ISO/IEC JTC 1/SC 22/OWGV N 0199

Draft language-specific annex for Ada resulting from workshop at 2009 Ada Europe conference

Date	22 June 2009
Contributed by	John Benito
Original file name	Annex Ada.doc
Notes	

Annex Outline

Annex Ada

(informative)

Language Specific Vulnerability Template

Ada. Ada specific information for vulnerabilities

Every vulnerability description of Clause 6 of the main document should be addressed in the annex in the same order even if there is simply a notation that it is not relevant to the language in question.

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.

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.2 General terminology and concepts

Use the LRM language here for implementation defined, unspecified, erroneous execution, bounded error, pragma.

Erroneous Execution:

<u>Expanded name</u>: 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*.

Ada.3.7 Choice of Clear Names [NAI]

Ada.3.7.0 Status and history

20090610 – J. Barnes & A. Burns at Workshop

Ada.3.7.1 Language-specific terminology and features

The term name used in this Technical Report corresponds to the term identifier in Ada. In this section, the term name will be used in this context.

Ada is not a case-sensitive language. Names may use an underscore to improve clarity.

Ada.3.7.2 Description of programming language vulnerability or application vulnerability

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 should be covered elsewhere in this Technical Report.

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

- <u>Mixed casing</u>. 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.

- <u>Underscores and periods</u>. 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.

- <u>Singular/plural forms</u>. 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.

- <u>International character sets</u>. Ada compilers strictly conform to the appropriate international standard for character sets.

- <u>Identifier length</u>. 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.

Ada.3.7.3 Mechanism of failure

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.3.7.4 Avoiding the vulnerability or mitigating its effects in Ada

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.3.7.5 Implications for standardization

This Technical Report should include a vulnerability to address concerns associated with overloading.

Ada.3.7.6 Bibliography

None

Ada.3.10 Identifier Name Reuse [YOW]

Ada.3.10.0 Status and history

20090609: Stephen Michell, JP Rosen – Ada Europe Vulnerabilities Workshop

Ada.3.10.1 Language-specific terminology and Features

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

Overriding: A subprogram *overrides* another if they have identical names and signatures, except that the controlling operand of one is a derived type of the overridden subprogram.

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.

Ada.3.10.2 Description of programming language vulnerability or application vulnerability

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 6.10.

Examples of the problem:

```
package body P is
        I : Integer;   -- static object called I

        procedure Calculate( X : in out Float;
                                   I : in      Integer) is -- parameter called I
        begin
                X := X * float(I*I); -- want to multiply static I * parameter I
```

```
                                          -- static I hidden, wrong product, no diagnostic
            I := I + 1;                    -- want to increment static I, hidden by parameter I
                                          -- get compiler diagnostic since we cannot assign
                                          -- to an in parameter.
      end Calculate;
   …
   end P;
```

Here, the parameter called I hides the static variable I because they have exactly the same type and the parameter *hides"* the outer name. This can be corrected by writing

```
      package body P is
            I : Float;   -- static object called P.I

            procedure Proc is
                  I : Integer -– local variable called P.Proc.I
            begin
                  I := 1; -- removal of local I causes diagnostic since must say I := 1.0 for float
            end Proc;
      end P;
```

Consider an example with more nested scope:

```
      procedure P is
            type Age is range 0..125;
            I : Age;
      begin
       <<INNER>> declare
                  I : Integer := 0;
            begin
                  P.Inner.I := P.Inner.I+1       -- increment local I
                                                -- removal of Inner.I causes diagnostic
                  P.I := P.I+1;                 -- increment outer I
                                                -- suceeds even if inner.I removed
            end Inner;
      end P;
```

In this example, P.I and P.Inner.I are *expanded names*.

Ada.3.10.3 Mechanism of failure

The failure associated with hiding due to nested scopes applies to Ada. 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.3.7).

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.3.10.4 Avoiding the vulnerability or mitigating its effects in Ada

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

- A good way to be guaranteed to keep names separated is to always use the *expanded*

name. This guarantees that, even if the simple name could produce a conflict, there is never any doubt as to usage in the mind of the human reader. Indeed, high integrity system guidelines recommend that distinct and representative names be used of items, and that each usage of a name be distinct.

