ISO/IEC JTC 1/SC 22/WG 23 N 0288

Revised draft language-specific annex for Ada

Date	10-December-2010
Contributed by	John Benito
Original file name	
Notes	Replaces N0258

Annex Ada

(informative)

Ada. Vulnerability descriptions for the language Ada

Ada.3.1.0 Status and history

20100619 WG9

Every vulnerability description of Clause 6 of the main document is addressed in the annex in the same order even if there is simply a note that it is not relevant to Ada.

This Annex specifies the characteristics of the Ada programming language that are related to the vulnerabilities defined in this Technical Report. When applicable, the techniques to mitigate the vulnerability in Ada applications are described in the associated section on the vulnerability.

20-Oct-2010 – Benito

Coerced the text into the outline adopted at meeting #15.

06-Dec-2010 — Benito

Fixed format and layout issues.

Ada.1 Identification of standards and associated documentation

ISO/IEC 8652:1995 Information Technology – Programming Languages—Ada.
ISO/IEC 8652:1995/COR.1:2001, Technical Corrigendum to Information Technology – Programming Languages—Ada.
ISO/IEC 8652:1995/AMD.1:2007, Amendment to Information Technology – Programming Languages—Ada.
ISO/IEC TR 15942:2000, Guidance for the Use of Ada in High Integrity Systems.
ISO/IEC TR 24718:2005, Guide for the use of the Ada Ravenscar Profile in high integrity systems.
Lecture Notes on Computer Science 5020, "Ada 2005 Rationale: The Language, the Standard Libraries," John Barnes, Springer, 2008.
Ada 95 Quality and Style Guide, SPC-91061-CMC, version 02.01.01. Herndon, Virginia: Software Productivity Consortium, 1992.
Ada Language Reference Manual, The consolidated Ada Reference Manual, consisting of the international standard (ISO/IEC 8652:1995): *Information Technology -- Programming Languages -- Ada*, as updated by changes from *Technical Corrigendum 1* (ISO/IEC 8652:1995:TC1:2000), and Amendment 1 (ISO/IEC 8526:AMD1:2007).

IEEE 754-2008, IEEE Standard for Binary Floating Point Arithmetic, IEEE, 2008.
IEEE 854-1987, IEEE Standard for Radix-Independent Floating-Point Arithmetic, IEEE, 1987

Ada.2 General terminology and concepts

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

Access object: An object of an access type.

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

Access type: The type for objects that designate (point to) other objects.

Access value: The value of an access type; a value that is either null or designates (points at) another object.

Allocator: The Ada term for the construct that allocates storage from the heap or from a storage pool.

Atomic and Volatile: Ada can force every access to an object to be an indivisible access to the entity in memory instead of possibly partial, repeated manipulation of a local or register copy. In Ada, these properties are specified by **pragma**s.

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

Attributes: Predefined characteristics of types and objects; attributes may be queried using syntax of the form <entity>'<attribute_name>.

Bit Ordering: Ada allows use of the attribute Bit_Order of a type to query or specify its bit ordering representation (High_Order_First and Low_Order_First). The default value is implementation defined and available at System.Bit_Order.

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

Case statement: A case statement provides multiple paths of execution dependent upon the value of the case expression. Only one of alternative sequences of statements will be selected.

Case expression: The case expression of a case statement is a discrete type.

Case choices: The choices of a case statement must be of the same type as the type of the expression in the case statement. All possible values of the case expression must be covered by the case choices.

Compilation unit: The smallest Ada syntactic construct that may be submitted to the compiler. For typical file-based implementations, the content of a single Ada source file is usually a single compilation unit.

Configuration pragma: A directive to the compiler that is used to select partition-wide or system-wide options. The **pragma** applies to all compilation units appearing in the compilation, unless there are none, in which case it applies to all future compilation units compiled into the same environment.

Controlled type: A type descended from the language-defined type Controlled or Limited_Controlled. A controlled type is a specialized type in Ada where an implementer can tightly control the initialization, assignment, and finalization of objects of the type. This supports techniques such as reference counting, hidden levels of indirection, reliable resource allocation, etc.

