Stephen Michell 2015-6-5 11:16 PM
Deleted: 2

ISO/IEC JTC 1/SC 22 N 0554

Stephen Michell 2017-2-20 9:16 AM
Deleted: 0000

Date: 2015-06-06

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:14 pt

ISO/IEC TR 24772-3

Stephen Michell 2015-6-5 11:16 PM
Deleted: 3-05

Edition 1

Stephen Michell 2015-6-5 11:16 PM
Deleted: 2

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Vulnerability descriptions for the programming language C

Stephen Michell 2015-6-5 11:16 PM
Deleted: Ada

Élément introductif — Élément principal — Partie n: Titre de la partie

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: International standard
Document subtype: if applicable
Document stage: (10) development stage
Document language: E

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Stephen Michell 2015-6-5 11:17 PM
Deleted: ,

Introduction

This Technical Report provides guidance for the programming language C so that application developers considering C or using C will be better able to avoid the programming constructs that lead to vulnerabilities in software written in the C language and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This technical can also be used in comparison with companion technical reports and with the language-independent report, TR 24772-1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This technical report part is intended to be used with TR 24772-1, which discusses programming language vulnerabilities in a language independent fashion.

It should be noted that this Technical Report is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such report can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

Stephen Michell 2015-6-5 11:17 PM
Deleted: Ada

Stephen Michell 2015-6-5 11:17 PM
Deleted: Ada

Stephen Michell 2015-6-5 11:17 PM
Deleted: Ada

Stephen Michell 2015-6-5 11:17 PM
Deleted: Ada

Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages through language selection and use – Vulnerability descriptions for the programming language C

Stephen Michell 2015-6-5 11:18 PM
Formatted: Font:Bold

Stephen Michell 2015-6-5 11:18 PM
Deleted: Ada

1. Scope

This Technical Report specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.

Vulnerabilities described in this technical report document the way that the vulnerability described in the language-independent writeup (in Tr 24772-1) are manifested in C.

Stephen Michell 2015-6-5 11:18 PM
Deleted: Ada

2. Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 80000–2:2009, *Quantities and units — Part 2: Mathematical signs and symbols to be use in the natural sciences and technology*
ISO/IEC 2382–1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*

ISO/IEC 8652:2012 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.
ISO IEC ???? 754-2008, Binary Floating Point Arithmetic, IEEE, 2008.
ISO IEC ???? 854-1987, Radix-Independent Floating-Point Arithmetic, IEEE, 1987

3. Terms and definitions, symbols and conventions

3.1 Terms and definitions

In addition to the terms and definitions found in TR 24772-1 clause 3, the following terms and definitions apply to this Technical Report.

Stephen Michell 2015-6-5 11:21 PM
Formatted: Normal

access: An execution-time action, to read or modify the value of an object. Where only one of two actions is meant, *read* or *modify*. Modify includes the case where the new value being stored is the same as the previous value. Expressions that are not evaluated do not access objects.

alignment: The requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address.

argument:

actual argument: The expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation.

behaviour: An external appearance or action.

implementation-defined behaviour: The *unspecified behaviour* where each implementation documents how the choice is made. An example of implementation-defined behaviour is the propagation of the high-order bit when a signed integer is shifted right.

locale-specific behaviour: The behaviour that depends on local conventions of nationality, culture, and language that each implementation documents. An example, locale-specific behaviour is whether the `islower()` function returns true for characters other than the 26 lower case Latin letters.

undefined behaviour: The use of a non-portable or erroneous program construct or of erroneous data, for which the C standard imposes no requirements. Undefined behaviour ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). An example of, undefined behaviour is the behaviour on integer overflow.

unspecified behaviour:The use of an unspecified value, or other behaviour where the C Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. For example, unspecified behaviour is the order in which the arguments to a function are evaluated.

bit: The unit of data storage in the execution environment large enough to hold an object that may have one of two values. It need not be possible to express the address of each individual bit of an object.

byte: The addressable unit of data storage large enough to hold any member of the basic character set of the execution environment. It is possible to express the address of each individual byte of an object uniquely. A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit.

character: An abstract member of a set of elements used for the organization, control, or representation of data.

single-byte character: The bit representation that fits in a byte.

multibyte character: The sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set.

wide character: The bit representation that will fit in an object capable of representing any character in the current locale. The C Standard uses the type name `wchar_t` for this object.

correctly rounded result: The representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision.

diagnostic message: The message belonging to an implementation-defined subset of the implementation's message output. The C Standard requires diagnostic messages for all constraint violations.

implementation: A particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment.

implementation limit: The restriction imposed upon programs by the implementation.

memory location: Either an object of scalar[1] type, or a maximal sequence of adjacent bit-fields all having nonzero width. A bit-field- and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields-fi, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two bit-field-fi in the same structure if all members declared between them are also bit-fields, no matter what the sizes of those intervening bit-fields happen to be. For example a structure declared as

```
struct {

char a;
int b:5, c:11, :0, d:8;
struct { int ee:8; } e;

}
```

contains four separate memory locations: The member `a`, and bit-fields `d` and `e.ee` are separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields `b` and `c` together constitute the fourth memory location. The bit-fields `b` and `c` can't be concurrently modified, but `b` and `a`, can be concurrently modified.

object: The region of data storage in the execution environment, the contents of which can represent values. When referenced, an object may be interpreted as having a particular type.

parameter:

formal parameter: The object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition.

recommended practice: A specification that is strongly recommended as being in keeping with the intent of the C Standard, but that may be impractical for some implementations.

runtime-constraint: A requirement on a program when calling a library function.

value: The precise meaning of the contents of an object when interpreted as having a specific type.

implementation-defined value: An unspecified value where each implementation documents how the choice for the value is selected.

indeterminate value: Is either an unspecified value or a trap representation.

[1] Integer types, Floating types and Pointer types are collectively called *scalar* types in the C Standard.

unspecified value:The valid value of the relevant type where the C Standard imposes no requirements on which value is chosen in any instance. An unspecified value cannot be a trap representation.

trap representation: An object representation that need not represent a value of the object type.

4 Language concepts

block-structured language:A language that has a syntax for enclosing structures between bracketed keywords, such as an `if` statement bracketed by `if` and `endif`, as in Fortran, or a code section bracketed by `BEGIN` and `END`, as in PL/1.

***comb-structured language*: A language that has an ordered set of keywords to define separate sections within a block, analogous to the multiple teeth or prongs in a comb separating sections of the comb. For example, in Ada, a block is a 4-pronged comb with keywords `declare`, `begin`, `exception`, `end`, and the `if` statement in Ada is a 4-pronged comb with keywords `if`, `then`, `else`, `end if`.**

5 General guidance for C

[See Template] [Thoughts welcomed as to what could be provided here. Possibly an opportunity for the language community to address issues that do not correlate to the guidance of section 6. For languages that provide non-mandatory tools, how those tools can be used to provide effective mitigation of vulnerabilities described in the following sections]

6 Specific Guidance for C

6.1 General

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

6.2 Type System [IHN]

6.2.1 Applicability to language

C is a statically typed language. In some ways C is both strongly and weakly typed as it requires all variables to be typed, but sometimes allows implicit or automatic conversion between types. For example, C will implicitly convert a `long int` to an `int` and potentially discard many significant digits. Note that integer sizes are implementation defined so that in some implementations, the conversion from a `long int` to an `int` cannot discard any digits since they are the same size. In some implementations, all integer types could be implemented as the same size.