Ada.3.10.5 Implications for standardization

- Ada could define a **pragma** Restrictions identifier No_Hiding that forbids the use of a declaration that result in a local homograph.

Ada.3.10.6 Bibliography

None

Ada.3.11 Type System [IHN]

Ada.3.11.0 Status and history

08-06-09   Created, EP at Vulnerabilities Workshop

Ada.3.11.1 Language-specific terminology and features

Ada does not use the C-specific terms "coercion" and "cast". It uses the terms "implicit conversion" and "explicit conversion" to refer to the same language concepts.
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, the program is rejected by the compiler. If a dynamic check fails, 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) Unchecked conversions: Conversions that are obtained by instantiating the generic subprogram Unchecked-Conversion are unsafe and enable all vulnerabilities mentioned in the TR 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.
   c) 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.

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.

Ada.3.11.2 Description of programming language vulnerability or application vulnerability

Unchecked conversions can cause all conceivable problems that result from circumventing a type system.

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.

Ada.3.11.3 Mechanism of failure

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

Applying instances of Unchecked_Conversion additionally risk undefined semantics, since the generated value may be a value that is not legal for the target type.

Failure to handle the exceptions raised by failed checks of dynamic subtype properties cause systems, threads or components to halt unexpectedly.

Ada.3.11.4 Avoiding the vulnerability or mitigating its effects in Ada

The vulnerabilities of the type system can be avoided or mitigated in Ada in the following ways:

- Instances of Unchecked_Conversions can be prohibited by using the Restriction **pragma** for No_Unchecked_Conversion. The restriction is then enforced by the compiler.
- Additionally, 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.

No other vulnerabilities exist.

Ada.3.11.5 Implications for standardization

None

Ada.3.11.6 Bibliography

None

Ada.3.12 Bit Representation [STR]

Ada.3.12.0 Status and history

20090610: T. Vardanega & M. Pinho at Vulnerabilities Workshop

Ada.3.12.1 Language-specific terminology and features

<u>Operational and Representation Attributes</u>: The values of certain implementation-dependent characteristics can be obtained by querying the applicable attributes. Some attributes are 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.

<u>Record Representation Clauses</u>: provide a way to specify the layout of components within records, that is, their order, position, and size.

<u>Storage Place Attributes</u>: 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.

<u>Bit Ordering</u>: 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.

<u>Modular Types</u>: a class of unsigned types in Ada that ranges 0..n, where n can be up to $2**N$ for N-bit word architectures. These types have wrap-around semantics for arithmetic operations, bit-wise "and" and "or" operations, and arithmetic and logical shift operations.

<u>Atomic</u> and <u>Volatile</u>: 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.

<u>Endianness</u>: the programmer may specify the endianness of the representation through the use of a **pragma**.

Ada.3.12.2 Description of programming language vulnerability or application vulnerability

In general, the type system of Ada protects against the vulnerabilities outlined in Section 6.12 of this Technical Report. 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 remain.

Ada.3.12.3 Mechanism of failure

See this Technical Report for failures due to the conceptual complexity of bit-level programming.

Ada.3.12.4 Avoiding the vulnerability or mitigating its effects in Ada

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 language processor 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 behavior 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.3.12.5 Implications for standardization

None

Ada.3.12.6 Bibliography

None

Ada.3.14 Enumerator Issues [CCB]

Section 6.14 of this Technical Report provides an explanation of the collection of issues associated with enumerations. This section of Appendix Ada specifies the characteristics of Ada that are symptomatic of these vulnerabilities and the mitigating features of the Ada programming language and associated toolset.

Ada.3.14.0 Status and history

20090609 – J. Tokar Vulnerabilities Workshop Input

Ada.3.14.1 Language-specific terminology and features

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.

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

Case Statement: A case statement selects for execution one of a number of alternative sequences of statements; the chosen alternative is defined by the value of an expression. The expression of a case statement is expected to be of any discrete type. The choices of a case statement must be of the same type as the type of the expression in the case statement.

Ada.3.14.2 Description of programming language vulnerability or application vulnerability

The full range of possible values of the expression in a **case** statement must be addressed by the case choices. Two distinct choices of a case statement can not cover the same value. 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.

The remaining vulnerability is that unexpected values are captured by the **others** clause. 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.