Dead store: An assignment to a variable that is not used in subsequent instructions. A variable that is declared but neither read nor written to in the program is an unused variable.

Default expression: an expression of the formal object type that may be 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. It can control, for example, the bounds of a component of the type if the component is an array. A discriminant for a task type can be used to pass data to a task of the type upon creation.

Endianness: the programmer may specify the endianness of the representation through the use of a **pragma**.

Enumeration Representation Clause: An enumeration representation clause may be used to specify the internal codes for enumeration literals.

Enumeration Type: An enumeration type is a discrete type defined by an enumeration of its values, which may be named by identifiers or character literals. In Ada, the types Character and Boolean are enumeration types. 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.

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: Represents a kind of exceptional situation. There are set of predefined exceptions in Ada in **package** Standard: Constraint_Error, Program_Error, Storage_Error, and Tasking_Error; one of them is raised when a language-defined check fails.

Expanded name: A variable V inside subprogram S in package P can be named V, or P.S.V. The name V is called the *direct name* while the name P.S.V is called the *expanded name*.

Explicit Conversion: The Ada term explicit conversion is equivalent to the term cast in Section 6.IHN.3.

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: A declaration can be *hidden*, either from direct visibility, or from all visibility, within certain parts of its scope. Where *hidden from all visibility*, it 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 a selector_name is still possible.

Homograph: Two declarations are *homographs* if they have the same name, and do not overload each other according to the rules of the language.

Identifier: Identifier is the Ada term that corresponds to the term name.

Idempotent behaviour: The property of an operations that has the same effect whether applied just once or multiple times. An example would be an operation that rounded a number up to the nearest even integer greater than or equal to its starting value.

Implementation defined: Aspects of semantics of the language specify a set of possible effects; the implementation may choose to implement any effect in the set. Implementations are required to document their behaviour in implementation-defined situations.

Implicit Conversion: The Ada term implicit conversion is equivalent to the term coercion.

> 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, e.g., 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.IHN as the result of a breach in a strong type system. Unchecked_Conversion is occasionally needed to interface with type-less data structures, e.g., 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.

Modular type: A modular type is an integer type with values in the **range** 0 .. modulus - 1. The modulus of a modular type can be up to 2**N for N-bit word architectures. A modular type has wrap-around semantics for arithmetic operations, bit-wise "and" and "or" operations, and arithmetic and logical shift operations.

Obsolescent Features: Ada has a number of features that have been declared to be obsolescent; this is equivalent to the term deprecated. These are documented in Annex J of the Ada Reference Manual.

Operational and Representation Attributes: The values of certain implementation-dependent characteristics can be obtained by querying the applicable 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.

Overriding Indicators: If an operation is marked as “overriding”, then the compiler will flag an error if the operation is incorrectly named or the parameters are not as defined in the parent. Likewise, if an operation is marked as “not overriding”, then the compiler will verify that there is no operation being overridden in parent types.

Partition: A partition is a program or part of a program that can be invoked from outside the Ada implementation.

Pointer: Synonym for “access object.”

Pragma: A directive to the compiler.

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 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: Specifies that all reads and updates on a volatile object are performed directly to memory.

Pragma Volatile_Components: Specifies that all reads and updates of an element of an array are performed directly to memory.

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: provide a way to specify the layout of components within records, that is, their order, position, and size.

Scalar Type: A Scalar type comprises enumeration types, integer types, and real types.

Separate Compilation: Ada requires that calls on libraries are checked for illegal situations as if the called routine were declared locally.

Storage Pool: A named location in an Ada program where all of the objects of a single access type will be allocated. 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 and do not suffer from fragmentation.

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 are used to specify the component position and size within the record.

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 task represents a separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks. An Ada program may be comprised of a collection of tasks.

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 may be used to disallow uses of Unchecked_Access.

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.

