]	36
6.51.1 Applicability to language	36
6.51.2 Guidance to language users	36

6.52 Unspecified Behaviour [BQF]	36
6.52.1 Applicability to language	36
6.52.2 Guidance to language users	37
6.53 Undefined Behaviour [EWF]	37
6.53.1 Applicability to language	37
6.53.2 Guidance to language users	38
6.54 Implementation-Defined Behaviour [FAB]	38
6.54.1 Applicability to language	38
6.54.2 Guidance to language users	39
6.55 Deprecated Language Features [MEM]	39
6.55.1 Applicability to language	39
6.55.2 Guidance to language users	39
6.56 Concurrency – Activation [CGA]	39
6.56.1 Applicability to language	39
6.56.2 Guidance to language users	39
6.57 Concurrency – Directed termination [CGT]	39
6.57.1 Applicability to language	39
6.57.2 Guidance to language users	40
6.58 Concurrent Data Access [CGX]	40
6.58.1 Applicability to language	40
6.58.2 Guidance to language users	40
6.59 Concurrency – Premature Termination [CGS]	40
6.59.1 Applicability to language	40
6.59.2 Guidance to language users	40
6.60 Protocol Lock Errors [CGM]	41
6.60.1 Applicability to language	41
6.60.2 Guidance to language users	41
6.61 Uncontrolled Format String [SHL]	41
7 Language specific vulnerabilities for Ada	41
8 Implications for standardization	41
Bibliography	43

Index 45

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, 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 Ada or using Ada will be better able to 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 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 through language selection and use – Vulnerability descriptions for the programming language Ada

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 Ada.

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 8652:2012, *Information Technology – Programming Languages—Ada*

[ISO/IEC TR 15942:2000](#), *Guidance for the Use of the Ada programming language in high integrity systems*

[ISO/IEC TR 24718:2005](#), *Guide for the use of the Ada Ravenscar Profile in high integrity systems*

[IEEE 754-2008](#), *Binary Floating Point Arithmetic*, IEEE, 2008

[ANSI/IEEE 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.

Abnormal representation: A representation of an object that is incomplete or that does not represent any valid value of the object's subtype.

Access-to-object: A pointer to an object.

Access-to-subprogram: A pointer to a subprogram (function or procedure).

Access type: The type for objects that designate (point to) objects or subprograms; often called a pointer type in other languages.

Access value: A value of an access type that is either null or designates another object or subprogram.

Allocator: A construct that allocates storage from the heap or from a storage pool.

Aspect specification: The mechanism used to specify assertions about the behaviour of subprograms, types and objects as well as operational and representational attributes of various kinds of entities.

Atomic: A 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.

Attribute: A characteristic of a declaration that can be queried by special syntax to return a value corresponding to the requested attribute.

Bit ordering: An implementation defined 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.

Bounded Error: An error that need not be detected either prior to or during run time, but if not detected falls within a bounded range of possible effects.

Case statement: A statement that provides multiple paths of execution dependent upon the value of the selecting expression, but which will have only one of the alternative sequences selected.

Case expression: The expression that provides multiple paths of execution dependent upon the value of the selecting expression, but which will have only one of the alternative dependent expressions evaluated.

Case choices: The alternatives defined in the case statement or case expression which must be of the same type as the type of the selecting expression in the case statement or case expression, and all possible values of the selecting expression must be covered by the case choices.

Compilation unit: The smallest Ada syntactic construct that may be submitted to the compiler, usually held in a single compilation file.

Configuration pragma: A directive to the compiler that is used to select partition-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.

Controlled type: A type descended from the language-defined type *Controlled* or *Limited_Controlled* which is a specialized type in Ada where an implementer can tightly control the initialization, assignment, and finalization of objects of the type.

Dead store: An assignment to a variable that is not used in subsequent instructions.

Default expression: An expression of the formal object type that is used to initialize the formal object if an actual object is not provided.

Discrete type: An integer type or an enumeration type.

Discriminant: A parameter for a composite type that is used at elaboration of each object of the type to configure the object.

Endianness: Byte ordering.

Enumeration Representation Clause: A clause used to specify the internal codes for enumeration literals.

Enumeration type: A discrete type defined by an enumeration of its values, which may be named by identifiers or character literals, including the types Character and Boolean.

Erroneous execution: The unpredictable result arising from an error that is not bounded by the language, but that, like a bounded error, need not be detected by the implementation either prior to or during run time.

Exception: A mechanism to detect an exceptional situation and to initiate processing dedicated to recover from the exceptional situation; exceptions are raised explicitly by user code or implicitly by language-defined checks.

Expanded name: A mechanism to disambiguate the name of an entity E within a **package** (or any other named enclosing entity) P by permitting the alternate name P.E instead of the simple name E.

Fixed-point types: Real-valued types 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).

Generic formal subprogram: A parameter to a generic package used to specify a subprogram or operator.

Hiding: The process where a declaration can be *hidden*, either from direct visibility, or from all visibility, within certain parts of its scope.

Homograph: A property of two declarations such that they have the same name, and do not overload each other according to the rules of the language.

Identifier: The simplest form of a name.

Idempotent behaviour: The property of an operation that has the same effect whether applied just once or multiple times.

Implementation defined: A set of possible effects of a construct where the implementation may choose to implement any effect in the set of effects.

Modular type: An integer type with values in the **range** 0.. modulus – 1 with wrap-around semantics for arithmetic operations, bit-wise "and" and "or" operations, and, for modular types defined in package Interfaces, arithmetic and logical shift operations.

Obsolescent features: Language features that have been declared to be obsolescent or deprecated and documented in Annex J of the Ada Reference Manual.

Operational and Representation Attributes: The values of certain implementation-dependent characteristics obtained by querying the applicable attributes and possibly specified by the user.

Overriding indicators: An 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.

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

Pointer: An access object or access value.

Pragma: A directive to the compiler.

Range check: A 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.

Record Representation Clauses: a mechanism to specify the layout of components within records, that is, their order, position, and size.

Scalar type: A set of types that includes enumeration types, integer types, and real types.

Selecting expression: The expression that determines which choice is taken in executing the case statement or evaluating the case expression; it is of discrete type.

Static expressions: Expressions with statically known operands that are computed with exact precision by the compiler.

Storage Place Attributes: for a component of a record, the attributes (integer) `Position`, `First_Bit` and `Last_Bit` used to specify the component position and size within the record.

Storage pool: A named location in an Ada program where all of the objects of a single access type will be allocated.

Storage subpool: A separately reclaimable subdivision of a storage pool that is identified by a subpool handle.

Subtype declaration: A 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.

Task: A separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks from the same program.

Unused variable: A variable that is declared but neither read nor written to in the program.

Volatile: A characteristic of an object that guarantees that updates to the object are always seen in the same order by all tasks; all atomic objects are volatile.

4 Language concepts

Enumeration type: The defining identifiers and defining character literals of an enumeration type must be distinct. The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.

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.

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.

Implementation defined: Implementations are required to document their behaviour in implementation-defined situations.

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 Section 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.

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`.

Pragmatic compiler directives

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

Pragma Atomic Components: Specifies that all reads and updates of an element of an array are indivisible.

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

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

Pragma Discard_Names: Specifies that storage used at run-time for the names of certain entities, particularly exceptions and enumeration literals, may be reduced.

Pragma Export: Specifies an Ada entity to be accessed by a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language.

Pragma Import: Specifies an entity defined in a foreign language that may be accessed from an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada.

Pragma Normalize Scalars: A configuration pragma that specifies that an otherwise uninitialized scalar object is set to a predictable value, but out of range if possible.

Pragma Pack: Specifies that storage minimization should be the main criterion when selecting the representation of a composite type.

Pragma Restrictions: Specifies that certain language features are not to be used in a given application. For example, the **pragma Restrictions (No_Obsolescent_Features)** prohibits the use of any deprecated features. This **pragma** is a configuration pragma which means that all program units compiled into the library must obey the restriction.

Pragma Suppress: Specifies that a run-time check need not be performed because the programmer asserts it will always succeed.

Pragma Unchecked Union: Specifies an interface correspondence between a given discriminated type and some C union. The **pragma** specifies that the associated type shall be given a representation that leaves no space for its discriminant(s).

Pragma Volatile: Applicable to a type, an object, or a component, and specifies that the associated objects are volatile, meaning that all updates to the objects are seen in the same order by all tasks.

Pragma Volatile Components: Applicable to an array type or an array object, and specifies that the associated components are volatile, meaning that all updates to the components are seen in the same order by all tasks.

Note: Each of these **pragmas** specifies that the similarly named aspect of the type, object, or component denoted by its argument is True.

Separate Compilation: Ada requires that calls on libraries are checked for invalid situations as if the called routine were part of the current compilation.

Storage Pool: A storage pool can be sized exactly to the requirements of the application by allocating only what is needed for all objects of a single type without using the centrally managed heap. Exceptions raised due to memory failures in a storage pool will not adversely affect storage allocation from other storage pools or from the heap. Storage pools for types whose values are of equal length do not suffer from fragmentation. Storage pools may be divided into subpools, to allow efficient reclamation of a portion of a storage pool.

The following Ada restrictions prevent the application from using allocators in various contexts:

pragma Restrictions(No_Allocators): prevents the use of all allocators.

pragma Restrictions(No_Standard_Allocators_After_Elaboration): prevents the use of allocators after the main program has commenced.

pragma Restrictions(No_Local_Allocators): prevents the use of allocators except within expressions that are evaluated as part of library-unit elaboration.

pragma Restrictions(No_Implicit_Heap_Allocations): prevents the implicit use of heap allocation by the Ada implementation, but allows explicit allocators.

pragma Restrictions(No_Anonymous_Allocators): prevents the use of allocators having an anonymous type.

pragma Restrictions(No_Access_Parameter_Allocators): prevents the use of allocators as the actual parameter for an access parameter.

pragma Restrictions(No_Coextensions): prevents the use of allocators as the initial value for an access discriminant.

pragma Default_Storage_Pool(null): specifies that no allocators are permitted for access types that do not specify their own `Storage_Pool` or `Storage_Size`.

pragma Restrictions(No_Unchecked_Deallocations): prevents allocated storage from being deallocated and hence effectively enforces storage pool memory approaches or a completely static approach to access types. Storage pools are not affected by this restriction as explicit routines to free memory for a storage pool can be created.