C allows implicit conversions as in the following example:

```
short a = 1023;
```

Stephen Michell 2015-6-5 11:20 PM
Deleted: For the purposes of this document, the terms and definitions given in ISO/IEC 2382–1, in TR 24772-1 and the following apply. Other terms are defined where they appear in *italic* type. ... [1]

Stephen Michell 2015-6-5 11:23 PM
Deleted: How Ada addresses issues in TR24772-1 section 5. ... [2]

Stephen Michell 2015-6-5 11:23 PM
Deleted: Ada

Stephen Michell 2015-6-5 11:23 PM
Deleted: Ada

Stephen Michell 2015-6-5 11:23 PM
Deleted: Ada

Stephen Michell 2015-6-5 11:23 PM
Deleted: Ada

```
int b;
b = a;
```

If an implicit conversion could result in a loss of precision such as in a conversion from a 32 bit `int` to a 16 bit `short int`:

```
int a = 100000;
short b;
b = a;
```

many compilers will issue a warning message.

C has a set of rules to determine how conversion between data types will occur. For instance, every integer type has an integer conversion rank that determines how conversions are performed. The ranking is based on the concept that each integer type contains at least as many bits as the types ranked below it.

The integer conversion rank is used in the usual arithmetic conversions to determine what conversions need to take place to support an operation on mixed integer types.

Other conversion rules exist for other data type-conversions. So even though there are rules in place and the rules are rather straightforward, the variety and complexity of the rules can cause unexpected results and potential vulnerabilities. For example, though there is a prescribed order in which conversions will take place, determining how the conversions will affect the final result can be difficult as in the following example:

```
long foo (short a, int b, int c, long d, long e, long f) {
    return (((b + f) * d - a + e) / c);
}
```

The implicit conversions performed in the `return` statement can be nontrivial to discern, but can greatly impact whether any of the intermediate values wrap around during the computation.

6.2.2 Guidance to language users

- Follow the advice provided in 6.3.5.
- Consideration of the rules for typing and conversions will assist in avoiding vulnerabilities. Make casts explicit to give the programmer a clearer vision and expectations of conversions.

6.3 Bit Representation [STR]

6.3.1 Applicability to language

C supports a variety of sizes for integers such as `short int`, `int`, `long int` and `long long int`. Each may either be signed or unsigned. C also supports a variety of bitwise operators that make bit manipulations easy such as left and right shifts and bitwise operators. These bit manipulations can cause unexpected results or vulnerabilities through miscalculated shifts or platform dependent variations.

Bit manipulations are necessary for some applications and may be one of the reasons that a particular application was written in C. Although many bit manipulations can be rather simple in C, such as masking off the bottom three bits in an integer, more complex manipulations can cause unexpected results. For instance, right shifting a signed integer is implementation defined in C, while shifting by an amount greater than or equal to the size of the data type is undefined behaviour. For instance, on a host where an `int` is of size 32 bits,

Stephen Michell 2015-6-5 11:24 PM
Deleted: Implicit conversions cause no application vulnerability, as long as resulting exceptions are properly handled. ... [3]

Stephen Michell 2015-6-5 11:25 PM
Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm, Tabs: 1.27 cm, Left

Stephen Michell 2015-6-5 11:25 PM
Deleted: ... [4]

Stephen Michell 2015-6-5 11:25 PM
Formatted: List Paragraph, No bullets or numbering

```
unsigned int foo(const int k) {
    unsigned int i = 1;
    return i << k;
}
```

is undefined for values of `k` greater than or equal to 32.

The storage representation for interfacing with external constructs can cause unexpected results. Byte orders may be in little-endian or big-endian format and unknowingly switching between the two can unexpectedly alter values.

6.3.2 Guidance to language users

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

- Only use bitwise operators on unsigned integer values as the results of some bitwise operations on signed integers are implementation defined.
- Use commonly available functions such as `htonl()`, `htons()`, `ntohl()` and `ntohs()` to convert from host byte order to network byte order and vice versa. This would be needed to interface between an i80x86 architecture where the Least Significant Byte is first with the network byte order, as used on the Internet, where the Most Significant Byte is first. ***Note:*** *functions such as these are not part of the C standard and can vary somewhat among different platforms.*
- In cases where there is a possibility that the shift is greater than the size of the variable, perform a check as the following example shows, or a modulo reduction before the shift:

```
unsigned int i;
unsigned int k;
unsigned int shifted_i;
…

if (k < sizeof(unsigned int)*CHAR_BIT)
    shifted_i = i << k;
else
// handle error condition
```

6.4 Floating-point Arithmetic [PLF]

6.4.1 Applicability to language

C permits the floating-point data types float, double and long double. Due to the approximate nature of floating-point representations, the use of float and double data types in situations where equality is needed or where rounding could accumulate over multiple iterations could lead to unexpected results and potential vulnerabilities in some situations.

As with most data types, C is flexible in how `float`, `double` and `long double` can be used. For instance, C allows the use of floating-point types to be used as loop counters and in equality statements. Even though a loop may be expected to only iterate a fixed number of times, depending on the values contained in the floating-point type and on the loop counter and termination condition, the loop could execute forever. For instance iterating a time sequence using 10 nanoseconds as the increment:

Stephen Michell 2015-6-5 11:25 PM
Deleted: In general, the type system of Ada protects against the vulnerabilities outlined in Section 6.4. However, the use of Unchecked_Conversion, calling foreign language routines, and unsafe manipulation of address representations voids these guarantees. ... [5]

Stephen Michell 2015-6-5 11:30 PM
Deleted:

Stephen Michell 2015-6-5 11:30 PM
Formatted: Font:Cambria, Not Bold

Stephen Michell 2015-6-5 11:26 PM
Deleted: ... [6]

```
float f;
for (f=0.0; f!=1.0; f+=0.00000001)
```

may or may not terminate after 10,000,000 iterations. The representations used for `f` and the accumulated effect of many iterations may cause `f` to not be identical to 1.0 causing the loop to continue to iterate forever.

Similarly, the Boolean test

```
float f=1.336f;
float g=2.672f;
if (f == (g/2))
```

may or may not evaluate to true. Given that `f` and `g` are constant values, it is expected that consistent results will be achieved on the same platform. However, it is questionable whether the logic performs as expected when a float that is twice that of another is tested for equality when divided by 2 as above. This can depend on the values selected due to the quirks of floating-point arithmetic.

6.4.2 Guidance to language users

- Do not use a floating-point expression in a Boolean test for equality. In C, implicit casts may make an expression floating-point even though the programmer did not expect it.

Check for an acceptable closeness in value instead of a test for equality when using floats and doubles to avoid rounding and truncation problems.

Do not convert a floating-point number to an integer unless the conversion is a specified algorithmic requirement or is required for a hardware interface.

6.5 Enumerator Issues [CCB]

6.5.1 Applicability to language

The enum type in C comprises a set of named integer constant values as in the example:

```
enum abc {A,B,C,D,E,F,G,H} var_abc;
```

The values of the contents of `abc` would be `A=0, B=1, C=2`, and so on. C allows values to be assigned to the enumerated type as follows:

```
enum abc {A,B,C=6,D,E,F=7,G,H} var_abc;
```

This would result in:

```
A=0, B=1, C=6, D=7, E=8, F=7, G=8, H=9
```

yielding both gaps in the sequence of values and repeated values.