The following Ada restrictions prevent the application from using any allocators:

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

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

pragma Restrictions(No_Implicit_Heap_Allocations): prevents the use of allocators that would use the heap, but permits allocations from storage pools.

pragma Restrictions(No_Unchecked_Deallocations): prevents allocated storage from being returned 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.

Ada.3Type System [IHN]

Ada.3.1 Applicability to language

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

Explicit conversions can violate the application type semantics. e.g., conversion from feet to meter, or, in general, between types that denote value of different units, without the appropriate conversion factors can cause application vulnerabilities. However, no undefined semantics can result and no values can arise that are outside the range of legal values of the target type.

Failure to apply correct 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 cause systems, threads or components to halt unexpectedly.

Unchecked conversions circumvent the type system and therefore can cause unspecified behaviour (see Section Ada.3.AMV).

Ada.3.2 Guidance to language users

- The predefined 'Valid attribute for a given subtype may be applied to any value to ascertain if the value is a legal value of the subtype. This is especially useful when interfacing with type-less systems or after Unchecked_Conversion.
- A conceivable measure to prevent incorrect unit conversions is to restrict 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. These bodies are to be critically reviewed for proper conversion factors.
- Exceptions raised by type and subtype conversions shall be handled.

Ada.4 Bit Representation [STR]

Ada.4.1 Applicability to language

In general, the type system of Ada protects against the vulnerabilities outlined in Section 6.STR. However, the use of Unchecked_Conversion, calling foreign language routines, and unsafe manipulation of address representations voids these guarantees.

The vulnerabilities caused by the inherent conceptual complexity of bit level programming are as described in Section 6.STR.

Ada.4.2 Guidance to language users

The vulnerabilities associated with the complexity of bit-level programming can be mitigated by:

- The use of 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.
- The use of pragma Atomic and **pragma** Atomic_Components to ensure that all updates to objects and components happen atomically.
- The use of pragma Volatile and **pragma** Volatile_Components to notify the compiler that objects and components must be read immediately before use as other devices or systems may be updating them between accesses of the program.
- The default object layout chosen by the compiler may be queried by the programmer to determine the expected behaviour of the final representation.

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.

Ada.5 Floating-point Arithmetic [PLF]

Ada.5.1 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.PLF.2.

Ada.5.2 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 (e.g., 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 (e.g., Generic_Elementary_Functions) for common mathematical operations (trigonometric operations, logarithms, etc.).
- Use an Ada implementation that supports Annex G (Numerics) of the Ada standard, 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 (e.g., 'Exponent).
- In cases where absolute precision is needed, consider replacement of floating-point types and operations with fixed-point types and operations.

Ada.6 Enumerator Issues [CCB]

Ada.6.1 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.CCB 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. In the latter case, the vulnerability described in 6.CCB does not exist.

The full range of possible values of the expression in a **case** statement must be covered by the case choices. Two distinct choices of a case statement can not cover the same value. Choices can be expressed by single values or subranges of values. 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 case choices. These restrictions are enforced at compile time. Identical rules apply to aggregates of arrays.

The remaining vulnerability is that unexpected values are captured by the **others** clause or a subrange as case choice after an additional enumeration literal has been added to the enumeration type definition. 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 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.

Ada.6.2 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.

Ada.7 Numeric Conversion Errors [FLC]

Ada.7.1 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, 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.

The occurrence of an exception on a conversion can disrupt a computation, which could potentially cause a failure mode or denial-of-service problems.

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.

Loss of precision can occur on explicit conversions from a floating-point type to an integer type, but in that case the loss of precision is being explicitly requested. Truncation cannot occur, and will lead to Constraint_Error if attempted.

There exist operations in Ada for performing shifts and rotations on values of unsigned types, but such operations are also explicit (function calls), so must be applied deliberately by the programmer, and can still only result in values that fit within the range of the result type of the operation.

Ada.7.2 Guidance to language users

- Use Ada's capabilities for user-defined scalar types and subtypes to avoid accidental mixing of logically incompatible value sets.
- Use 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.