Unsafe Programming: In recognition of the occasional need to step outside the type system or to perform “risky” operations, Ada provides clearly identified language features to do so. Examples include the generic `Unchecked_Conversion` for unsafe type-conversions or `Unchecked_Deallocation` for the deallocation of heap objects regardless of the existence of surviving references to the object. If unsafe programming is employed in a unit, then the unit needs to specify the respective generic unit in its context clause, thus identifying potentially unsafe units. Similarly, there are ways to create a potentially unsafe global pointer to a local object, using the `Unchecked_Access` attribute. A restriction pragma can be used to disallow uses of `Unchecked_Access`. The **`pragma Suppress`** allows an implementation to omit certain run-time checks.

User-defined floating-point types: Types declared by the programmer that allow specification of digits of precision and optionally a range of values.

User-defined scalar types: Types declared by the programmer for defining ordered sets of values of various kinds, namely integer, enumeration, floating-point, and fixed-point types. The typing rules of the language prevent intermixing of objects and values of distinct types.

5 General guidance for Ada

Ada has been designed with emphasis on software engineering principles that support the development of high-integrity applications. For example, Ada is strongly typed thereby preventing vulnerabilities associated with type mismatch. Similarly, Ada includes boundary checking on arrays as part of the standard language which prevents buffer overflow vulnerabilities. Most of the language may be used to develop applications without known vulnerabilities. The following sections provide guidance to mitigate against known vulnerabilities in Ada.

6 Specific Guidance for Ada

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 Ada code. This section mirrors TR 24772-1 clause 6 in that the vulnerability “Type System [IHN]” is found in clause 6.2 of TR 24772-1, and Ada specific guidance is found in clause 6.2 in this TR.

6.2 Type System [IHN]

6.2.1 Applicability to language

Implicit conversions cause no application vulnerability, as long as the resulting exceptions are properly handled.

Assignment between types cannot be performed except by using an explicit conversion.

Failure to apply correct unit conversion factors when explicitly converting among types for different units will result in application failures due to incorrect values.

Failure to handle the exceptions raised by failed checks of dynamic subtype properties causes the execution of the whole system, a thread, or an inner nested scope to halt abruptly.

Unchecked conversions circumvent the type system and therefore can cause unspecified behaviour (see [6.38 Type-breaking Reinterpretation of Data \[AMV\]](#)).

6.2.2 Guidance to language users

- Apply the predefined ‘Valid attribute for a given subtype to any value as needed to ascertain if the value is a valid value of the subtype. This is especially useful when interfacing with type-less systems or after Unchecked_Conversion.
- Consider restricting explicit conversions to the bodies of user-provided conversion functions that are then used as the only means to effect the transition between unit systems. Review these bodies critically for proper conversion factors.
- Handle exceptions raised by type and subtype-conversions.
- Consider using the restriction No_Unchecked_Conversion to prevent circumventing the type system.

6.3 Bit Representation [STR]

6.3.1 Applicability to language

In general, the type system of Ada protects against the vulnerabilities outlined in Section 6.3 of TR 24772-1. The vulnerabilities caused by the inherent conceptual complexity of bit level programming are as described in Section 6.3 of TR 24772-1.

6.3.2 Guidance to language users

In order to mitigate the vulnerabilities associated with the complexity of bit-level programming

- Use record and array types with the appropriate representation specifications added so that the objects are accessed by their logical structure rather than their physical representation. These representation specifications may address: order, position, and size of data components and fields.
- Use **pragma Atomic** and **pragma Atomic_Components** to ensure that all updates to objects and components happen atomically.
- Use **pragma Volatile** and **pragma Volatile_Components** to ensure that all tasks see updates to the associated objects or array components in the same order.
- Query the default object layout chosen by the compiler to determine the expected behaviour of the final representation.
- Use the restriction **No_Unchecked_Conversion** to prevent circumventing the type system.

For the traditional approach to bit-level programming, Ada provides modular types and literal representations in arbitrary base from 2 to 16 to deal with numeric entities and correct handling of the sign bit. The use of **pragma Pack** on arrays of Booleans provides a type-safe way of manipulating bit strings and eliminates the use of error-prone arithmetic operations.

6.4 Floating-point Arithmetic [PLF]

6.4.1 Applicability to language

Ada specifies adherence to the IEEE Floating Point Standards (IEEE-754-2008, IEEE-854-1987).

The vulnerability in Ada is as described in Section 6.4.2 of TR 24772-1.

6.4.2 Guidance to language users

- Rather than using predefined types, such as `Float` and `Long_Float`, whose precision may vary according to the target system, declare floating-point types that specify the required precision (for example, **digits 10**). Additionally, specifying ranges of a floating point type enables constraint checks which prevents the propagation of infinities and NaNs.
- Avoid comparing floating-point values for equality. Instead, use comparisons that account for the approximate results of computations. Consult a numeric analyst when appropriate.
- Make use of static arithmetic expressions and static constant declarations when possible, since static expressions in Ada are computed at compile time with exact precision.
- Use Ada's standardized numeric libraries (for example, `Generic_Elementary_Functions`) for common mathematical operations (trigonometric operations, logarithms, and others).
- Use an Ada implementation that supports the Numerics Annex of the Ada language reference manual, and employ the "strict mode" of that Annex in cases where additional accuracy requirements must be met by floating-point arithmetic and the operations of predefined numerics packages, as defined and guaranteed by the Annex.
- Avoid direct manipulation of bit fields of floating-point values, since such operations are generally target-specific and error-prone. Instead, make use of Ada's predefined floating-point attributes (such as `'Exponent`).
- In cases where absolute precision is needed, consider replacement of floating-point types and operations with fixed-point types and operations.

6.5 Enumerator Issues [CCB]

6.5.1 Applicability to language

Enumeration representation specification may be used to specify non-default representations of an enumeration type, for example when interfacing with external systems. All of the values in the enumeration type must be

defined in the enumeration representation specification. The numeric values of the representation must preserve the original order. For example:

```
type IO_Types is (Null_Op, Open, Close, Read, Write, Sync);  
for IO_Types use (Null_Op => 0, Open => 1, Close => 2,  
                  Read => 4, Write => 8, Sync => 16);
```

An array may be indexed by such a type. Ada does not prescribe the implementation model for arrays indexed by an enumeration type with non-contiguous values. Two options exist: Either the array is represented “with holes” and indexed by the values of the enumeration type, or the array is represented contiguously and indexed by the position of the enumeration value rather than the value itself. In the former case, the vulnerability described in 6.6 exists only if unsafe programming is applied to access the array or its components outside the protection of the type system. Within the type system, the semantics are well defined and safe. The vulnerability of unexpected but well-defined program behaviour upon extending an enumeration type exists in Ada. In particular, subranges or **others** choices in aggregates and case statements are susceptible to unintentionally capturing newly added enumeration values.

6.5.2 Guidance to language users

- For **case** statements and aggregates, do not use the **others** choice.
- For **case** statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition.

6.6 Numeric Conversion Errors [FLC]

6.6.1 Applicability to language

Ada does not permit implicit conversions between different numeric types, hence cases of implicit loss of data due to truncation cannot occur as they can in languages that allow type coercion between types of different sizes.

In the case of explicit conversions, Ada language rules prevent numeric conversion errors, as follows:

- Range bound checks are applied, so no truncation can occur, and an exception will be generated if the operand of the conversion exceeds the bounds of the target type or subtype.
- Ada permits the definition of subtypes of existing types that can impose a restricted range of values, and implicit conversions can occur for values of different subtypes belonging to the same type, but such conversions still involve range checks that prevent any loss of data or violation of the bounds of the target subtype.

Precision is lost only on explicit conversion from a real type to an integer type or a real type of less precision.

6.6.2 Guidance to language users

- Use Ada's capabilities for user-defined scalar types and subtypes to avoid accidental mixing of logically incompatible value sets.
- Do not suppress range checks on conversions involving scalar types and subtypes to prevent generation of invalid data.
- Use static analysis tools during program development to verify that conversions cannot violate the range of their target.

6.7 String Termination [CJM]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as strings in Ada are not delimited by a termination character. Ada programs that interface to languages that use null-terminated strings and manipulate such strings directly should apply the vulnerability mitigations recommended for that language.

6.8 Buffer Boundary Violation (Buffer Overflow) [HCB]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada (see [6.9 Unchecked Array Indexing \[XYZ\]](#) and [6.10 Unchecked Array Copying \[XYW\]](#)).

6.9 Unchecked Array Indexing [XYZ]

6.9.1 Applicability to language

All array indexing is checked automatically in Ada, and raises an exception when indexes are out of bounds. This is checked in all cases of indexing, including when arrays are passed to subprograms.

An explicit suppression of the checks can be requested by use of **pragma Suppress**, in which case the vulnerability would apply; however, such suppression is easily detected, and generally reserved for tight time-critical loops, even in production code.

6.9.2 Guidance to language users

- Do not suppress the checks provided by the language.
- Use Ada's support for whole-array operations, such as for assignment and comparison, plus aggregates for whole-array initialization, to reduce the use of indexing.
- Write explicit bounds tests to prevent exceptions for indexing out of bounds.

6.10 Unchecked Array Copying [XYW]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as Ada allows arrays to be copied by simple assignment (":="). The rules of the language ensure that no overflow can happen; instead, the exception `Constraint_Error` is raised if the target of the assignment is not able to contain the value assigned to it. Since array copy is provided by the language, Ada does not provide unsafe functions to copy structures by address and length.