If a poorly constructed `enum` type is used in loops, problems can arise. Consider the enumerated type `abc` defined above used in a loop:

```
int x[8];

for (i=A; i<=H; i++){
```

Stephen Michell 2015-6-5 11:27 PM
Formatted: Normal

Stephen Michell 2015-6-5 11:27 PM
Deleted: ... [7]

Stephen Michell 2015-6-5 11:27 PM
Formatted: Font:(Default) Courier New, English (US)

Stephen Michell 2015-6-5 11:29 PM
Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 1.9 cm + Indent at: 2.54 cm

Stephen Michell 2015-6-5 11:30 PM
Formatted: List Paragraph

Stephen Michell 2015-6-5 11:28 PM
Deleted: ... [8]

Stephen Michell 2015-6-5 11:30 PM
Formatted: English (US)

```
    t = x[i];

}
```

Because the enumerated type `abc` has been renumbered and because some numbers have been skipped, the array will go out of bounds and there is potential for unintentional gaps in the use of `x`.

6.5.2 Guidance to language users

- Follow the guidance of 6.6.5.
- Use enumerated types in the default form starting at 0 and incrementing by 1 for each member if possible. The use of an enumerated type is not a problem if it is well understood what values are assigned to the members.
- Avoid using loops that iterate over an enum that has representation specified for the enums, unless it can be guaranteed that there are no gaps or repetition of representation values within the enum definition.
- Use an enumerated type to select from a limited set of choices to make possible the use of tools to detect omissions of possible values such as in switch statements.
- Use the following format if the need is to start from a value other than 0 and have the rest of the values be sequential:

```
enum abc {A=5,B,C,D,E,F,G,H} var_abc;
```

- Use the following format if gaps are needed or repeated values are desired and so as to be explicit as to the values in the `enum`, then:

```
enum abc {
        A=0,
        B=1,
        C=6,
        D=7,
        E=8,
        F=7,
        G=8,
        H=9
    } var_abc;
```

6.6 Numeric Conversion Errors [FLC]

6.6.1 Applicability to language

C permits implicit conversions. That is, C will automatically perform a conversion without an explicit cast. For instance, C allows

```
int i;
float f=1.25f;
i = f;
```

This implicit conversion will discard the fractional part of `f` and set `i` to 1. If the value of `f` is greater than `INT_MAX`, then the assignment of `f` to `i` would be undefined.

Stephen Michell 2015-6-5 11:31 PM
Deleted: 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: ... [9]

Stephen Michell 2015-6-5 11:32 PM
Formatted: Kern at 16 pt

Stephen Michell 2015-6-5 11:32 PM
Formatted: Normal, Space Before: 0 pt, After: 0 pt, No bullets or numbering

Stephen Michell 2015-6-5 11:32 PM
Deleted: For **case** statements and aggregates, do not use the **others** choice. ... [10]

Stephen Michell 2015-6-5 11:32 PM
Formatted: Kern at 16 pt

The rules for implicit conversions in C are defined in the C standard. For instance, integer types smaller than `int` are promoted when an operation is performed on them. If all values of Boolean, character or integer type can be represented as an `int`, the value of the smaller type is converted to an `int`; otherwise, it is converted to an unsigned `int`.

Integer promotions are applied as part of the usual arithmetic conversions to certain argument expressions; operands of the unary +, –, and ~ operators, and operands of the shift operators. The following code fragment shows the application of integer promotions:

```
char c1, c2;
c1 = c1 + c2;
```

Integer promotions require the promotion of each variable (`c1` and `c2`) to `int` size. The two `int` values are added and the sum is truncated to fit into the `char` type.

Integer promotions are performed to avoid arithmetic errors resulting from the overflow of intermediate values. For example:

```
signed char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3;
```

In this example, the value of `c1` is multiplied by `c2`. The product of these values is then divided by the value of `c3` (according to operator precedence rules). Assuming that signed char is represented as an 8-bit value, the product of `c1` and `c2` (300) cannot be represented. Because of integer promotions, however, `c1`, `c2`, and `c3` are each converted to `int`, and the overall expression is successfully evaluated. The resulting value is truncated and stored in `cresult`. Because the final result (75) is in the range of the signed `char` type, the conversion from `int` back to `signed char` does not result in lost data. It is possible that the conversion could result in a loss of data should the data be larger than the storage location.

A loss of data (truncation) can occur when converting from a signed type to a signed type with less precision. For example, the following code can result in truncation:

```
signed long int sl = LONG MAX;
signed char sc = (signed char)sl;
```

The C standard defines rules for integer promotions, integer conversion rank, and the usual arithmetic conversions. The intent of the rules is to ensure that the conversions result in the same numerical values, and that these values minimize surprises in the rest of the computation.

Stephen Michell 2015-6-5 11:33 PM

Deleted: 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. ... [11]

6.6.2 Guidance to language users

- Check the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type. Any conversion from a type with larger precision to a smaller precision type could potentially result in a loss of data. In some instances, this loss of precision is desired. Such cases should be explicitly acknowledged in comments. For example, the following code could be used to check whether a conversion from an unsigned integer to an unsigned character will result in a loss of precision:

```
unsigned int i;
unsigned char c;
…
if (i <= UCHAR_MAX) {  // check against the maximum value
                       // for an object of type unsigned char
   c = (unsigned char) i;
}
else {
   // handle error condition
}
```

- Close attention should be given to all warning messages issued by the compiler regarding multiple casts. Making a cast in C explicit will both remove the warning and acknowledge that the change in precision is on purpose.

6.7 String Termination [CJM]

6.7.1 Applicability to language

A string in C is composed of a contiguous sequence of characters terminated by and including a null character (a byte with all bits set to 0). Therefore strings in C cannot contain the null character except as the terminating character. Inserting a null character in a string either through a bug or through malicious action can truncate a string unexpectedly. Alternatively, not putting a null character terminator in a string can cause actions such as string copies to continue well beyond the end of the expected string. Overflowing a string buffer through the intentional lack of a null terminating character can be used to expose information or to execute malicious code.

6.7.2 Guidance to language users

Use the safer and more secure functions for string handling that are defined in normative Annex K from ISO/IEC 9899:2011 [4] or the ISO TR24731-2 — *Part II: Dynamic allocation functions*. Both of these define alternative string handling library functions to the current Standard C Library. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated. One implementation of these functions has been released as the Safe C Library.

6.8 Buffer Boundary Violation (Buffer Overflow) [HCB]

6.8.1 Applicability to language

A buffer boundary violation condition occurs when an array is indexed outside its bounds, or pointer arithmetic results in an access to storage that occurs outside the bounds of the object accessed.