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 causing "holes" in the array object. Normally, these holes are not accessible in Ada due to the strict type checking on the array index type and constraint checking on the array index value. However, the holes may be accessed as the result of the use of unchecked conversions.

Ada.3.14.3 Mechanism of failure

Use of the **others** clause for a case statement may result in unintended values being captured by this clause.

Use of Unchecked_Conversion on enumeration types as array indices may result in access to undefined regions of the array and lead to erroneous execution.

Ada.3.14.4 Avoiding the vulnerability or mitigating its effects in Ada

This vulnerability can be avoided or mitigated in Ada in the following ways:
*   For case statements, do not use the **others** choice.
*   Do not use Unchecked_Conversion on enumeration types as array indices.

Ada.3.14.5 Implications for standardization

None

Ada.3.14.6 Bibliography

None

Ada.3.27 Initialization of Variables [LAV]

Ada.3.27.0 Status and history

20090609 Tucker Taft, Rod Chapman – Workshop in Brest

Ada.3.27.1 Language-specific terminology and features

Pointer: objects of an access type.
Controlled types
Pragma Suppress
Pragma Normalize

Ada.3.27.2 Description of programming language vulnerability or application vulnerability

In Ada, uninitialized variables can only lead to value failure. Pointers are initialized to null by default, therefore this is not an issue for Ada.

In Ada, access to arrays with indices that are variables that have not been initialized will not update random memory. Similarly, case statements *– more words are needed here to explain why wild jumps will not happen in Ada.*

Pragma Suppress implications.

Ada.3.27.3 Mechanism of failure

Failure can occur when a 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.

Ada.3.27.4 Avoiding the vulnerability or mitigating its effects in Ada

Scalars are not initialized by default in Ada. Default initialization for record types and controlled types may be specified by the user.

Determine what else is needed here.

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 of when replacing with a constant 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.

- The **pragma** Normalize_Scalars can be used to ensure that scalars 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.3.27.5 Implications for standardization

None

Ada.3.27.6 Bibliography

None

Ada.SPARK SPARK Ada specific information for vulnerabilities

Ada.SPARK.1 Identification of standards and associated documentation

Ada.SPARK.2 General terminology and concepts

Ada.SPARK.3

Ada.SPARK.3.14 Enumerator Issues [CCB]

For the SPARK Ada programming languages all of the discussion in section Ada.3.14 is applicable with the constraints described in this section.

Ada.SPARK.3.14.1 Language-specific terminology and features

As in Ada.3.14.1.

Ada.SPARK.3.14.2 Description of programming language vulnerability or application vulnerability

As in Ada.3.14.2.

Ada.SPARK.3.14.3 Mechanism of failure

<TBD>

Ada.SPARK.3.14.4 Avoiding the vulnerability or mitigating its effects in SPARK

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

…

Ada.SPARK.3.27 Initialization of Variables [LAV]

All uninitialized variables are always detectable prior to compilation using sound static information-flow analysis [1].

Ada.SPARK.3.27.1 Language-specific terminology and features

Description of SPARK relevant features and terminology

Ada.SPARK.3.27.2 Description of programming language vulnerability or application vulnerability

Susceptibility and examples.

None. All uninitialized variables are always detected.

Ada.SPARK.3.27.3 Mechanism of failure

Not applicable.

Ada.SPARK.3.27.4 Avoiding the vulnerability or mitigating its effects in SPARK

This vulnerability is eliminated by use of static information flow analysis.

Ada.SPARK.3.27.5 Implications for standardization

None.

Ada.SPARK.3.27.6 Bibliography

[1] Information-Flow and Data-Flow Analysis of while-Programs. Bernard Carré and Jean-Francois Bergeretti, ACM Transactions on Programming Languages and Systems (TOPLAS) Vol. 7 No. 1, January 1985. pp 37-61.

OVERLOADING BASKET
The enumeration literals that are associated with enumeration types that are declared within packages and accessed by other Ada program units through the **with** clause may be accessed without qualified notation with the inclusion of the **use** clause. As mentioned previously in the *<xxx>* section of this annex, the **use** clause introduces a number of vulnerabilities that may result in an Ada program operating differently dependent upon the presence or absence of the **use** clause.

The **use type** feature of Ada 2005 could be extended to include enumeration literals thereby avoiding the need to use fully extended names or the **use** clause to access enumeration literals.