6.11 Pointer Type Conversions [HFC]

6.11.1 Applicability to language

The mechanisms available in Ada to alter the type of a pointer value are unchecked type-conversions and type-conversions involving pointer types derived from a common root type. In addition, uses of the unchecked address taking capabilities can create pointer types that misrepresent the true type of the designated entity (see Section 13.10 of the Ada language reference manual).

The vulnerabilities described in TR 24772-1 Section 6.11 exist in Ada only if unchecked type-conversions or unsafe taking of addresses are applied (see [4 Language concepts](#)). Other permitted type-conversions can never misrepresent the type of the designated entity.

Checked type-conversions that affect the application semantics adversely are possible. For example, when a pointer to a class-wide type is changed to a leaf type a run-time check is required.

6.11.2 Guidance to language users

- Do not use the features explicitly identified as unsafe.
- Use 'Access which is always type safe.
- Consider using the restriction `No_Unchecked_Conversion`, `No_Unchecked_Access`, and `No_Use_Of_Attribute(Address)` to prevent circumventing the type system.

6.12 Pointer Arithmetic [RVG]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as Ada does not allow pointer arithmetic.

6.13 Null Pointer Dereference [XYH]

6.13.1 Applicability to the language

In Ada, this vulnerability is mitigated by compile-time or run-time checks that ensure that no null-value can be dereferenced. Of course, the `Constraint_Error` exception implicitly raised upon such dereferencing needs to be handled or else the vulnerability of a failing system or components prevails.

6.13.2 Guidance to language users

- Use non-null access types where possible.
- Write explicit checks for null values to avoid exceptions being raised.
- Handle exceptions raised by attempts to dereference null values.

6.14 Dangling Reference to Heap [XYK]

6.14.1 Applicability to language

Use of `Unchecked_Deallocation` can cause dangling references to the heap. The vulnerabilities described in TR 24772-1 Section 6.14 exist in Ada, when this feature is used, since `Unchecked_Deallocation` may be applied even though there are outstanding references to the deallocated object.

Ada provides a model in which whole collections of heap-allocated objects can be deallocated safely, automatically and collectively when the scope of the root access type or the scope of any associated storage pool object ends.

For global access types, unless storage pools are used, allocated objects can only be deallocated through an instantiation of the generic procedure `Unchecked_Deallocation`.

6.14.2 Guidance to language users

- Use local access types where possible.
- Do not use `Unchecked_Deallocation` or consider applying the restriction `No_Unchecked_Deallocation` to enforce this.
- Use `Controlled` types and reference counting.
- Consider the use of storage pools and subpools.

6.15 Arithmetic Wrap-around Error [FIF]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as wrap-around arithmetic in Ada is limited to modular types. Arithmetic operations on such types use modulo arithmetic, and thus no such operation can create an invalid value of the type.

For non-modular arithmetic, Ada raises the predefined exception `Constraint_Error` whenever a wrap-around occurs but implementations are allowed to refrain from doing so when a correct final value is obtained. In Ada there is no confusion between logical and arithmetic shifts.

6.16 Using Shift Operations for Multiplication and Division [PIK]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as shift operations in Ada are limited to the modular types declared in the standard package `Interfaces`, which are not signed entities.

6.17 Choice of Clear Names [NAI]

6.17.1 Applicability to language

There are two possible issues: the use of the identical name for different purposes (overloading) and the use of similar names for different purposes.

This vulnerability does not address overloading, which is covered in Section 6.20 Identifier Name Reuse [YOW] of TR 24772-1.

The risk of confusion by the use of similar names might occur through:

- Mixed casing. Ada treats upper and lower case letters in names as identical. Confusion may arise through an attempt to use `Item` and `ITEM` as distinct identifiers with different meanings.
- Underscores and periods. Ada permits single underscores in identifiers and they are significant. Thus `BigDog` and `Big_Dog` are different identifiers. But multiple underscores (which might be confused with a single underscore) are forbidden, thus `Big__Dog` is forbidden. Leading and trailing underscores are also forbidden. Periods are not permitted in identifiers at all.
- Singular/plural forms. Ada does permit the use of identifiers which differ solely in this manner such as `Item` and `Items`. However, the user might use the identifier `Item` for a single object of a type `T` and the identifier `Items` for an object denoting an array of items that is of a type `array (...) of T`. The use of `Item` where `Items` was intended or vice versa will be detected by the compiler because of the type violation and the program rejected so no vulnerability would arise.
- International character sets. Ada compilers strictly conform to the appropriate International Standard for character sets.
- Identifier length. All characters in an identifier in Ada are significant. Thus `Long_IdentifierA` and `Long_IdentifierB` are always different. An identifier cannot be split over the end of a line. The only restriction on the length of an identifier is that enforced by the line length and this is guaranteed by the language standard to be no less than 200.

Ada permits the use of names such as `X`, `XX`, and `XXX` (which might all be declared as integers) and a programmer could easily, by mistake, write `XX` where `X` (or `XXX`) was intended. Ada does not attempt to catch such errors.

The use of the wrong name will typically result in a failure to compile so no vulnerability will arise. But, if the wrong name has the same type as the intended name, then an incorrect executable program will be generated.

6.17.2 Guidance to language users

This vulnerability can be avoided or mitigated in Ada in the following ways:

- Avoid the use of similar names to denote different objects of the same type.
- Adopt a project convention for dealing with similar names
- See the Ada Quality and Style Guide.

6.18 Dead store [WXQ]

6.18.1 Applicability to language

This vulnerability exists in Ada as described in TR 24772-1 Section 6.18, with the exception that in Ada if a variable is read by a different thread (task) than the thread that wrote a value to the variable it is not a dead store. Simply marking a variable as being `Volatile` is usually considered to be too error-prone for inter-thread (task) communication by the Ada community, and Ada has numerous facilities for safer inter thread communication.

Ada compilers do exist that detect and generate compiler warnings for dead stores.

The error in TR 24772-1 Section 6.18.3 that the planned reader misspells the name of the store is possible but highly unlikely in Ada since all objects must be declared and typed and the existence of two objects with almost identical names and compatible types (for assignment) in the same scope would be readily detectable.

6.18.2 Guidance to Language Users

- Use Ada compilers that detect and generate compiler warnings for dead stores.
- Use static analysis tools to detect such problems.

6.19 Unused Variable [YZS]

6.19.1 Applicability to language

This vulnerability exists in Ada as described in Section 6.19 of TR 24772-1, although Ada compilers do exist that detect and generate compiler warnings for unused variables.

6.19.2 Guidance to language users

- Do not declare variables of the same type with similar names. Use distinctive identifiers and the strong typing of Ada (for example through declaring specific types such as `Pig_Counter is range 0 .. 1000`; rather than just `Pig: Integer`;) to reduce the number of variables of the same type.
- Use Ada compilers that detect and generate compiler warnings for unused variables.
- Use static analysis tools to detect unused variables.

6.20 Identifier Name Reuse [YOW]

6.20.1 Applicability to language

Ada is a language that permits local scope, and names within nested scopes can hide identical names declared in an outer scope. As such it is susceptible to the vulnerability. For subprograms and other overloaded entities the problem is reduced by the fact that hiding also takes the signatures of the entities into account. Entities with different signatures, therefore, do not hide each other.

Name collisions with keywords cannot happen in Ada because keywords are reserved.

The mechanism of failure identified in TR 24772-1 Section 6.20.3 regarding the declaration of non-unique identifiers in the same scope cannot occur in Ada because all characters in an identifier are significant.

6.20.2 Guidance to language users

- Use *expanded names* whenever confusion may arise.
- Use Ada compilers that generate compile time warnings for declarations in inner scopes that hide declarations in outer scopes.
- Use static analysis tools that detect the same problem.

6.21 Namespace Issues [BJL]

This vulnerability is not applicable to Ada because Ada does not attempt to disambiguate conflicting names imported from different packages. Instead, use of a name with conflicting imported declarations causes a compile time error. The programmer can disambiguate the name usage by using an expanded name that identifies the exporting package.

6.22 Initialization of Variables [LAV]

6.22.1 Applicability to language

As in many languages, it is possible in Ada to make the mistake of using the value of an uninitialized variable. However, as described below, Ada prevents some of the most harmful possible effects of using the value.

The vulnerability does not exist for pointer variables (or constants). Pointer variables are initialized to **null** by default, and every dereference of a pointer that is not null-excluding is checked for a null value.

The checks mandated by the type system apply to the use of uninitialized variables as well. Use of an out-of-bounds value in relevant contexts causes an exception, regardless of the origin of the faulty value. (See [6.36 Ignored Error Status and Unhandled Exceptions \[OYB\]](#) regarding exception handling.) Thus, the only remaining vulnerability is the potential use of a faulty but subtype-conformant value of an uninitialized variable, since it is technically indistinguishable from a value legitimately computed by the application.

For scalar types, the `Default_Value` aspect may be specified to provide a default initial value for otherwise uninitialized objects of the type.

For record types, default initializations may be specified as part of the type definition. For record types, aggregate values may be used to initialize an object to ensure that all components of the object have been initialized with a value.

For controlled types (those descended from the language-defined type `Controlled` or `Limited_Controlled`), the user may also specify an `Initialize` procedure which is invoked on all default-initialized objects of the type.

The **pragma** `Normalize_Scalars` can be used to ensure that scalar variables are always initialized by the compiler in a repeatable fashion. This **pragma** is designed to initialize variables to an out-of-range value if there is one, to avoid hiding errors.

Lastly, the user can query the validity of a given value. The expression `X'Valid` yields true if the value of the scalar variable `X` conforms to the subtype of `X` and false otherwise. Thus, the user can protect against the use of out-of-bounds uninitialized or otherwise corrupted scalar values.