In C, the subscript operator `[]` is defined such that `E1[E2]` is identical to `(*((E1)+(E2)))`, so that in either representation, the value in location `(E1+E2)` is returned. C does not perform bounds checking on arrays, so the following code:

```
int foo(const int i) {
    int x[] = {0,0,0,0,0,0,0,0,0,0};
    return x[i];
```

Stephen Michell 2015-6-5 11:34 PM
Formatted: Normal, Indent: Left: 1.27 cm, No bullets or numbering

Stephen Michell 2015-6-5 11:34 PM
Formatted: Font:(Default) Courier New

Stephen Michell 2015-6-5 11:34 PM
Formatted: Font:(Default) Courier New,

Stephen Michell 2015-6-5 11:34 PM
Formatted: Font:(Default) Courier New,

Stephen Michell 2015-6-5 11:34 PM
Formatted: Font:(Default) Courier New

Stephen Michell 2015-6-5 11:34 PM
Deleted: Use Ada's capabilities for user-defined scalar types and subtypes to avoid accidental mixing of logically incompatible value sets. ... [12]

Stephen Michell 2015-6-5 11:34 PM
Formatted: Normal, Space Before: 0 pt, After: 0 pt, No bullets or numbering

Stephen Michell 2015-6-5 11:35 PM
Deleted: With the exception of unsafe programming (see *4 Concepts*), this vulnerability is not applicable to Ada as strings in Ada are not delimited by a termination character. Ada programs that interface to languages that use null-terminated strings and manipulate such strings directly should apply the vulnerability mitigations recommended for that language.

```
}
```

will return whatever is in location `x[i]` even if, `i` were equal to -10 or 10 (assuming either subscript was still within the address space of the program). This could be sensitive information or even a return address, which if altered by changing the value of `x[-10]` or `x[10]`, could change the program flow.

The following code is more appropriate and would not violate the boundaries of the array `x`:

```
int foo( const int i) {
      int x[X_SIZE] = {0};
      if (i < 0 || i >= X_SIZE) {
        return ERROR_CODE;
      }
      else {
            return x[i];
      }
}
```

A buffer boundary violation may also occur when copying, initializing, writing or reading a buffer if attention to the index or addresses used are not taken. For example, in the following move operation there is a buffer boundary violation:

```
char buffer_src[]={"abcdefg"};
char buffer_dest[5]={0};
strcpy(buffer_dest, buffer_src);
```

the `buffer_src` is longer than the `buffer_dest`, and the code does not check for this before the actual copy operation is invoked. A safer way to accomplish this copy would be:

```
char buffer_src[]={"abcdefg"];
char buffer_dest[5]={0};
strncpy(buffer_dest, buffer_src, sizeof(buffer_dest) -1);
```

this would not cause a buffer bounds violation, however, because the destination buffer is smaller than the source buffer, the destination buffer will now hold "abcd", the 5th element of the array would hold the null character.

Stephen Michell 2015-6-6 12:01 AM
Formatted: Font:Not Bold

6.8.2 Guidance to language users

- Validate all input values.
- Check any array index before use if there is a possibility the value could be outside the bounds of the array.
- Use length restrictive functions such as `strncpy()` instead of `strcpy()`.
- Use stack guarding add-ons to detect overflows of stack buffers.
- Do not use the deprecated functions or other language features such as `gets()`.
- Be aware that the use of all of these measures may still not be able to stop all buffer overflows from happening. However, the use of them can make it much rarer for a buffer overflow to occur and much harder to exploit it.
- Use the safer and more secure functions for string handling from the normative annex K of C11 [4], Bounds-checking interfaces. The functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Optionally, failing functions call a

runtime-constraint handler to report the error. Data is never written past the end of an array. All string results are null terminated. In addition, these functions are re-entrant: they never return pointers to static objects owned by the function. Annex K also contains functions that address insecurities with the C input-output facilities.

6.9 Unchecked Array Indexing [XYZ]

6.9.1 Applicability to language

C does not perform bounds checking on arrays, so though arrays may be accessed outside of their bounds, the value returned is undefined and in some cases may result in a program termination. For example, in C the following code is valid, though, for example, if `i` has the value 10, the result is undefined:

```
int foo(const int i) {
    int t;
    int x[] = {0,0,0,0,0};
    t = x[i];
    return t;
}
```

The variable `t` will likely be assigned whatever is in the location pointed to by `x[10]` (assuming that `x[10]` is still within the address space of the program).

6.9.2 Guidance to language users

- Perform range checking before accessing an array since C does not perform bounds checking automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
- Use the safer and more secure functions for string handling from the normative annex K of C11 [4], *Bounds-checking interfaces*. These are alternative string handling library functions. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated.

6.10 Unchecked Array Copying [XYW]

6.10.1 Applicability to language

A buffer overflow occurs when some number of bytes (or other units of storage) is copied from one buffer to another and the amount being copied is greater than is allocated for the destination buffer.

In the interest of ease and efficiency, C library functions such as `memcpy(void * restrict s1, const void * restrict s2, size_t n)` and `memmove(void *s1, const void *s2, size_t n)` are used to copy the contents from one area to another. `memcpy()` and `memmove()` simply copy memory and no checks are made as to whether the destination area is large enough to accommodate the `n` units of data being copied. It is assumed that the calling routine has ensured that adequate space has been provided in

Stephen Michell 2015-6-6 12:02 AM
Deleted: . ... [13]

Stephen Michell 2015-6-6 12:02 AM
Formatted: Portuguese (Brazil)

Stephen Michell 2015-6-6 12:03 AM
Formatted: Font:Not Bold

Stephen Michell 2015-6-6 12:03 AM
Deleted: 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. ... [14]

Stephen Michell 2015-6-6 12:04 AM
Deleted: . ... [15]

the destination. Problems can arise when the destination buffer is too small to receive the amount of data being copied or if the indices being used for either the source or destination are not the intended indices.

6.10.2 Guidance to language users

- Perform range checking before calling a memory copying function such as `memcpy()` and `memmove()`. These functions do not perform bounds checking automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
- Use the safer and more secure functions for string handling from the normative annex K of C11 [4], Bounds-checking interfaces.

6.11 Pointer Type Conversions [HFC]

6.11.1 Applicability to language

C allows casting the value of a pointer to and from another data type. These conversions can cause unexpected changes to pointer values.

Pointers in C refer to a specific type, such as integer. If sizeof(int) is 4 bytes, and ptr is a pointer to integers that contains the value 0x5000, then ptr++ would make ptr equal to 0x5004. However, if ptr were a pointer to char, then ptr++ would make ptr equal to 0x5001. It is the difference due to data sizes coupled with conversions between pointer data types that cause unexpected results and potential vulnerabilities. Due to arithmetic operations, pointers may not maintain correct memory alignment or may operate upon the wrong memory addresses.

6.11.2 Guidance to language users

- Follow the advice provided by 6.12.5.
- Maintain the same type to avoid errors introduced through conversions.
- Heed compiler warnings that are issued for pointer conversion instances. The decision may be made to avoid all conversions so any warnings must be addressed. Note that casting into and out of "void *" pointers will most likely not generate a compiler warning as this is valid in C.

6.12 Pointer Arithmetic [RVG]

6.12.1 Applicability to language

When performing pointer arithmetic in C, the size of the value to add to a pointer is automatically scaled to the size of the type of the pointed-to object. For instance, when adding a value to the byte address of a 4-byte integer, the value is scaled by a factor 4 and then added to the pointer. The effect of this scaling is that if a pointer `P` points to the `i-th` element of an array object, then `(P) + N` will point to the `i+n-th` element of the array. Failing to understand how pointer arithmetic works can lead to miscalculations that result in serious errors, such as buffer overflows.