Ada.8 String Termination [CJM]

With the exception of unsafe programming, 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.

Ada.9 Buffer Boundary Violation (Buffer Overflow) [HCB]

With the exception of unsafe programming, this vulnerability is not applicable to Ada as this vulnerability can only happen as a consequence of unchecked array indexing or unchecked array copying, which do not occur in Ada (see Ada.3.XYZ and Ada.3.XYW).

Ada.10 Unchecked Array Indexing [XYZ]

Ada.10.1 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.

Programmers can write explicit bounds tests to prevent an exception when indexing out of bounds, but failure to do so does not result in accessing storage outside of arrays.

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.

Ada.10.2 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.

Ada.11 Unchecked Array Copying [XYW]

With the exception of unsafe programming, 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.

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

Ada.12.1 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 Section 6.HFC exist in Ada only if unchecked type conversions or unsafe taking of addresses are applied (see Section Ada.2). Other permitted type conversions can never misrepresent the type of the designated entity.

Checked type conversions that affect the application semantics adversely are possible.

Ada.12.2 Guidance to language users

- This vulnerability can be avoided in Ada by not using the features explicitly identified as unsafe.
- Use 'Access which is always type safe.

Ada.13 Pointer Arithmetic [RVG]

With the exception of unsafe programming, this vulnerability is not applicable to Ada as Ada does not allow pointer arithmetic.

Ada.14 Null Pointer Dereference [XYH]

In Ada, this vulnerability does not exist, since compile-time or run-time checks ensure that no null value can be dereferenced.

Ada provides an optional qualification on access types that specifies and enforces that objects of such types cannot have a null value. Non-nullness is enforced by rules that statically prohibit the assignment of either **null** or values from sources not guaranteed to be non-null.

Ada.15 Dangling Reference to Heap [XYK]

Ada.15.1 Applicability to language

Use of Unchecked_Deallocation can cause dangling references to the heap. The vulnerabilities described in 6.XYK 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 ends.

For global access types, allocated objects can only be deallocated through an instantiation of the generic procedure Unchecked_Deallocation.

Ada.15.2 Guidance to language users

- Use local access types where possible.
- Do not use Unchecked_Deallocation.
- Use Controlled types and reference counting.

Ada.16 Wrap-around Error [XYY]

With the exception of unsafe programming, 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.

Ada raises the predefined exception Constraint_Error whenever an attempt is made to increment an integer above its maximum positive value or to decrement an integer below its maximum negative value. Operations to shift and rotate numeric values apply only to modular integer types, and always produce values that belong to the type. In Ada there is no confusion between logical and arithmetic shifts.

Ada.17 Sign Extension Error [XZI]

With the exception of unsafe programming, this vulnerability is not applicable to Ada as Ada does not, explicitly or implicitly, allow unsigned extension operations to apply to signed entities or vice-versa.

Ada.18 Choice of Clear Names [NAI]

Ada.18.1 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 Ada.3.YOW.

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. Thus no confusion can 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.

Ada.18.2 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. See the Ada Quality and Style Guide.

Ada.19 Dead store [WXQ]

Ada.20 Unused Variable [YZS]

Ada.20.1 Applicability to language

Variables might be unused for various reasons:

- Declared for future use. The programmer might have declared the variable knowing that it will be used when the program is complete or extended. Thus, in a farming application, a variable Pig might be declared for later use if the farm decides to expand out of dairy farming.
- The declaration is wrong. The programmer might have mistyped the identifier of the variable in its declaration, thus Peg instead of Pig.
- The intended use is wrong. The programmer might have mistyped the identifier of the variable in its use, thus Pug instead of Pig.

An unused variable declared for later use does not of itself introduce any vulnerability. The compiler will warn of its absence of use if such warnings are switched on.

If the declaration is wrong, then the program will not compile assuming that the uses are correct. Again there is no vulnerability.