6.22.2 Guidance to language users

This vulnerability can be avoided or mitigated in Ada in the following ways:

- If the compiler has a mode that detects use before initialization, then enable this mode and treat any such warnings as errors.
- Where appropriate, specify explicit initializations or default initializations.
- Use the `pragma Normalize_Scalars` to cause out-of-range default initializations for scalar variables.
- Use the `'Valid` attribute to identify out-of-range scalar values caused by the use of uninitialized variables, without incurring the raising of an exception. Note an implementation may raise an exception for an `Unchecked_Conversion` in this case.

Common advice that should be avoided is to perform a “junk initialization” of variables. Initializing a variable with an inappropriate default value such as zero can result in hiding underlying problems, because the compiler or other static analysis tools will then be unable to detect that the variable has been used prior to receiving a correctly computed value.

6.23 Operator Precedence/Order of Evaluation [JCW]

6.23.1 Applicability to language

Since this vulnerability is about “incorrect beliefs” of programmers, there is no way to establish a limit to how far incorrect beliefs can go. However, Ada is less susceptible to that vulnerability than many other languages, since

- Ada only has six levels of precedence and associativity is closer to common expectations. For example, an expression like `A = B or C = D` will be parsed as expected, as `(A = B) or (C = D)`.
- Mixed logical operators are not allowed without parentheses, for example, “`A or B or C`” is valid, as well as “`A and B and C`”, but “`A and B or C`” is not; the user must write “`(A and B) or C`” or “`A and (B or C)`”.
- Assignment is not an operator in Ada.

6.23.2 Guidance to language users

Apply the general mitigation measures in Ada like in any other language.

6.24 Side-effects and Order of Evaluation [SAM]

6.24.1 Applicability to language

There are no operators in Ada with direct side effects on their operands using the language-defined operations, especially not the increment and decrement operation. Ada does not permit multiple assignments in a single expression or statement.

There is the possibility though to have side effects through function calls in expressions where the function modifies globally visible variables or “`in out`” or “`out`” parameters. Ada disallows multiple uses of the same variable within a single expression if one or more of the uses are as “`in out`” or “`out`” parameters. Operators in Ada are functions with only “`in`” parameters, so, when defined by the user, although they cannot modify their own operands, they may modify global state and therefore have side effects.

Ada allows the implementation to choose the order of evaluation of expressions with operands of the same precedence level, the order of association is left-to-right. The operands of a binary operation are also evaluated in an arbitrary order, as happens for the parameters of any function call. In the case of user-defined operators with side effects on global state, this implementation dependency can cause unpredictability of the side effects.

6.24.2 Guidance to language users

- Make use of one or more programming guidelines which prohibit functions that modify global state, and can be enforced by static analysis.
- Minimize use of “**in out**” and “**out**” parameters for functions.
- Keep expressions simple. Complicated code is prone to error and difficult to maintain.
- Always use brackets to indicate order of evaluation of operators of the same precedence level.

6.25 Likely Incorrect Expression [KOA]

6.25.1 Applicability to language

An instance of this vulnerability consists of two syntactically similar constructs such that the inadvertent substitution of one for the other may result in a program which is accepted by the compiler but does not reflect the intent of the author.

The examples given in TR 24772-1 Section 6.25 are not problems in Ada because of Ada's strong typing and because an assignment is not an expression in Ada.

In Ada, a type-conversion and a qualified expression are syntactically similar, differing only in the presence or absence of a single character:

Type_Name (Expression) -- a type-conversion

vs.

Type_Name'(Expression) -- a qualified expression

Typically, the inadvertent substitution of one for the other results in either a semantically incorrect program which is rejected by the compiler or in a program which behaves in the same way as if the intended construct had been written. In the case of a constrained array subtype, the two constructs differ in their treatment of sliding (conversion of an array value with bounds 100 .. 103 to a subtype with bounds 200 .. 203 will succeed; qualification will fail a run-time check).

Similarly, a timed entry call and a conditional entry call with an else-part that happens to begin with a **delay** statement differ only in the use of “**else**” vs. “**or**” (or even “**then abort**” in the case of an `asynchronous_select` statement).

Probably the most common correctness problem resulting from the use of one kind of expression where a syntactically similar expression should have been used has to do with the use of short-circuit vs. non-short-circuit Boolean-valued operations (for example, “**and then**” and “**or else**” vs. “**and**” and “**or**”), as in

```
if (Ptr /= null) and (Ptr.all.Count > 0) then ... end if;
```

-- should have used “**and then**” to avoid dereferencing null

6.25.2 Guidance to language users

- Compilers and other static analysis tools can detect some cases (such as the preceding example). Use these capabilities.
- Consider using short-circuit forms by default (errors resulting from the incorrect use of short-circuit forms are much less common), though this can make it more difficult to express the distinction between the cases where short-circuited evaluation is known to be needed (either for correctness or for performance) and those where it is not.

6.26 Dead and Deactivated Code [XYQ]

6.26.1 Applicability to language

Ada allows the usual sources of dead code (described in TR 24772-1 Section 6.26) that are common to most conventional programming languages.

6.26.2 Guidance to language users

Use implementation-specific mechanisms, if provided, to support the elimination of dead code. In some cases, use **pragmas** such as `Restrictions`, `Suppress`, or `Discard_Names` to inform the compiler that some code whose generation would normally be required for certain constructs would be dead because of properties of the overall system, and that therefore the code need not be generated. For example:

```
package Pkg is
  type Enum is (Aaa, Bbb, Ccc);
  pragma Discard_Names( Enum );
end Pkg;
```

If `Pkg.Enum'Image` and related attributes (e.g., `Value`, `Wide_Image`) of the type `Enum` are never used, and if the implementation normally builds a table of the enumeration literals, then the **pragma** allows the elimination of the table.

6.27 Switch Statements and Static Analysis [CLL]

6.27.1 Applicability to language

With the exception of unsafe programming (see [4 Language concepts](#)) and the use of default cases, this vulnerability is not applicable to Ada as Ada ensures that a case statement provides exactly one alternative for each value of the expression's subtype. This restriction is enforced at compile time. The **others** clause may be used as the last choice of a case statement to capture any remaining values of the case expression type that are not covered by the preceding case choices. If the value of the expression is outside of the range of this subtype (e.g., due to an uninitialized variable), then the resulting behaviour is well-defined (`Constraint_Error` is raised). Control does not flow from one alternative to the next. Upon reaching the end of an alternative, control is transferred to the end of the **case** statement.

The remaining vulnerability is that unexpected values are captured by the **others** clause or a subrange as case choice. For example, when the range of the type `Character` was extended from 128 characters to the 256 characters in the Latin-1 character type, an **others** clause for a **case** statement with a `Character` type case expression originally written to capture cases associated with the 128 characters type now also captures the 128

additional cases introduced by the extension of the type Character. Some of the new characters may have needed to be covered by the existing case choices or new case choices.

6.27.2 Guidance to language users

- For **case** statements and aggregates, avoid the use of the **others** choice.
- For **case** statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition.¹

6.28 Demarcation of Control Flow [EOJ]

This vulnerability is not applicable to Ada as the Ada syntax describes several types of compound statements that are associated with control flow including **if** statements, **loop** statements, **case** statements, **select** statements, and extended **return** statements. Each of these forms of compound statements require unique syntax that marks the end of the compound statement.

6.29 Loop Control Variables [TEX]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as Ada defines a **for loop** where the number of iterations is controlled by a loop control variable (called a loop parameter). This value has a constant view and cannot be updated within the sequence of statements of the body of the loop.

6.30 Off-by-one Error [XZH]

6.30.1 Applicability to language

Confusion between the need for < and <= or > and >= in a test.

A **for loop** in Ada does not require the programmer to specify a conditional test for loop termination. Instead, the starting and ending value of the loop are specified which eliminates this source of off-by-one errors. A **while loop** however, lets the programmer specify the loop termination expression, which could be susceptible to an off-by-one error. There are also special **for loop** structures that iterate through an entire array or container. These avoid the need to specify any bounds for the iteration.

Confusion as to the index range of an algorithm.

Although there are language defined attributes to symbolically reference the start and end values for a loop iteration, the language does allow the use of explicit values and loop termination tests. Off-by-one errors can result in these circumstances.

Care should be taken when using the 'Length attribute in the loop termination expression. The expression should generally be relative to the 'First value.

The strong typing of Ada eliminates the potential for buffer overflow associated with this vulnerability. If the error is not statically caught at compile time, then a run-time check generates an exception if an attempt is made to access an element outside the bounds of an array.

¹ This case is somewhat specialized but is important, since enumerations are the one case where subranges turn *bad* on the user.

Failing to allow for storage of a sentinel value.

Ada does not use sentinel values to terminate arrays. There is no need to account for the storage of a sentinel value, therefore this particular vulnerability concern does not apply to Ada.

6.30.2 Guidance to language users

- Whenever possible, use a **for loop** instead of a **while loop**.
- Whenever possible, use the form of iteration that takes the name of the array or container and nothing more.
- When indices are necessary, use the 'First, 'Last, and 'Range attributes for loop termination, e.g. **for I in MyArray'Range loop...**
- If the 'Length attribute must be used, take extra care to ensure that the index computation considers the starting index value for the array.

6.31 Structured Programming [EWD]

6.31.1 Applicability to language

Ada programs can exhibit many of the vulnerabilities noted in TR 24772-1 Section 6.31: leaving a **loop** at an arbitrary point, local jumps (**goto**), and multiple exit points from subprograms.

Ada however does not suffer from non-local jumps and multiple entries to subprograms.

6.31.2 Guidance to language users

Minimize the use of **goto**, **loop exit** statements, **return** statements in **procedures** and more than one **return** statement in a **function**. Use multiple exit points only if it makes the code of the exited construct significantly clearer.