In C, arrays have a strong relationship to pointers. The following example will illustrate arithmetic in C involving a pointer and how the operation is done relative to the size of the pointer's target. Consider the following code snippet:

Stephen Michell 2015-6-6 12:04 AM
Deleted: With the exception of unsafe programming (see *4 Concepts*), this vulnerability is not applicable to Ada as Ada allows arrays to be copied by simple assignment (":="). The rules of the language ensure that no overflow can happen; instead, the exception Constraint_Error is raised if the target of the assignment is not able to contain the value assigned to it. Since array copy is provided by the language, Ada does not provide unsafe functions to copy structures by address and length

Stephen Michell 2015-6-6 12:06 AM
Formatted: English (UK)

Stephen Michell 2015-6-6 12:06 AM
Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm, Tabs: 1.27 cm, Left

Stephen Michell 2015-6-6 12:06 AM
Formatted: English (US)

Stephen Michell 2015-6-6 12:08 AM
Formatted: Normal, Space After: 0 pt, Widow/Orphan control, Hyphenate

Stephen Michell 2015-6-6 12:08 AM
Formatted: Font:(Default) +Theme Body

Stephen Michell 2015-6-6 12:08 AM
Formatted: Font:(Default) +Theme Body

Stephen Michell 2015-6-6 12:08 AM
Formatted: Font:(Default) +Theme Body

Stephen Michell 2015-6-6 12:08 AM
Formatted: Font:(Default) +Theme Body

Stephen Michell 2015-6-6 12:08 AM
Formatted: Font:(Default) +Theme Body

Stephen Michell 2015-6-6 12:08 AM
Formatted: Font:(Default) +Theme Body

Stephen Michell 2015-6-6 12:08 AM
Formatted: Font:(Default) +Theme Body

Stephen Michell 2015-6-6 12:07 AM
Deleted: 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). ... [16]

Stephen Michell 2015-6-6 12:08 AM
Formatted: Space After: 12 pt, Widow/Orphan control, Hyphenate

Stephen Michell 2015-6-6 12:08 AM
Formatted: Font:Not Italic, No underline, Font color: Auto

Stephen Michell 2015-6-6 12:08 AM
Formatted

Stephen Michell 2015-6-6 12:08 AM
Formatted

Stephen Michell 2015-6-6 12:08 AM
Deleted: <#>This vulnerability can be avoided in Ada by not using the features explicitly identified as unsafe. ... [17]

```
int buf[5];
int *buf_ptr = buf;
```

where the address of `buf` is 0x1234, after the assignment `buf_ptr` points to `buf[0]`. Adding 1 to `buf_ptr` will result in `buf_ptr` being equal to 0x1238 on a host where an `int` is 4 bytes; `buf_ptr` will then point to `buf[1]`. Not realizing that address operations will be in terms of the size of the object being pointed to can lead to address miscalculations and undefined behaviour.

6.12.2 Guidance to language users

- Consider an outright ban on pointer arithmetic due to the error-prone nature of pointer arithmetic.
- Verify that all pointers are assigned a valid memory address for use.

Stephen Michell 2015-6-6 12:10 AM

Deleted: With the exception of unsafe programming (see *4 Concepts*), this vulnerability is not applicable to Ada as Ada does not allow pointer arithmetic.

Stephen Michell 2015-6-6 12:11 AM

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm, Tabs: 1.27 cm, Left

6.13 Null Pointer Dereference [XYH]

6.13.1 Applicability to language

C allows memory to be dynamically allocated primarily through the use of `malloc()`, `calloc()`, and `realloc()`. Each will return the address to the allocated memory. Due to a variety of situations, the memory allocation may not occur as expected and a null pointer will be returned. Other operations or faults in logic can result in a memory pointer being set to null. Using the null pointer as though it pointed to a valid memory location can cause a segmentation fault and other unanticipated situations.

Space for 10000 integers can be dynamically allocated in C in the following way:

```
int *ptr = malloc(10000*sizeof(int));  // allocate space for 10000 ints
```

`malloc()` will return the address of the memory allocation or a null pointer if insufficient memory is available for the allocation. It is good practice after the attempted allocation to check whether the memory has been allocated via an `if` test against `NULL`:

```
if (ptr != NULL)   // check to see that the memory could be allocated
```

Memory allocations usually succeed, so neglecting this test and using the memory will usually work. That is why neglecting the null test will frequently go unnoticed. An attacker can intentionally create a situation where the memory allocation will fail leading to a segmentation fault.

Faults in logic can cause a code path that will use a memory pointer that was not dynamically allocated or after memory has been deallocated and the pointer was set to null as good practice would indicate.

6.13.2 Guidance to language users

- Check whether a pointer is null before dereferencing it. As this can be overly extreme in many cases (such as in a `for` loop that performs operations on each element of a large segment of memory), judicious checking of the value of the pointer at key strategic points in the code is recommended.

6.14 Dangling Reference to Heap [XYK]

Stephen Michell 2015-6-6 12:12 AM
Deleted: . ... [18]

6.14.1 Applicability to language

C allows memory to be dynamically allocated primarily through the use of `malloc()`, `calloc()`, and `realloc()`. C allows a considerable amount of freedom in accessing the dynamic memory. Pointers to the dynamic memory can be created to perform operations on the memory. Once the memory is no longer needed, it can be released through the use of `free()`. However, freeing the memory does not prevent the use of the pointers to the memory and issues can arise if operations are performed after memory has been freed.

Consider the following segment of code:

```
int foo() {
  int *ptr = malloc (100*sizeof(int));/* allocate space for 100 integers*/
  if (ptr != NULL) {/* check to see that the memory could be allocated */
                    /* perform some operations on the dynamic memory */
    free (ptr);     /* memory is no longer needed, so free it */
                    /* program continues performing other operations */
    ptr[0] = 10;    /* ERROR – memory being used after released */
    …
  }
  …
  }
```

The use of memory in C after it has been freed is undefined. Depending on the execution path taken in the program, freed memory may still be free or may have been allocated via another `malloc()` or other dynamic memory allocation. If the memory that is used is still free, use of the memory may be unnoticed. However, if the memory has been reallocated, altering of the data contained in the memory can result in data corruption. Determining that a dangling memory reference is the cause of a problem and locating it can be difficult.

Setting and using another pointer to the same section of dynamically allocated memory can also lead to undefined behaviour. Consider the following section of code:

```
int foo() {
  int *ptr = malloc (100*sizeof(int));/* allocate space for 100 integers */
  if (ptr != NULL) {                   /* check to see that the memory
                                          could be allocated */
    int ptr2 = &ptr[10];               /* set ptr2 to point to the 10th
                                          element of the allocated memory */
      …                                /* perform some operations on the
                                          dynamic memory */
    free (ptr);                        /* memory is no longer needed */
    ptr = NULL;                        /* set ptr to NULL to prevent ptr
                                          from being used again */
    …                                  /* program continues performing
                                          other operations */
    ptr2[0] = 10;                      /* ERROR – memory is being used
                                       after it has been released via ptr2 */
    …
    }
```

```
    return (0);
    }
```

Dynamic memory was allocated via a `malloc()` and then later in the code, `ptr2` was used to point to an address in the dynamically allocated memory. After the memory was freed using `free(ptr)` and the good practice of setting `ptr` to `NULL` was followed to avoid a dangling reference by `ptr` later in the code, a dangling reference still existed using `ptr2`.