If the use is wrong, then there is a vulnerability if a variable of the same type with the same name is also declared. Thus, if the program correctly declares Pig and Pug (of the same type) but inadvertently uses Pug instead of Pig, then the program will be incorrect but will compile.

Ada.20.2 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.
- Unused variables can be easily detected by the compiler, whereas dead stores can be detected by static analysis tools.

Ada.21 Identifier Name Reuse [YOW]

Ada.21.1 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 of 6.YOW. 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.

The failure associated with common substrings of identifiers cannot happen in Ada because all characters in a name are significant (see section Ada.3.NAI).

Name collisions with keywords cannot happen in Ada because keywords are reserved. Library names Ada, System, Interfaces, and Standard can be hidden by the creation of subpackages. For all except package Standard, the expanded name Standard.Ada, Standard.System and Standard.Interfaces provide the necessary qualification to disambiguate the names.

Ada.21.2 Guidance to language users

Use *expanded names* whenever confusion may arise.

Ada.22 Namespace Issues [BJL]

With the exception of unsafe programming, this vulnerability is not applicable to Ada as Ada provides packages to control namespaces and enforces block structure semantics.

Ada.23 Initialization of Variables [LAV]

Ada.23.1 Applicability to language

In Ada, referencing an uninitialized scalar (numeric or enumeration-type) variable is considered a bounded error, with the possible outcomes being the raising of a Program_Error or Constraint_Error exception, or continuing execution with some value of the variable's type, or some other implementation-defined behaviour. The implementation is required to prevent an uninitialized variable used as an array index resulting in updating memory outside the array. Similarly, using an uninitialized variable in a **case** statement cannot result in a jump to something other than one of the case alternatives. Typically Ada implementations keep track of which variables might be uninitialized, and presume they contain any value possible for the given size of the variable, rather than presuming they are within whatever value range that might be associated with their declared type or subtype. The vulnerability associated with use of an uninitialized scalar variable is therefore that some result will be calculated incorrectly or an exception will be raised unexpectedly, rather than a completely undefined behaviour.

Pointer variables are initialized to null by default, and every dereference of a pointer is checked for a **null** value. Therefore the only vulnerability associated with pointers is that a Constraint_Error might be raised if a pointer is dereferenced that was not correctly initialized.

In general in Ada it is possible to suppress run-time checking, using **pragma** Suppress. In the presence of such a pragma, if a condition arises that would have resulted in a check failing and an exception being raised, then the behaviour is completely undefined ("erroneous" in Ada terms), and could include updating random memory or execution of unintended machine instructions.

Ada provides a generic function for unchecked conversion between (sub)types. If an uninitialized variable is passed to an instance of this generic function and the value is not within the declared range of the target subtype, then the subsequent execution is erroneous.

Failure can occur when a scalar variable (including a scalar component of a composite variable) is not initialized at its point of declaration, and there is a reference to the value of the variable on a path that never assigned to the variable. The effects are bounded as described above, with the possible effect being an incorrect result or an unexpected exception.

Ada.23.2 Guidance to language users

Scalar variables are not initialized by default in Ada. Pointer types are default-initialized to null. Default initialization for record types may be specified by the user. 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.

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

- Whenever possible, a variable should be replaced by an initialized constant, if in fact there is only one assignment to the variable, and the assignment can be performed at the point of initialization. Moving the object declaration closer to its point of use by creating a local declare block can increase the frequency at which such a replacement is possible. Note that initializing a variable with an inappropriate default value such as zero can result in hiding underlying problems, because static analysis tools or the compiler itself will then be unable to identify use before correct initialization.
- If the compiler has a mode that detects use before initialization, then this mode should be enabled and any such warnings should be treated as errors.
- 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.

Ada.24 Operator Precedence/Order of Evaluation [JCW]

Ada.24.1 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, i.e. (A = B) or (C = D).
- Mixed logical operators are not allowed without parentheses, i.e., "A or B or C" is legal, as well as "A and B and C", but "A and B or C" is not (must write "(A and B) or C" or "A and (B or C)".
- Assignment is not an operator in Ada.