6.32 Passing Parameters and Return Values [CSJ]

6.32.1 Applicability to language

Ada employs the mechanisms (for example, modes **in**, **out** and **in out**) that are recommended in Section 6.32 of TR 24772-1. These mode definitions are not optional, mode **in** being the default. The remaining vulnerability is aliasing when a large object is passed by reference. In addition, in Ada, a function result type must be specified and the return value should be assigned to the same type variable, making it much more obvious to the reader if a function result is not being used.

6.32.2 Guidance to language users

- Follow avoidance advice in TR 24772-1 Section 6.32.

6.33 Dangling References to Stack Frames [DCM]

6.33.1 Applicability to language

In Ada, the attribute 'Address yields a value of some system-specific type that is not equivalent to a pointer. The attribute 'Access provides an access value (what other languages call a pointer). Addresses and access values are not automatically convertible, although a predefined set of generic functions can be used to convert one into the other. Access values are typed, that is to say, they can only designate objects of a particular type or class of types.

As in other languages, it is possible to apply the 'Address attribute to a local variable, and to make use of the resulting value outside of the lifetime of the variable. However, 'Address is very rarely used in this fashion in Ada. Most commonly, programs use 'Access to designate objects and subprograms, and the language enforces accessibility checks whenever code attempts to use this attribute to provide access to a local object outside of its scope. These accessibility checks eliminate the possibility of dangling references.

As for all other language-defined checks, accessibility checks can be disabled over any portion of a program by using the Suppress **pragma**. The attribute `Unchecked_Access` produces values that are exempt from accessibility checks.

6.33.2 Guidance to language users

- Only use the 'Address attribute on static objects (for example, a register address).
- Do not use 'Address to provide indirect untyped access to an object.
- Do not convert between 'Address and access types.
- Use access types in all circumstances when indirect access is needed.
- Do not suppress accessibility checks.
- Avoid use of the attribute `Unchecked_Access`.
- Use 'Access attribute in preference to 'Address.
- Consider applying the restriction `No_Use_Of_Attribute(Address)` to enforce that 'Address is not used.
- Consider applying the restriction `No_Unchecked_Access` to enforce that `Unchecked_Access` is not used.

6.34 Subprogram Signature Mismatch [OTR]

6.34.1 Applicability to language

There are two concerns identified with this vulnerability. The first is the corruption of the execution stack due to the incorrect number or type of actual parameters. The second is the corruption of the execution stack due to calls to externally compiled modules. Ada does not support variadic subprograms, which eliminates a common source for this vulnerability.

In Ada, at compilation time, the parameter association is checked to ensure that the type of each actual parameter matches the type of the corresponding formal parameter. In addition, the formal parameter specification may include default expressions for a parameter. Hence, the procedure may be called with some actual parameters missing. In this case, if there is a default expression for the missing parameter, then the call will be compiled without any errors. If default expressions are not specified, then the procedure call with insufficient actual parameters will be flagged as an error at compilation time.

Caution must be used when specifying default expressions for formal parameters, as their use may result in successful compilation of subprogram calls with an incorrect signature. The execution stack will not be corrupted in this event but the program may be executing with unexpected values. The most appropriate use of default expressions is when, without them, there would end up being an overloading of the same name with fewer parameters that performed essentially the same operation. When calling externally compiled modules that are Ada program units, the type matching and subprogram interface signatures are monitored and checked as part of the compilation and linking of the full application. When calling externally compiled modules in other programming languages, additional steps are needed to ensure that the number and types of the parameters for these external modules are correct.

6.34.2 Guidance to language users

- Minimize the use of default expressions for formal parameters.
- Manage interfaces between Ada program units and program units in other languages by using **pragma** Import to specify subprograms that are defined externally and **pragma** Export to specify subprograms that are used externally. These **pragmas** specify the imported and exported aspects of the subprograms, this includes the calling convention. All parameters need to be specified when using **pragma** Import and **pragma** Export.
- Use the **pragma** Convention to identify when an Ada entity should use the calling conventions of a different programming language facilitating the correct usage of the execution stack when interfacing with other programming languages.
- Use the Valid attribute to check if an object that is part of an interface with another language has a valid value for its type.

6.35 Recursion [GDL]

6.35.1 Applicability to language

Ada permits recursion. The exception `Storage_Error` is raised when the recurring execution results in insufficient storage.

6.35.2 Guidance to language users

- If recursion is used, then use a `Storage_Error` exception handler to handle insufficient storage due to recurring execution.
- Use the asynchronous control construct to time the execution of a recurring call and to terminate the call if the time limit is exceeded.
- Alternatively, monitor the depth of the recursion such as by passing a recursion depth value that is incremented for each level of recursion, and use a subtype constraint or explicit comparison against a maximum depth limit to trigger handling of the situation.
- Consider applying the restriction `No_Recursion` or `No_Reentrancy` to eliminate this vulnerability.

6.36 Ignored Error Status and Unhandled Exceptions [OYB]

6.36.1 Applicability to language

Ada offers a set of predefined exceptions for error conditions that may be detected by checks that are compiled into a program. In addition, the programmer may define exceptions that are appropriate for their application. These exceptions are handled using an exception handler. Exceptions may be handled in the environment where the exception occurs or may be propagated out to an enclosing scope.

6.36.2 Guidance to language users

- In addition to the mitigations defined in TR 24772-1, values delivered to an Ada program from an external device may be checked for validity prior to being used. Use the result of the Valid attribute for this purpose.
- Consider using the call `Ada.Task_Termination.Set_Dependents_Fallback_Handler` to install a handler that will be invoked whenever a task terminates.

6.37 Fault Tolerance and Failure Strategies [REW]

6.37.1 Applicability to language

An Ada system that consists of multiple tasks is subject to the same hazards as multithreaded systems in other languages. A task that fails, for example, because its execution violates a language-defined check, terminates quietly, unless a Termination handler has been established using the `Ada.Task_Termination` package.

Any other task that attempts to communicate with a terminated task will receive the exception `Tasking_Error`. The undisciplined use of the **abort** statement or the asynchronous transfer of control feature may destroy the functionality of a multitasking program.

6.37.2 Guidance to language users

- Include exception handlers for every task, so that their unexpected termination can be handled and possibly communicated to the execution environment, or establish a Termination handler for all tasks. For high-integrity systems, exception handling is often forbidden. However, a top-level exception handler or Termination handler can be used to restore the overall system to a coherent state. Use objects of controlled types to ensure that resources are properly released if a task terminates unexpectedly.
- Use the **abort** statement sparingly, if at all.
- Define interrupt handlers to handle signals that come from the hardware or the operating system. This mechanism can also be used to add robustness to a concurrent program.
- Make use of the `Ada.Task_Termination` package (defined in the Systems Programming Annex of the Ada Reference Manual) to monitor task termination and its causes.
- Make use of the various **pragmas**, restrictions, and other language features defined in the High Integrity Systems Annex of the Ada Reference Manual when writing systems for high-reliability applications. For example, the **pragma** `Detect_Blocking` can be used to ensure detection of a potentially blocking operation occurring within a protected operation, and to raise an exception in that case.

6.38 Type-breaking Reinterpretation of Data [AMV]

6.38.1 Applicability to language

`Unchecked_Conversion` can be used to bypass the type-checking rules, and its use is thus unsafe, as is its equivalent in any other language. The same applies to the use of `Unchecked_Union`, even though the language specifies various inference rules that the compiler must use to catch statically detectable constraint violations. The fact that `Unchecked_Conversion` is a generic function that must be instantiated explicitly (and given a meaningful name) hinders its undisciplined use, and places a loud marker in the code wherever it is used. Well-written Ada code will have a small set of instantiations of `Unchecked_Conversion`. Most implementations require the source and target types to have the same size in bits, to prevent accidental truncation or missing sign extensions.

Type reinterpretation is a universal programming need, and no usable programming language can exist without some mechanism that bypasses the type model. Ada provides these mechanisms with some additional safeguards, and makes their use purposely verbose, to alert the writer and the reader of a program to the presence of an unchecked operation.

6.38.2 Guidance to language users

- Use `Unchecked_Union` only in multi-language programs that need to communicate data between Ada and C or C++. Otherwise the use of discriminated types prevents "punning" between values of two distinct types that happen to share storage.

- Avoid using address clauses to obtain overlays. If the types of the objects are the same, then a renaming declaration is preferable. Otherwise, use the **pragma Import** to inhibit the initialization of one of the entities so that it does not interfere with the initialization of the other one.
- Consider applying the restrictions `No_Use_Of_Pragma(Unchecked_Union)`, `No_Use_Of_Aspect(Unchecked_Union)`, `No_Use_Of_Attribute(Address)`, and `No_Unchecked_Conversion` to ensure this vulnerability cannot arise.

6.39 Memory Leak [XYL]

6.39.1 Applicability to language

For objects that are allocated from the heap without the use of reference counting, the memory leak vulnerability is possible in Ada. For objects that must allocate from a storage pool, the vulnerability is present but is restricted to this single pool, which makes it easier to detect memory leaks by verification. Subpools may be used to further reduce the possibility for memory leaks. For objects of a controlled type that uses referencing counting and that are not part of a cyclic reference structure, the vulnerability does not exist.

Ada ensures that objects designated by an access type declared in a nested scope are finalized when execution leaves the nested scope, however, it is implementation defined whether storage is reclaimed for this case. Associating an access type with a storage pool can ensure that the storage reclamation takes place.

Ada does not mandate the use of a garbage collector, but Ada implementations are free to provide such memory reclamation. For applications that use and return memory on an implementation that provides garbage collection, the issues associated with garbage collection exist in Ada.

6.39.2 Guidance to language users

- Use storage pools and subpools where possible.
- Use controlled types and reference counting to implement explicit storage management systems that cannot have storage leaks.
- Declare access types in a nested scope where possible.
- Use a completely static model where all storage is allocated from global memory and explicitly managed under program control.

6.40 Templates and Generics [SYM]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as the Ada generics model is based on imposing a contract on the structure and operations of the types that can be used for instantiation. Also, explicit instantiation of the generic is required for each particular type.