6.14.2 Guidance to language users

- Follow the advice provided by 6.15.2.
- Set a freed pointer to null immediately after a `free()` call, as illustrated in the following code:

```
free (ptr);
ptr = NULL;
```

- Do not create and use additional pointers to dynamically allocated memory.
- Only reference dynamically allocated memory using the pointer that was used to allocate the memory.

6.15 Arithmetic Wrap-around Error [FIF]

6.15.1 Applicability to language

Given the limited size of any computer data type, continuously adding one to the data type eventually will cause the value to go from a the maximum possible value to a small value. C permits this to happen without any detection or notification mechanism.

C is often used for bit manipulation. Part of this is due to the capabilities in C to mask bits and shift them. Another part is due to the relative closeness C has to assembly instructions. Manipulating bits on a signed value can inadvertently change the sign bit resulting in a number potentially going from a large positive value to a large negative value.

For example, consider the following code for a `short int` containing 16 bits:

```
int foo( short int i ) {
    i++;
    return i;
}
```

Calling `foo` with the value of 32767 would cause undefined behaviour, such as wrapping to -32768. Manipulating a value in this way can result in unexpected results such as overflowing a buffer.

In C, bit shifting by a value that is greater than the size of the data type or by a negative number is undefined. The following code, where a `int` is 16 bits, would be undefined when `j` is greater than or equal to 16 or negative:

```
int foo( int i, const int j ) {
    return i>>j;
}
```

6.15.2 Guidance to language users

- Be aware that any of the following operators have the potential to wrap in C:

Stephen Michell 2015-6-6 12:13 AM
Deleted: Use of Unchecked_Deallocation can cause dangling references to the heap. The vulnerabilities described in 6.15 exist in Ada, when this feature is used, since Unchecked_Deallocation may be applied even though there are outstanding references to the deallocated object. ... [19]

Stephen Michell 2015-6-6 12:14 AM
Formatted: Indent: Left: 1.27 cm, No bullets or numbering

Stephen Michell 2015-6-6 12:14 AM
Deleted: <#>Use local access types where possible. ... [20]

Stephen Michell 2015-6-6 12:16 AM
Deleted: With the exception of unsafe programming (see *4 Concepts*), this vulnerability is not applicable to Ada as wrap-around arithmetic in Ada is limited to modular types. Arithmetic operations on such types use modulo arithmetic, and thus no such operation can create an invalid value of the type. ... [21]

Stephen Michell 2015-6-6 12:15 AM
Formatted: Heading 3

Stephen Michell 2015-6-6 12:16 AM
Formatted: List Paragraph, Space Before: 6 pt, After: 6 pt, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm, No widow/orphan control, Don't hyphenate, Don't allow hanging

```
a + b     a - b     a * b     a++       a--       a += b
a -= b    a *= b    a << b    a >> b    -a
```

- Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. These techniques can be omitted if it can be shown at compile time that overflow or underflow is not possible.
- Only conduct bit manipulations on unsigned data types. The number of bits to be shifted by a shift operator should lie between 1 and (n-1), where n is the size of the data type.

6.16 Using Shift Operations for Multiplication and Division [PIK]

6.16.1 Applicability to language

The issues for C are well defined in the main body of this document in ***Error! Reference source not found.***. Also ee, ***Error! Reference source not found.***

Stephen Michell 2015-6-6 12:19 AM
Deleted: With the exception of unsafe programming (see *4 Concepts*), this vulnerability is not applicable to Ada as shift operations in Ada are limited to the modular types declared in the standard package Interfaces, which are not signed entities

6.16.2 Guidance to language users

The guidance for C users is well defined in the main body of this document in ***Error! Reference source not found.***. Also see, ***Error! Reference source not found.***.

6.17 Choice of Clear Names [NAI]

6.17.1 Applicability to language

C is somewhat susceptible to errors resulting from the use of similarly appearing names. C does require the declaration of variables before they are used. However, C allows scoping so that a variable that is not declared locally may be resolved to some outer block and a human reviewer may not notice that resolution. Variable name length is implementation specific and so one implementation may resolve names to one length whereas another implementation may resolve names to another length resulting in unintended behaviour.

As with the general case, calls to the wrong subprogram or references to the wrong data element (when missed by human review) can result in unintended behavior.

Stephen Michell 2015-6-6 12:20 AM
Formatted: Normal, Space After: 0 pt, Widow/Orphan control, Hyphenate

6.17.2 Guidance to language users

Stephen Michell 2015-6-6 12:20 AM
Deleted: There are two possible issues: the use of the identical name for different purposes (overloading) and the use of similar names for different purposes ... [22]

- Use names that are clear and non-confusing.
- Use consistency in choosing names.
- Keep names short and concise in order to make the code easier to understand.
- Choose names that are rich in meaning.
- Keep in mind that code will be reused and combined in ways that the original developers never imagined.
- Make names distinguishable within the first few characters due to scoping in C. This will also assist in averting problems with compilers resolving to a shorter name than was intended.
- Do not differentiate names through only a mixture of case or the presence/absence of an underscore character.
- Avoid differentiating through characters that are commonly confused visually such as ‘O’ and ‘0’, ‘l’ (lower case ‘L’), ‘I’ (capital ‘I’) and ‘1’, ‘S’ and ‘5’, ‘Z’ and ‘2’, and ‘n’ and ‘h’.
- Coding guidelines should be developed to define a common coding style and to avoid the above dangerous practices.

6.18 Dead store [WXQ]

Stephen Michell 2015-6-6 12:21 AM
Deleted: This vulnerability can be avoided or mitigated in Ada in the following ways: ... [23]

6.18.1 Applicability to language

This vulnerability exists in Ada as described in section 6.20, with the exception that in Ada if a variable is read by a different thread (task) than the thread that wrote a value to the variable it is not a dead store. Simply marking a variable as being Volatile is usually considered to be too error-prone for inter-thread (task) communication by the Ada community, and Ada has numerous facilities for safer inter thread communication.

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

The error in 6.20.3 that the planned reader misspells the name of the store is possible but highly unlikely in Ada since all objects must be declared and typed and the existence of two objects with almost identical names and compatible types (for assignment) in the same scope would be readily detectable.

6.18.2 Guidance to Language Users

- Use Ada compilers that detect and generate compiler warnings for unused variables or use static analysis tools to detect such problems.

6.19 Unused Variable [YZS]

6.19.1 Applicability to language

This vulnerability exists in Ada as described in section 6.21, although Ada compilers do exist that detect and generate compiler warnings for unused variables.

6.19.2 Guidance to language users

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

6.20 Identifier Name Reuse [YOW]

6.20.1 Applicability to language

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

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

The mechanism of failure identified in section 6.22.3 regarding the declaration of non-unique identifiers in the same scope cannot occur in Ada because all characters in an identifier are significant.

6.20.2 Guidance to language users

- Use *expanded names* whenever confusion may arise.

- Use Ada compilers that generate compile time warnings for declarations in inner scopes that hide declarations in outer scopes.
- Use static analysis tools that detect the same problem.

6.21 Namespace Issues [BJL]

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

6.22 Initialization of Variables [LAV]

6.22.1 Applicability to language

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

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

The checks mandated by the type system apply to the use of uninitialized variables as well. Use of an out-of-bounds value in relevant contexts causes an exception, regardless of the origin of the faulty value. (See ***Error! Reference source not found.***, regarding exception handling.) Thus, the only remaining vulnerability is the potential use of a faulty but subtype-conformant value of an uninitialized variable, since it is technically indistinguishable from a value legitimately computed by the application.