Ada.24.2 Guidance to language users

The general mitigation measures can be applied to Ada like any other language.

Ada.25 Side-effects and Order of Evaluation [SAM]

Ada.25.1 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. Although functions only have "**in**" parameters, meaning that they are not allowed to modify the value of their parameters, they may modify the value of global variables. Operators in Ada are functions, 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 association of the operators with operands of the same precedence level (in the absence of parentheses imposing a specific association). 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, this implementation dependency can cause unpredictability of the side effects.

Ada.25.2 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.
- 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.

Ada.26 Likely Incorrect Expression [KOA]

Ada.26.1 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 6.KOA 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 a 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 (i.e., "**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
```

Ada.26.2 Guidance to language users

- Compilers and other static analysis tools can detect some cases (such as the preceding example) where short-circuited evaluation could prevent the failure of a run-time check.
- Developers may also choose to use short-circuit forms by default (errors resulting from the incorrect use of short-circuit forms are much less common), but this makes it more difficult for the author 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.

Ada.27 Dead and Deactivated Code [XYQ]

Ada.27.1 Applicability to language

Ada allows the usual sources of dead code (described in 6.XYQ.3) that are common to most conventional programming languages.

Ada.27.2 Guidance to language users

Implementation specific mechanisms may be provided to support the elimination of dead code. In some cases, **pragma**s such as Restrictions, Suppress, or Discard_Names may be used 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.

Ada.28 Switch Statements and Static Analysis [CLL]

With the exception of unsafe programming, this vulnerability is not applicable to Ada as

Ada requires that a case statement provide exactly one alternative for each value of the expression's subtype. 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.

Ada.29 Demarcation of Control Flow [EOJ]

With the exception of unsafe programming, 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.

Ada.30 Loop Control Variables [TEX]

With the exception of unsafe programming, 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.

Ada.31 Off-by-one Error [XZH]

Ada.31.1 Applicability to language

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

A **for loop** in Ada does not involve the programmer having 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.

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.

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.

Ada.31.2 Guidance to language users

- Whenever possible, a **for loop** should be used instead of a **while loop**.
- Whenever possible, the 'First, 'Last, and 'Range attributes should be used for loop termination. If the 'Length attribute must be used, then extra care should be taken to ensure that the length expression considers the starting index value for the array.

Ada.32 Structured Programming [EWD]

Ada.32.1 Applicability to language

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

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

Ada.32.2 Guidance to language users

Avoid the use of **goto**, **loop exit** statements, **return** statements in **procedure**s and more than one **return** statement in a **function**.

Ada.33 Passing Parameters and Return Values [CSJ]

Ada.33.1 Applicability to language

Ada employs the mechanisms (e.g., modes **in**, **out** and **in out**) that are recommended in Section 6.CSJ. These mode definitions are not optional, mode **in** being the default. The remaining vulnerability is aliasing when a large object is passed by reference.

Ada.33.2 Guidance to language users

- Follow avoidance advice in Section 6.CSJ.

Ada.34 Dangling References to Stack Frames [DCM]

Ada.34.1 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 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 provide pointers to static objects, 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.

Ada.34.2 Guidance to language users

- Only use 'Address attribute on static objects (e.g., a register address).
- Do not use 'Address to provide indirect untyped access to an object.
- Do not use conversion 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.

Ada.35 Subprogram Signature Mismatch [OTR]

Ada.35.1 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.

In Ada, at compilation time, the parameter association is checked to ensure that the type of each actual parameter is 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.

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.

Ada.35.2 Guidance to language users

- Do not use default expressions for formal parameters.
- Interfaces between Ada program units and program units in other languages can be managed using **pragma** Import to specify subprograms that are defined externally and **pragma** Export to specify subprograms that are used externally. These **pragma**s specify the imported and exported aspects of the subprograms, this includes the calling convention. Like subprogram calls, all parameters need to be specified when using **pragma** Import and **pragma** Export.