Therefore, the compiler is able to check the generic body for programming errors, independently of actual instantiations. At each actual instantiation, the compiler will also check that the instantiated type meets all the requirements of the generic contract.

Ada also does not allow for ‘special case’ generics for a particular type, therefore behaviour is consistent for all instantiations.

6.41 Inheritance [RIP]

6.41.1 Applicability to language

The vulnerability documented in TR 24772-1 Section 6.41 applies to Ada.

Ada allows only a restricted form of multiple inheritance, where only one of the multiple ancestors (the parent) may implement operations. All other ancestors (interfaces) can only specify the operations' signature. Therefore, Ada does not suffer from multiple inheritance related vulnerabilities.

6.41.2 Guidance to language users

- Use the overriding indicators on potentially inherited subprograms to ensure that the intended contract is obeyed, thus preventing the accidental redefinition or failure to redefine an operation of the parent.
- Use the mechanisms of mitigation described in TR 24772-1.

6.42 Extra Intrinsic [LRM]

The vulnerability does not apply to Ada, because all subprograms, whether intrinsic or not, belong to the same name space. This means that all subprograms must be explicitly declared, and the same name resolution rules apply to all of them, whether they are predefined or user-defined. If two subprograms with the same name and signature are visible (that is to say nameable) at the same place in a program, then a call using that name will be rejected as ambiguous by the compiler, and the programmer will have to specify (for example, by means of an expanded name) which subprogram is meant.

6.43 Argument Passing to Library Functions [TRJ]

6.43.1 Applicability to language

The general vulnerability that parameters might have values precluded by preconditions of the called routine applies to Ada as well.

However, to the extent that the preclusion of values can be expressed as part of the type system of Ada, the preconditions are checked by the compiler statically or dynamically and thus are no longer vulnerabilities. For example, any range constraint on values of a parameter can be expressed in Ada by means of type or subtype declarations. Type violations are detected at compile time, subtype violations cause run-time exceptions. In addition, preconditions, postconditions, type invariants, and subtype predicates can be specified explicitly to express more complex restrictions to be observed by callers. These are checked at run-time depending on the Assertion_Policy in effect, and can be recognized by other static analysis tools as part of program verification.

6.43.2 Guidance to language users

- Exploit the type and subtype system of Ada to express restrictions on the values of parameters and results.
- Specify explicit preconditions and postconditions for subprograms wherever practical.
- Specify subtype predicates and type invariants for subtypes and private types when appropriate.
- Specify the exception raised or other response to values that do not satisfy the precondition.

6.44 Inter-language Calling [DJS]

6.44.1 Applicability to Language

The vulnerability applies to Ada, however Ada provides mechanisms to interface with common languages, such as C, C++, Fortran and COBOL, so that vulnerabilities associated with interfacing with these languages can be avoided.

6.44.2 Guidance to Language Users

- Use the inter-language methods and syntax specified by the Ada Reference Manual when the routines to be called are written in languages that the ARM specifies an interface with.
- Use interfaces to the C programming language where the other language system(s) are not covered by the ARM, but the other language systems have interfacing to C.
- Make explicit checks on all return values from foreign system code artifacts, for example by using the 'Valid attribute or by performing explicit tests to ensure that values returned by inter-language calls conform to the expected representation and semantics of the Ada application.
- 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.

6.45 Dynamically-linked Code and Self-modifying Code [NYY]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as Ada supports neither dynamic linking nor self-modifying code. The latter is possible only by exploiting other vulnerabilities of the language in the most malicious ways and even then it is still very difficult to achieve.

6.46 Library Signature [NSQ]

6.46.1 Applicability to language

Ada provides mechanisms to explicitly interface to modules written in other languages. **Pragmas** Import, Export and Convention permit the name of the external unit and the interfacing convention to be specified.

Even with the use of **pragma** Import, **pragma** Export and **pragma** Convention the vulnerabilities stated in Section 6.46 are possible. Names and number of parameters change under maintenance; calling conventions change as compilers are updated or replaced, and languages for which Ada does not specify a calling convention may be used.

6.46.2 Guidance to language users

- The mitigation mechanisms of TR 24772-1 Section 6.46.5 are applicable.

6.47 Unanticipated Exceptions from Library Routines [HJW]

6.47.1 Applicability to language

Ada programs are capable of handling exceptions at any level in the program, as long as any exception naming and delivery mechanisms are compatible between the Ada program and the library components. In such cases the normal Ada exception handling processes will apply, and either the calling unit or some subprogram or task in its

call chain will catch the exception and take appropriate programmed action. If no action is taken to handle the exception, the task or program where the exception occurred will terminate.

If the library convention is to report error codes and not by exceptions, then, if the library components themselves are written in Ada, then Ada's exception handling mechanisms let all called units trap any exceptions that are generated and return error conditions instead. If such exception handling mechanisms are not put in place, then exceptions can be unexpectedly delivered to a caller.

If the interface between the Ada units and the library routine being called does not adequately address the issue of naming, generation and delivery of exceptions across the interface, then the vulnerabilities as expressed in TR 24772-1 Section 6.47 apply.

6.47.2 Guidance to language users

- Ensure that the interfaces with libraries written in other languages are compatible in the naming and generation of exceptions.
- Put appropriate exception handlers in all routines that call library routines, including the catch-all exception handler **when others =>**.
- Put appropriate exception handlers in all routines that are called by library routines, including the catch-all exception handler **when others =>**.
- Document any exceptions that may be raised by any Ada units being used as library routines.

6.48 Pre-Processor Directives [NMP]

This vulnerability is not applicable to Ada as Ada does not have a pre-processor.

6.49 Suppression of Language-defined Run-time Checking [MXB]

6.49.1 Applicability to Language

The vulnerability exists in Ada since **pragma Suppress** permits explicit suppression of language-defined checks on a unit-by-unit basis or on partitions or programs as a whole. (The language-defined default, however, is to perform the runtime checks that prevent the runtime vulnerabilities.) **Pragma Suppress** can suppress all language-defined checks or 12 individual categories of checks (see Section 11.5 of the Ada language reference manual).

6.49.2 Guidance to Language Users

- Do not suppress language defined checks.
- If language-defined checks must be suppressed, use static analysis to prove that the code is correct for all combinations of inputs.
- If language-defined checks must be suppressed, use explicit checks at appropriate places in the code to ensure that errors are detected before any processing that relies on the correct values.

6.50 Provision of Inherently Unsafe Operations [SKL]

6.50.1 Applicability to Language

In recognition of the occasional need to step outside the type system or to perform “risky” operations, Ada provides clearly identified language features to do so. Examples include the generic `Unchecked_Conversion` for

unsafe type-conversions or `Unchecked_Deallocation` for the deallocation of heap objects regardless of the existence of surviving references to the object. If unsafe programming is employed in a unit, then the unit needs to specify the respective generic unit in its context clause, thus identifying potentially unsafe units. Similarly, there are ways to create a potentially unsafe global pointer to a local object, using the `Unchecked_Access` attribute.

6.50.2 Guidance to language users

- Avoid the use of unsafe programming practices.
- Use the `Restrictions` pragma to prevent the inadvertent use of unsafe language constructs.
- Carefully scrutinize any code that refers to a program unit explicitly designated to provide unchecked operations.

6.51 Obscure Language Features [BRS]

6.51.1 Applicability to language

Ada is a rich language and provides facilities for a wide range of application areas. Because some areas are specialized, it is likely that a programmer not versed in a special area might misuse features for that area. For example, the use of tasking features for concurrent programming requires knowledge of this domain. Similarly, the use of exceptions and exception propagation and handling requires a deeper understanding of control flow issues than some programmers possess.

6.51.2 Guidance to language users

- Use the `pragma Restrictions` to prevent the use of obscure features of the language.
- Similarly, avoid features in a Specialized Needs Annex of the Ada language reference manual unless the application area concerned is well-understood.
- The restriction `No_Dependence` prevents the use of specified pre-defined or user-defined libraries.

6.52 Unspecified Behaviour [BQF]

6.52.1 Applicability to language

In Ada, there are two main categories of unspecified behaviour, one having to do with unspecified aspects of normal run-time behaviour, and one having to do with *bounded errors*, errors that need not be detected at run-time but for which there is a limited number of possible run-time effects (though always including the possibility of raising `Program_Error` exception).

For the normal behaviour category, there are several distinct aspects of run-time behaviour that might be unspecified, including:

- Order in which certain actions are performed at run-time;
- Number of times a given element operation is performed within an operation invoked on a composite or container object;
- Results of certain operations within a language-defined generic package if the actual associated with a particular formal subprogram does not meet stated expectations (such as “<” providing a strict weak ordering relationship);
- Whether distinct instantiations of a generic or distinct invocations of an operation produce distinct values for tags or access-to-subprogram values.

The index entry in the Ada language reference manual for *unspecified* provides the full list. Similarly, the index entry for *bounded error* provides the full list of references to places in the Ada language reference manual where a bounded error is described.

Failure can occur due to unspecified behaviour when the programmer did not fully account for the possible outcomes, and the program is executed in a context where the actual outcome was not one of those handled, resulting in the program producing an unintended result.

6.52.2 Guidance to language users

As in any language, the vulnerability can be reduced in Ada by avoiding situations that have unspecified behaviour, or by fully accounting for the possible outcomes.

Particular instances of this vulnerability can be avoided or mitigated in Ada in the following ways:

- For situation where order of evaluation or number of evaluations is unspecified, use only operations with no side-effects, or idempotent behaviour, to avoid the vulnerability;
- For situations involving generic formal subprograms, ensure that the actual subprogram satisfies all of the stated expectations;
- For situations involving unspecified values, avoid depending on equality between potentially distinct values;
- For situations involving bounded errors, avoid the problem completely, by ensuring in other ways that all requirements for correct operation are satisfied before invoking an operation that might result in a bounded error. See [6.22 Initialization of Variables \[LAV\]](#) for a discussion of uninitialized variables in Ada, a common cause of a bounded error.