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.36 Ignored Error Status and Unhandled Exceptions [OYB]*

For record types, default initializations may be specified as part of the type definition.

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

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

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

6.22.2 Guidance to language users

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

- If the compiler has a mode that detects use before initialization, then this mode should be enabled and any such warnings should be treated as errors.
- Where appropriate, explicit initializations or default initializations can be specified.
- The pragma Normalize_Scalars can be used to cause out-of-range default initializations for scalar variables.
- The 'Valid attribute can be used to identify out-of-range values caused by the use of uninitialized variables, without incurring the raising of an exception.

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

6.23 Operator Precedence/Order of Evaluation [JCW]

6.23.1 Applicability to language

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

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

6.23.2 Guidance to language users

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

6.24 Side-effects and Order of Evaluation [SAM]

6.24.1 Applicability to language

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

There is the possibility though to have side effects through function calls in expressions where the function modifies globally visible variables. 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 order of evaluation of expressions with operands of the same precedence level, the order of association is left-to-right. The operands of a binary operation are also evaluated in an arbitrary order, as happens for the parameters of any function call. In the case of user-defined operators with side effects, this implementation dependency can cause unpredictability of the side effects.

6.24.2 Guidance to language users

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

6.25 Likely Incorrect Expression [KOA]

6.25.1 Applicability to language

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

The examples given in 6.27 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 (for example, "**and then**" and "**or else**" vs. "**and**" and "**or**"), as in