- The **pragma** Convention may be used 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.
- In addition, the Valid attribute may be used to check if an object that is part of an interface with another language has a valid value and type.

Ada.36 Recursion [GDL]

Ada.36.1 Applicability to language

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

Ada.36.2 Guidance to language users

- If recursion is used, then a Storage_Error exception handler may be used to handle insufficient storage due to recurring execution.
- Alternatively, the asynchronous control construct may be used to time the execution of a recurring call and to terminate the call if the time limit is exceeded.

- In Ada, the **pragma** Restrictions may be invoked with the parameter No_Recursion. In this case, the compiler will ensure that as part of the execution of a subprogram the same subprogram is not invoked.

Ada.37 Returning Error Status [NZN]

Ada.37.1 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.

As described in Section 6.NZN, there is some complexity in understanding the exception handling methodology especially with respect to object-oriented programming and multi-threaded execution.

Ada.37.2 Guidance to language users

- In addition to the mitigations defined in the main text, values delivered to an Ada program from an external device may be checked for validity prior to being used. This is achieved by testing the Valid attribute.

Ada.38 Termination Strategy [REU]

Ada.38.1 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.

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.

Ada.38.2 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.
- Use objects of controlled types to ensure that resources are properly released if a task terminates unexpectedly.
- The **abort** statement should be used sparingly, if at all.
- For high-integrity systems, exception handling is usually forbidden. However, a top-level exception handler can be used to restore the overall system to a coherent state.
- 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.
- Annex C of the Ada Reference Manual (Systems Programming) defines the package Ada.Task_Termination to be used to monitor task termination and its causes.
- Annex H of the Ada Reference Manual (High Integrity Systems) describes several **pragma**, restrictions, and other language features to be used when writing systems for high-reliability applications. For example, the **pragma** Detect_Blocking forces an implementation to detect a potentially blocking operation within a protected operation, and to raise an exception in that case.

Ada.39 Type-breaking Reinterpretation of Data [AMV]

Ada.39.1 Applicability to language

Unchecked_Conversion can be used to bypass the type-checking rules, and its use is thus unsafe, as 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.

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.

Ada.39.2 Guidance to language users

- 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 sign extension.
- Unchecked_Union should only be used 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.
- Using address clauses to obtain overlays should be avoided. If the types of the objects are the same, then a renaming declaration is preferable. Otherwise, the **pragma** Import should be used to inhibit the initialization of one of the entities so that it does not interfere with the initialization of the other one.

Ada.40 Memory Leak [XYL]

Ada.40.1 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 can be present but is restricted to the single pool and which makes it easier to detect by verification. For objects that are 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 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.

Ada.40.2 Guidance to language users

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

Ada.41 Templates and Generics [SYM]

With the exception of unsafe programming, 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.

Ada.42 Inheritance [RIP]

Ada.42.1 Applicability to language

The vulnerability documented in Section 6.RIP applies to Ada.

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

Ada.42.2 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 the main body of the document.

Ada 43 Extra Intrinsics [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 a qualified name) which subprogram is meant.

Ada.44 Argument Passing to Library Functions [TRJ]

Ada.44.1 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 runtime exceptions.

Ada.44.2 Guidance to language users

- Exploit the type and subtype system of Ada to express preconditions (and postconditions) on the values of parameters.
- Document all other preconditions and ensure by guidelines that either callers or callees are responsible for checking the preconditions (and postconditions). Wrapper subprograms for that purpose are particularly advisable.
- Specify the response to invalid values.

Ada.45 Dynamically-linked Code and Self-modifying Code [NYY]

With the exception of unsafe programming, 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.

Ada.46 Library Signature [NSQ]

Ada.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.NSQ 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.

Ada.46.2 Guidance to language users

- The mitigation mechanisms of Section 6.NSQ.5 are applicable.

Ada.47 Unanticipated Exceptions from Library Routines [HJW]

Ada.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, or the task or program will terminate.

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 an 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 Section 6.HJW apply.