6.53 Undefined Behaviour [EWF]

6.53.1 Applicability to language

In Ada, undefined behaviour is called *erroneous execution*, and can arise from certain errors that are not required to be detected by the implementation, and whose effects are not in general predictable.

There are various kinds of errors that can lead to erroneous execution, including:

- Changing a discriminant of a record (by assigning to the record as a whole) while there remain active references to subcomponents of the record that depend on the discriminant;
- Referring via an access value, task id, or tag, to an object, task, or type that no longer exists at the time of the reference;
- Referring to an object whose assignment was disrupted by an abort statement, prior to invoking a new assignment to the object;
- Sharing an object between multiple tasks without adequate synchronization;
- Suppressing a language-defined check that is in fact violated at run-time;
- Specifying the address or alignment of an object in an inappropriate way;
- Using `Unchecked_Conversion`, `Address_To_Access_Conversions`, or calling an imported subprogram to create a value, or reference to a value, that has an *abnormal* representation.

The full list is given in the index of the Ada language reference manual under *erroneous execution*.

Any occurrence of erroneous execution represents a failure situation, as the results are unpredictable, and may involve overwriting of memory, jumping to unintended locations within memory, and other uncontrolled events.

6.53.2 Guidance to language users

The common errors that result in erroneous execution can be avoided in the following ways:

- Ensure that all data shared between tasks are either private within a protected object or marked Atomic;
- Upon any use of `Unchecked_Deallocation`, carefully check to be sure that there are no remaining references to the object;
- Use `pragma Suppress` sparingly, and only after the code has undergone extensive verification.

The other errors that can lead to erroneous execution are less common, but clearly in any given Ada application, care must be taken when using features such as:

- `abort`;
- `Unchecked_Conversion`;
- `Address_To_Access_Conversions`;
- The results of imported subprograms;
- Discriminant-changing assignments to global variables.

The mitigations described in TR 24772-1 Section 6.55.5 are applicable here.

6.54 Implementation-Defined Behaviour [FAB]

6.54.1 Applicability to language

There are a number of situations in Ada where the language semantics are implementation defined, to allow the implementation to choose an efficient mechanism, or to match the capabilities of the target environment. Each of these situations is identified in Annex M of the Ada language reference manual, and implementations are required to provide documentation associated with each item in Annex M to provide the programmer with guidance on the implementation choices.

A failure can occur in an Ada application due to implementation-defined behaviour if the programmer presumed the implementation made one choice, when in fact it made a different choice that affected the results of the execution. In many cases, a compile-time message or a run-time exception will indicate the presence of such a problem. For example, the range of integers supported by a given compiler is implementation defined. However, if the programmer specifies a range for an integer type that exceeds that supported by the implementation, then a compile-time error will be indicated, and if at run time a computation exceeds the base range of an integer type, then a `Constraint_Error` is raised.

Failure due to implementation-defined behaviour is generally due to the programmer presuming a particular effect that is not matched by the choice made by the implementation. As indicated above, many such failures are indicated by compile-time error messages or run-time exceptions. However, there are cases where the implementation-defined behaviour might be silently misconstrued, such as if the implementation presumes `Ada.Exceptions.Exception_Information` returns a string with a particular format, when in fact the implementation does not use the expected format. If a program is attempting to extract information from `Exception_Information` for the purposes of logging propagated exceptions, then the log might end up with misleading or useless information if there is a mismatch between the programmer's expectation and the actual implementation-defined format.

Many implementation-defined limits have associated constants declared in language-defined packages, generally `package System`. In particular, the maximum range of integers is given by `System.Min_Int .. System.Max_Int`, and other limits are indicated by constants such as `System.Max_Binary_Modulus`, `System.Memory_Size`,

System.Max_Mantissa, and similar. Other implementation-defined limits are implicit in normal 'First and 'Last attributes of language-defined (sub) types, such as System.Priority'First and System.Priority'Last. Furthermore, the implementation-defined representation aspects of types and subtypes can be queried by language-defined attributes. Thus, code can be parameterized to adjust to implementation-defined properties without modifying the code.

6.54.2 Guidance to language users

- Be aware of the contents of Annex M of the Ada language reference manual and avoid implementation-defined behaviour whenever possible.
- Make use of the constants and subtype attributes provided in package System and elsewhere to avoid exceeding implementation-defined limits.
- Minimize use of any predefined numeric types, as the ranges and precisions of these are all implementation defined. Instead, declare your own numeric types to match your particular application needs.
- When there are implementation-defined formats for strings, such as Exception_Information, localize any necessary processing in packages with implementation-specific variants.

6.55 Deprecated Language Features [MEM]

6.55.1 Applicability to language

If obsolescent language features are used, then the mechanism of failure for the vulnerability is as described in Section 6.55.3 of TR 24772-1.

6.55.2 Guidance to language users

- Use **pragma Restrictions** (No_Obsolescent_Features) to prevent the use of any obsolescent features.
- Refer to Annex J of the Ada language reference manual to determine whether a feature is obsolescent.

6.56 Concurrency – Activation [CGA]

6.56.1 Applicability to language

Ada is open to this vulnerability but provides features for its mitigation. A task failing during activation will always raise an exception in the activating task (e.g., Tasking_Error). The activating task does not continue executing until all its dependent tasks have completed activation. A task can always check that another task is executable (i.e., not terminated).

6.56.2 Guidance to language users

- Always have a handler to catch activation failures.
- If possible declare all tasks statically at the library level.

6.57 Concurrency – Directed termination [CGT]

6.57.1 Applicability to language

Ada defines abort-deferred regions in which task termination will not occur. On a single processor, abort is defined to be immediate if the task is not in such a region. On multiprocessors abort may not be immediate but will be before any synchronization (dispatching) point.

6.57.2 Guidance to language users

- Use the ‘Terminated and ‘Callable attributes to check that a task has terminated.
- Minimize the size of any abort-deferred region.
- Remove any possibility of unbounded loops in abort-deferred regions.
- Where possible do not use forced termination (**abort**), or apply the restriction No_Abort_Statements to eliminate the use of this construct.

6.58 Concurrent Data Access [CGX]

6.58.1 Applicability to language

Ada does allow tasks to access unprotected shared variables. However the standard means of programming data that is shared between tasks is to use a protected object that enforces serial access. Atomic updates on some simple types are supported (if supported by the hardware).

6.58.2 Guidance to language users

- When possible, use protected objects for shared data.
- Statically determine that no unprotected data is used directly by more than one task.
- When shared variables are used, employ model checking or equivalent methodologies to prove the absence of race conditions.

6.59 Concurrency – Premature Termination [CGS]

6.59.1 Applicability to language

An Ada task can terminate silently, however in general the tasking model is robust and a number of features are available to mitigate against this vulnerability – see guidance below.

6.59.2 Guidance to language users

- If possible, do not use the abort feature, or apply the restriction No_Abort_Statements to eliminate the use of this construct.
- All tasks should contain an exception handler at the outer level to prevent silent termination due to unhandled exceptions.
- Make use of package Ada.Task_Termination to force a handler to be executed when a task terminates.
- Use attributes ‘Terminated and ‘Callable to confirm that a task has not terminated (although care is needed here as a task could terminate immediately after this call is made).
- Place all data that would be vulnerable to premature task termination in an abort-deferred region (e.g., a protected object).
- Make used of timed task communication that will time-out if the called task does not respond.

6.60 Protocol Lock Errors [CGM]

6.60.1 Applicability to language

Ada is open to the errors identified in this vulnerability but supports a number of features that aid mitigation – see guidance below.

6.60.2 Guidance to language users

- Make use of loosely coupled, non-blocking communication using protected objects; on a single processor using a scheduling regime based on ceiling protocols, this is guaranteed to be deadlock free (if the tasks and protected objects are assigned the correct priorities – a static property that can be checked offline).
- For multicore, consider assigning all interacting tasks to the same CPU then treat each such group as a separate independent entity.
- Minimize the use of dynamic priorities and dynamic ceiling priorities (so that the static values can be verified).
- Where possible stay within the constraints defined by the Ravenscar profile [17].
- If synchronous communication (rendezvous) is employed, use model checking or equivalent to prove that the program is free from deadlocks etc.
- Always handle exceptions that can arrive from rendezvous or protected objects (unless they can be proved to not be raised).
- Guard against protocol failures by using timed communication, watchdog timers (programmed using Ada's timed events) and time-stamped data (using the Ada's clock facilities).

Do not use unprotected shared data for synchronization between tasks.

6.61 Uncontrolled Format String [SHL]

With the exception of unsafe programming (see [4 Language concepts](#)), this vulnerability is not applicable to Ada as Ada does not allow undefined string lengths (see [6.7 String Termination \[CJM\]](#)).

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 \[SAM\]](#) and [6.52 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 [6.53 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 [6.54 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 [6.39 Memory Leak \[XYL\]](#)).
- Ada could provide mechanisms to prevent further extensions of a type hierarchy (see [6.41 Inheritance \[RIP\]](#)).
- Ada standardization committees can work with other programming language standardization committees to define library interfaces that include more than a program calling interface. In particular, mechanisms to qualify and quantify ranges of behaviour, such as preconditions, postconditions and type invariants, would be helpful (see [6.46 Library Signature \[NSQ\]](#)).