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

-- should have used "and then" to avoid dereferencing null
```

6.25.2 Guidance to language users

- Compilers and other static analysis tools can detect some cases (such as the preceding example).
- 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.

6.26 Dead and Deactivated Code [XYQ]

6.26.1 Applicability to language

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

6.26.2 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. For example, given the following:

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

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

6.27 Switch Statements and Static Analysis [CLL]

6.27.1 Applicability to language

With the exception of unsafe programming (see *4 Language concepts*) and the use of default cases, this vulnerability is not applicable to Ada as Ada ensures that a case statement provides exactly one alternative for each value of the expression's subtype. This restriction is enforced at compile time. The **others** clause may be used as the last choice of a case statement to capture any remaining values of the case expression type that are not covered by the preceding case choices. If the value of the expression is outside of the range of this subtype (for example, 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.

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *4 Concepts*

The remaining vulnerability is that unexpected values are captured by the **others** clause or a subrange as case choice. For example, when the range of the type Character was extended from 128 characters to the 256 characters in the Latin-1 character type, an **others** clause for a **case** statement with a Character type case expression originally written to capture cases associated with the 128 characters type now captures the 128 additional cases introduced by the extension of the type Character. Some of the new characters may have needed to be covered by the existing case choices or new case choices.

6.27.2 Guidance to language users

- For **case** statements and aggregates, avoid the use of the **others** choice.
- For **case** statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition. [2]

6.28 Demarcation of Control Flow [EOJ]

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

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

6.29 Loop Control Variables [TEX]

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

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *4 Concepts*

6.30 Off-by-one Error [XZH]

6.30.1 Applicability to language

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

A **for loop** in Ada does not require the programmer to specify a conditional test for loop termination. Instead, the starting and ending value of the loop are specified which eliminates this source of off-by-one errors. A **while loop** however, lets the programmer specify the loop termination expression, which could be susceptible to an off-by-one error.

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.

6.30.2 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.

6.31 Structured Programming [EWD]

6.31.1 Applicability to language

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

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

6.31.2 Guidance to language users

Avoid the use of **goto**, **loop exit** statements, **return** statements in **procedure**s and more than one **return** statement in a **function** If not following this guidance caused the function code to be clearer – short of appropriate restructuring – then multiple exit points should be used.

6.32 Passing Parameters and Return Values [CSJ]

6.32.1 Applicability to language

Ada employs the mechanisms (for example, modes **in**, **out** and **in out**) that are recommended in Section 6.34. These mode definitions are not optional, mode **in** being the default. The remaining vulnerability is aliasing when a large object is passed by reference.

6.32.2 Guidance to language users

- Follow avoidance advice in Section 6.24.

6.33 Dangling References to Stack Frames [DCM]

6.33.1 Applicability to language

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

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

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

6.33.2 Guidance to language users

- Only use 'Address attribute on static objects (for example, 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.
- Use ‘Access attribute in preference to ‘Address.

6.34 Subprogram Signature Mismatch [OTR]

6.34.1 Applicability to language

There are two concerns identified with this vulnerability. The first is the corruption of the execution stack due to the incorrect number or type of actual parameters. The second is the corruption of the execution stack due to calls to externally compiled modules.

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

Caution must be used when specifying default expressions for formal parameters, as their use may result in successful compilation of subprogram calls with an incorrect signature. The execution stack will not be corrupted in this event but the program may be executing with unexpected values.

When calling externally compiled modules that are Ada program units, the type matching and subprogram interface signatures are monitored and checked as part of the compilation and linking of the full application. When calling externally compiled modules in other programming languages, additional steps are needed to ensure that the number and types of the parameters for these external modules are correct.

6.34.2 Guidance to language users

- 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.

6.35 Recursion [GDL]

6.35.1 Applicability to language

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

6.35.2 Guidance to language users

- If recursion is used, then 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.

6.36 Ignored Error Status and Unhandled Exceptions [OYB]

6.36.1 Applicability to language

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

As described in 6.38, there is some complexity in understanding the exception handling methodology especially with respect to object-oriented programming and multi-threaded execution.

6.36.2 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.

6.37 Fault Tolerance and Failure Strategies [REW]

6.37.1 Applicability to language

An Ada system that consists of multiple tasks is subject to the same hazards as multithreaded systems in other languages. A task that fails, for example, because its execution violates a language-defined check, terminates quietly.

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

6.37.2 Guidance to language users

- Include exception handlers for every task, so that their unexpected termination can be handled and possibly communicated to the execution environment.
- 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.

6.38 Type-breaking Reinterpretation of Data [AMV]

6.38.1 Applicability to language

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

6.38.2 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.

6.39 Memory Leak [XYL]

6.39.1 Applicability to language

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

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

6.39.2 Guidance to language users

- Use storage pools 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.

6.40 Templates and Generics [SYM]

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

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

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

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *4 Concepts*

6.41 Inheritance [RIP]

6.41.1 Applicability to language

The vulnerability documented in Section 6.43 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.

6.41.2 Guidance to language users

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

6.42 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.

6.43 Argument Passing to Library Functions [TRJ]

6.43.1 Applicability to language

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

However, to the extent that the preclusion of values can be expressed as part of the type system of Ada, the preconditions are checked by the compiler statically or dynamically and thus are no longer vulnerabilities. For example, any range constraint on values of a parameter can be expressed in Ada by means of type or subtype declarations. Type violations are detected at compile time, subtype violations cause run-time exceptions.

6.43.2 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.
- Library providers should specify the response to invalid values.

6.44 Inter-language Calling [DJS]

6.44.1 Applicability to Language

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

6.44.2 Guidance to Language Users

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

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

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

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *4 Concepts*

6.46 Library Signature [NSQ]

6.46.1 Applicability to language

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

Even with the use of **pragma** Import, **pragma** Export and **pragma** Convention the vulnerabilities stated in Section 6.48 are possible. Names and number of parameters change under maintenance; calling conventions change as compilers are updated or replaced, and languages for which Ada does not specify a calling convention may be used.

6.46.2 Guidance to language users

- The mitigation mechanisms of Section 6.48.5 are applicable.

6.48 Unanticipated Exceptions from Library Routines [HJW]

6.48.1 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 a caller.

If the interface between the Ada units and the library routine being called does not adequately address the issue of naming, generation and delivery of exceptions across the interface, then the vulnerabilities as expressed in Section 6.49 apply.

6.47.2 Guidance to language users

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

6.48 Pre-Processor Directives [NMP]

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

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

6.49.1 Applicability to Language

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

6.49.2 Guidance to Language Users

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

6.50 Provision of Inherently Unsafe Operations [SKL]

6.50.1 Applicability to Language

In recognition of the occasional need to step outside the type system or to perform "risky" operations, Ada provides clearly identified language features to do so. Examples include the generic Unchecked_Conversion for unsafe type-conversions or Unchecked_Deallocation for the deallocation of heap objects regardless of the existence of surviving references to the object. If unsafe programming is employed in a unit, then the unit needs to specify the respective generic unit in its context clause, thus identifying potentially unsafe units. Similarly, there are ways to create a potentially unsafe global pointer to a local object, using the Unchecked_Access attribute.

6.51 Obscure Language Features [BRS]

6.51.1 Applicability to language

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

6.51.2 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.

6.52 Unspecified Behaviour [BQF]

6.52.1 Applicability to language

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

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.

6.52.2 Guidance to language users

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

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

- For 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 on Initialization of Variables [LAV] for a discussion of uninitialized variables in Ada, a common cause of a bounded error.

6.53 Undefined Behaviour [EWF]

6.53.1 Applicability to language

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

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

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

The full list is given in the index of the Ada 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, and other uncontrolled events.

6.53.2 Guidance to language users

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

- 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 6.55.5 are applicable here.

6.54 Implementation-Defined Behaviour [FAB]

6.54.1 Applicability to language

There are a number of situations in Ada where the language semantics are implementation defined, to allow the implementation to choose an efficient mechanism, or to match the capabilities of the target environment. Each of these situations is identified in Annex M of the Ada 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.

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

- 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.

6.55 Deprecated Language Features [MEM]

6.55.1 Applicability to language

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

6.55.2 Guidance to language users

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

6.56 Concurrency – Activation [CGA]

6.56.1 Applicability to language

6.56.2 Guidance to language users

6.57 Concurrency – Directed termination [CGT]

6.57.1 Applicability to language

6.57.2 Guidance to language users

6.58 Concurrent Data Access [CGX]

6.58.1 Applicability to language

6.58.2 Guidance to language users

6.59 Concurrency – Premature Termination [CGS]

6.59.1 Applicability to language

6.59.2 Guidance to language users

6.60 Protocol Lock Errors [CGM]

6.60.1 Applicability to language

6.60.2 Guidance to language users

6.61 Uncontrolled Format String [SHL]

7 Language specific vulnerabilities for Ada

8 Implications for standardization

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

- Some languages (for example, 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 (see *6.22 Initialization of Variables [LAV]*).
- **Pragma** Restrictions could be extended to allow the use of these features to be statically checked (see *6.31 Structured Programming [EWD]*).
- When appropriate, language-defined checks should be added to reduce the possibility of multiple outcomes from a single construct, such as by disallowing side-effects in cases where the order of evaluation could affect the result (see *6.52 Unspecified Behaviour [BQF]*).
- When appropriate, language-defined checks should be added to reduce the possibility of erroneous execution, such as by disallowing unsynchronized access to shared variables (see *6.53 Undefined Behaviour [EWF]*).
- Language standards should specify relatively tight boundaries on implementation-defined behaviour whenever possible, and the standard should highlight what levels represent a portable minimum

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.22 Initialization of Variables [LAV]*

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.31 Structured Programming [EWD]*

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.52 Unspecified Behaviour [BQF]*

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.53 Undefined Behaviour [EWF]*

capability on which programmers may rely. For languages like Ada that allow user declaration of numeric types, the number of predefined numeric types should be minimized (for example, strongly discourage or disallow declarations of Byte_Integer, Very_Long_Integer, and similar, in **package** Standard) (see *6.54 Implementation-Defined Behaviour [FAB]*).

- Ada could define a **pragma** Restrictions identifier No_Hiding that forbids the use of a declaration that result in a local homograph (see *6.20 Identifier Name Reuse [YOW]*).
- Add the ability to declare in the specification of a function that it is pure, that is, it has no side effects (see *6.24 Side-effects and Order of Evaluation [SAM]*).
- **Pragma** Restrictions could be extended to restrict the use of 'Address attribute to library level static objects (see *6.33 Dangling References to Stack Frames [DCM]*).
- Future standardization of Ada should consider implementing a language-provided reference counting storage management mechanism for dynamic objects (see *6.39 Memory Leak [XYL]*).
- Provide mechanisms to prevent further extensions of a type hierarchy (see *6.41 Inheritance [RIP]*).
- Future standardization of Ada should consider support for arbitrary pre- and postconditions (see *6.43 Argument Passing to Library Functions [TRJ]*).
- 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 (see *6.46 Library Signature [NSQ]*).

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.54 Implementation-Defined Behaviour [FAB]*

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.20 Identifier Name Reuse [YOW]*

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.24 Side-effects and Order of Evaluation [SAM]*

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.33 Dangling References to Stack Frames [DCM]*

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.39 Memory Leak [XYL]*

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.41 Inheritance [RIP]*

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.43 Argument Passing to Library Functions [TRJ]*

Stephen Michell 2017-2-20 9:16 AM
Deleted: *6.48 Library Signature [NSQ]*

Stephen Michell 2017-2-20 9:16 AM
Formatted: Font:Italic, Underline, Font color: Custom Color(RGB(0,112,192))

Bibliography

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

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

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

[7] ISO/IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

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

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

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

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

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

[20] Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B).December 1992.

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

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

[23] J Barnes, High Integrity Software - the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

1. Lecture Notes on Computer Science 5020, "Ada 2012 Rationale: The Language, the Standard Libraries," John Barnes, Springer, 2012. ???????

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

[29] Lions, J. L. ARIANE 5 Flight 501 Failure Report. Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

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

[34] Goldberg, David, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, vol 23, issue 1 (March 1991), ISSN 0360-0300, pp 5-48.

[35] IEEE Standards Committee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008. Institute of Electrical and Electronics Engineers, New York, 2008.

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

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

[38] GAO Report, Patriot *Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, B-247094, Feb. 4, 1992, http://archive.gao.gov/t2pbat6/145960.pdf

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

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

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

[43] Ada 95 Quality and Style Guide, SPC-91061-CMC, version 02.01.01. Herndon, Virginia: Software Productivity Consortium, 1992. Available from: http://www.adaic.org/docs/95style/95style.pdf

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

[45] Subramanian, S., Tsai, W.-T., & Rayadurgam, S. (1998). Design Constraint Violation Detection in Safety-Critical Systems. The 3rd IEEE International Symposium on High-Assurance Systems Engineering , 109 - 116.

[46] Lundqvist, K and Asplund, L., "*A Formal Model of a Run-Time Kernel for Ravenscar*", The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999

Index