Ada.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** =>.
- Document any exceptions that may be raised by any Ada units being used as library routines.

Ada.48 Pre-Processor Directives [NMP]

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

Ada.49 Obscure Language Features [BRS]

Ada.49.1 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 may possess.

Ada.49.2 Guidance to language users

The **pragma** Restrictions can be used to prevent the use of certain features of the language. Thus, if a program should not use feature X, then writing **pragma** Restrictions (No_X); ensures that any attempt to use feature X prevents the program from compiling.

Similarly, features in a Specialized Needs Annex should not be used unless the application area concerned is well-understood by the programmer.

Ada.50 Unspecified Behaviour [BQF]

Ada.50.1 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).

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 Standard for *unspecified* provides the full list. Similarly, the index entry for *bounded error* provides the full list of references to places in the Ada Standard 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.

Ada.50.2 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 situations where order of evaluation or number of evaluations is unspecified, using only operations with no side-effects, or idempotent behaviour, will avoid the vulnerability;
- For situations involving generic formal subprograms, care should be taken that the actual subprogram satisfies all of the stated expectations;
- For situations involving unspecified values, care should be taken not to depend on equality between potentially distinct values;
- For situations involving bounded errors, care should be taken to avoid the situation 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 the Ada Annex section Ada.3.28 on Initialization of Variables [LAV] for a discussion of uninitialized variables in Ada, a common cause of a bounded error.

Ada.51 Undefined Behaviour [EWF]

Ada.51.1 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 Standard 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, etc.

Ada.51.2 Guidance to language users

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

- All data shared between tasks should be within a protected object or marked Atomic, whenever practical;
- Any use of Unchecked_Deallocation should be carefully checked to be sure that there are no remaining references to the object;
- **pragma** Suppress should be used 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 Section 6.EWF.5 are applicable here.

Ada.52 Implementation-Defined Behaviour [FAB]

Ada.52.1 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 Standard, 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.

Ada.52.2 Guidance to language users

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, etc. 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.

- Programmers should be aware of the contents of Annex M of the Ada Standard and avoid implementation-defined behaviour whenever possible.
- Programmers should make use of the constants and subtype attributes provided in package System and elsewhere to avoid exceeding implementation-defined limits.

- Programmers should minimize use of any predefined numeric types, as the ranges and precisions of these are all implementation defined. Instead, they should declare their own numeric types to match their particular application needs.
- When there are implementation-defined formats for strings, such as Exception_ Information, any necessary processing should be localized in packages with implementation-specific variants.

Ada.53 Deprecated Language Features [MEM]

Ada.53.1 Applicability to language

If obsolescent language features are used, then the mechanism of failure for the vulnerability is as described in Section 6.MEM.3.

Ada.53.2 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 reference manual to determine if a feature is obsolescent.

Ada.54 Implications for standardization

Future standardization efforts should consider:

- Some languages (e.g., Java) require that all local variables either be initialized at the point of declaration or on all paths to a reference. Such a rule could be considered for Ada.
- **Pragma** Restrictions could be extended to allow the use of these features to be statically checked.
- 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.
- When appropriate, language-defined checks should be added to reduce the possibility of erroneous execution, such as by disallowing unsynchronized access to shared variables.
- 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, etc., in **package** Standard).
- Ada could define a **pragma** Restrictions identifier No_Hiding that forbids the use of a declaration that results in a local homograph.
- Add the ability to declare in the specification of a function that it is pure, i.e., it has no side effects.
- **Pragma** Restrictions could be extended to restrict the use of 'Address attribute to library level static objects.
- Future Standardization of Ada should consider implementing a language-provided reference counting storage management mechanism for dynamic objects.
- Provide mechanisms to prevent further extensions of a type hierarchy.
- Future standardization of Ada should consider support for arbitrary pre- and postconditions.
- 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 pre-conditions, post-conditions and invariants, would be helpful.

- 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 pre-conditions, post-conditions and invariants, would be helpful.