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] Barnes, John, *High Integrity Software - the SPARK Approach to Safety and Security*. Addison-Wesley. 2002.
- [24] Barnes, John, [Lecture Notes on Computer Science 8338](#), “Ada 2012 Rationale: The Language—The Standard Libraries,” Springer, 2013.
- [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

- Abnormal representation, 9
- abort**, 25, 31, 37, 39, 40
- Access object, 9
- Access type, 9
- Access value, 10
- Access-to-subprogram, 9
- Allocator, 10
- AMV – Type-breaking Reinterpretation of Data, 31
- Aspect specification, 10
- Atomic, 10, 12, 17, 37, 39
- Attribute, 10
 - 'Access, 20, 29
 - 'Callable, 39, 40
 - 'Terminated, 39, 40
 - 'Valid, 16, 24
 - 'Valid, 24
 - 'Access, 28, 29
 - 'Address, 28, 29, 41
 - 'Alignment, 13
 - 'Component_Size, 13
 - 'Exponent, 17
 - 'First, 27, 28, 38
 - 'Image, 26
 - 'Last, 28, 38
 - 'Length, 27, 28
 - 'Range, 28
 - 'Size, 13
 - 'Unchecked_Access, 15, 29, 35
 - 'Valid, 34
- Bit ordering, 10
- BJL – Namespace Issues, 23
- Bounded Error, 10
- BQF – Unspecified Behaviour, 36
- BRS – Obscure Language Features, 35
- Case choices, 10
- Case expression, 10
- Case statement, 10, 18, 26
- CCB – Enumerator Issues, 17
- CGA – Concurrency – Activation, 39
- CGM – Protocol Lock Errors, 40
- CGS – Concurrency – Premature Termination, 40
- CGT – Concurrency – Directed termination, 39
- CGX – Concurrent Data Access, 39
- CJM – String Termination, 19
- CLL – Switch Statements and Static Analysis, 26
- Compilation unit, 10
- Configuration pragma, 10, 14
- Controlled type, 10
- CSJ – Passing Parameters and Return Values, 28
- DCM – Dangling References to Stack Frames, 28
- Dead store, 10
- Default expression, 10
- Discrete type, 10
- Discriminant, 10, 37
- DJS – Inter-language Calling, 33
- Endianness, 10
- Enumeration Representation Clause, 10
- Enumeration type, 11, 12
- EOJ – Demarcation of Control Flow, 27
- Erroneous execution, 11
- EWD – Structured Programming, 28
- EWF – Undefined Behaviour, 37
- Exception, 11, 12, 13, 14, 16, 18, 19, 23, 24, 27, 30, 31, 33, 34, 35, 36, 38, 39, 40
 - Constraint_Error, 12, 13, 19, 21, 26, 38
 - Program_Error, 12, 13, 36
 - Storage_Error, 12, 30
 - Tasking_Error, 12, 31, 39
- Exception Information, 38
- Expanded name, 11
- Explicit conversions, 13, 16
- FAB – Implementation-Defined Behaviour, 38
- FIF – Arithmetic Wrap-around Error, 21
- Fixed-point types, 11
- FLC – Numeric Conversion Errors, 18
- GDL – Recursion, 30
- Generic formal subprogram, 11
- HCB – Buffer Boundary Violation (Buffer Overflow), 19
- HFC – Pointer Type Conversions, 19
- Hiding, 11, 12, 41
 - hidden from all visibility, 12
 - hidden from direct visibility, 12
- HJW – Unanticipated Exceptions from Library Routines, 34
- Homograph, 11
- Idempotent behaviour, 11
- Identifier, 11

- Identifier length, 21
- IHN–Type System, 16
- Implementation defined, 11, 12
- Implicit conversions, 13, 16
- International character sets, 21

- JCW – Operator Precedence/Order of Evaluation, 24
- Junk initialization, 24

- KOA – Likely Incorrect Expression, 25

- Language concepts, 12, 19, 20, 21, 26, 27, 32, 34, 41
- Language Vulnerabilities
 - Argument Passing to Library Functions [TRJ], 33
 - Arithmetic Wrap-around Error [FIF], 21
 - Bit Representation [STR], 16
 - Buffer Boundary Violation (Buffer Overflow) [HCB], 19
 - Choice of Clear Names [NAI], 21
 - Concurrency – Activation [CGA], 39
 - Concurrency – Directed termination [CGT], 39
 - Concurrency – Premature Termination [CGS], 40
 - Concurrent Data Access [CGX], 39
 - Dangling Reference to Heap [XYK], 20
 - Dangling References to Stack Frames [DCM], 28
 - Dead and Deactivated Code [XYQ], 26
 - Dead store [WXQ], 22
 - Demarcation of Control Flow [EOJ], 27
 - Deprecated Language Features [MEM], 39
 - Dynamically-linked Code and Self-modifying Code [NYY], 34
 - Enumerator Issues [CCB], 17
 - Extra Intrinsic [LRM], 33
 - Fault Tolerance and Failure Strategies [REW], 30
 - Floating-point Arithmetic [PLF], 17
 - Identifier Name Reuse [YOW], 22
 - Ignored Error Status and Unhandled Exceptions [OYB], 30
 - Implementation-Defined Behaviour [FAB], 38
 - Inheritance [RIP], 32
 - Initialization of Variables [LAV], 23
 - Inter-language Calling [DJS], 33
 - Library Signature [NSQ], 34
 - Likely Incorrect Expression [KOA], 25
 - Loop Control Variables [TEX], 27
 - Memory Leak [XYL], 32
 - Namespace Issues [BJL], 23
 - Numeric Conversion Errors [FLC], 18
 - Obscure Language Features [BRS], 35
 - Off-by-one Error [XZH], 27
 - Operator Precedence/Order of Evaluation [JCW], 24
 - Passing Parameters and Return Values [CSJ], 28
 - Pointer Arithmetic [RVG], 20
 - Pointer Type Conversions [HFC], 19
 - Protocol Lock Errors [CGM], 40
 - Provision of Inherently Unsafe Operations [SKL], 35
 - Recursion [GDL], 30
 - Side-effects and Order of Evaluation [SAM], 24
 - String Termination [CJM], 19
 - Structured Programming [EWD], 28
 - Subprogram Signature Mismatch [OTR], 29
 - Suppression of Language-defined Run-time Checking [MXB], 35
 - Switch Statements and Static Analysis [CLL], 26
 - Templates and Generics [SYM], 32
 - Type System [IHN], 16
 - Type-breaking Reinterpretation of Data [AMV], 31
 - Unanticipated Exceptions from Library Routines [HJW], 34
 - Unchecked Array Indexing [XYZ], 19
 - Uncontrolled Format String [SHL], 41
 - Undefined Behaviour [EWF], 37
 - Unspecified Behaviour [BQF], 36
 - Unused Variable [YZS], 22
 - Using Shift Operations for Multiplication and Division [PIK], 21
- Language Vulnerability
 - Unchecked Array Copying [XYW], 19
- LAV – Initialization of Variables, 23
- LRM – Extra Intrinsic, 33

- MEM – Deprecated Language Features, 39
- Mixed casing, 21
- Modular type, 11
- MXB – Suppression of Language-defined Run-time Checking, 35

- NAI – Choice of Clear Names, 21
- NSQ – Library Signature, 34
- NYY – Dynamically-linked Code and Self-modifying Code, 34

- Obsolescent features, 11
- Operational and Representation Attributes, 11, 13
- OTR – Subprogram Signature Mismatch, 29
- Overriding indicators, 11
- OYB – Ignored Error Status and Unhandled Exceptions, 30

- Partition, 11
- PIK – Using Shift Operations for Multiplication and Division, 21
- PLF – Floating-point Arithmetic, 17
- Pointer, 11, 23
- Polymorphic Variable, 13

Postconditions, 33, 41
 Pragma, 11, 35
 Configuration pragma, 10
 pragma Atomic, 13, 17
 pragma Atomic_Components, 13, 17
 pragma Convention, 13, 30, 34
 pragma Default_Storage_Pool, 15
 pragma Detect_Blocking, 13
 pragma Discard_Names, 13
 pragma Export, 14, 30, 34
 pragma Import, 14, 30, 31, 34
 pragma Normalize_Scalars, 14, 23, 24
 pragma Pack, 14
 pragma Restrictions, 14, 15, 36, 39, 41
 pragma Suppress, 14, 15, 19, 35, 37
 pragma Unchecked Union, 14
 pragma Volatile, 14, 17
 pragma Volatile_Components, 14, 17
 Preconditions, 33, 41
 Program verification, 33

 Range check, 12
 Record Representation Clauses, 12
 REU – Termination Strategy, 30
 REW – Fault Tolerance and Failure Strategies, 30
 RIP – Inheritance, 32
 RVG – Pointer Arithmetic, 20

 SAM – Side-effects and Order of Evaluation, 24
 Scalar type, 12
 Separate Compilation, 14
 SHL – Uncontrolled Format String, 41
 Singular/plural forms, 21
 SKL – Provision of Inherently Unsafe Operations, 35
 Static expressions, 12
 Storage Place Attributes, 12

 Storage pool, 10, 12, 14, 15, 32
 Storage subpool, 12, 14, 32
 STR – Bit Representation, 16
 Subtype declaration, 12
 SYM – Templates and Generics, 32
 Symbols and conventions, 9

 Task, 12, 30, 31, 40
 Terms and definitions, 9
 TEX – Loop Control Variables, 27
 TRJ – Argument Passing to Library Functions, 33
 Type conversion, 12, 13, 19
 Type invariants, 33, 41

 Unchecked conversions, 13, 16
 Unchecked_Conversion, 13, 15, 16, 31, 35, 37
 Underscores and periods, 21
 Unsafe Programming, 15, 18, 19, 20, 21, 26, 27, 32, 34, 35, 41
 Unused variable, 12
 User-defined floating-point types, 15
 User-defined scalar types, 15

 Volatile, 12, 17, 22

 WXQ – Dead store, 22

 XYK – Dangling Reference to Heap, 20
 XYL – Memory Leak, 32
 XYQ – Dead and Deactivated Code, 26
 XYW – Unchecked Array Copying, 19
 XYZ – Unchecked Array Indexing, 19
 XZH – Off-by-one Error, 27

 YOW – Identifier Name Reuse, 22
 YZS – Unused Variable, 22