ISO/IEC JTC 1/SC 22 N 0000

Date: 2012-01-16

ISO/IEC TR 24772

Edition 2

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

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: (20) 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.

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

Introduction

All programming languages contain constructs that are incompletely specified, exhibit undefined behaviour, are implementation-dependent, or are difficult to use correctly. The use of those constructs may therefore give rise to *vulnerabilities*, as a result of which, software programs can execute differently than intended by the writer. In some cases, these vulnerabilities can compromise the safety of a system or be exploited by attackers to compromise the security or privacy of a system.

This Technical Report is intended to provide guidance spanning multiple programming languages, so that application developers will be better able to avoid the programming constructs that lead to vulnerabilities in software written in their chosen 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 or to select a programming language that avoids anticipated problems.

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.

Furthermore, to focus its limited resources, the working group developing this report decided to defer comprehensive treatment of several subject areas until future editions of the report. These subject areas include:

- Object-oriented language features (Although some simple issues related to inheritance are described in RIP)
- Numerical analysis (although some simple items regarding the use of floating point are described in PLF)
- Inter-language operability

Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

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 are described in a generic manner that is applicable to a broad range of programming languages.

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/IEC 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*

3. Terms and definitions, symbols and conventions

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382–1 and the following apply. Other terms are defined where they appear in *italic* type.

3.1.1 Communication

3.1.1.1
protocol
set of rules and supporting structures for the interaction of threads

> **Note 1:** A protocol can be tightly embedded and rely upon data in memory and hardware to control interaction of threads or can be applied to more loosely coupled arrangements, such as message communication spanning networks and computer systems.

3.1.1.2
stateless protocol
communication or cooperation between threads where no state is preserved in the protocol itself (example HTTP or direct access to a shared resource)

Note 1: Since most interaction between threads require that state be preserved, the cooperating threads must use values of the resources(s) themselves or add additional communication exchanges to maintain state. Stateless protocols require that the application provide explicit resource protection and locking mechanisms to guarantee the correct creation, view, access to, modification of, and destruction of the resource – for example, the state needed for correct handling of the resource.

3.1.2 Execution model

3.1.2.1
thread
sequential stream of execution

Note 1: Although the term thread is used here and the context portrayed is that of shared memory threads executing as part of a process, everything documented applies equally to other variants of concurrency such as interrupt handlers being enabled by a process, processes being created on the same system using operating system routines, or processes created as a result of distributed messages sent over a network. The mitigation approaches will be similar to those listed in the relevant vulnerability descriptions, but the implications for standardization would be dependent on how much language support is provided for the programming of the concurrent system.

3.1.2.2
thread activation
creation and setup of a thread up to the point where the thread begins execution

Note 1: Threads may depend upon one or more other threads to define its access to other objects to be accessed and to determine its duration.

3.1.2.3
activated thread
thread that is created and then begins execution as a result of thread activation

3.1.2.4
activating thread
thread that exists first and makes the library calls or contains the language syntax that causes the activated thread to be activated

Note 1: The Activating Thread may or may not wait for the Activated Thread to finish activation and may or may not check for errors if the activation fails. The Activating Thread may or may not be permitted to terminate until after the Activated Thread terminates.

3.1.2.5
static thread activation

creation and initiation of a thread by program initiation, an operating system or runtime kernel, or by another thread as part of a declarative part of the thread before it begins execution

Note 1: In static activation, a static analysis can determine exactly how many threads will be created and how much resource, in terms of memory, processors, cpu cycles, priority ranges and inter-thread communication structures, will be needed by the executing program before the program begins.

3.1.2.6
dynamic thread activation
creation and initiation of a thread by another thread (including the main program) as an executable, repeatable command, statement or subprogram call

3.1.2.7
thread abort
request to stop and shut down a thread immediately

Note 1: The request is asynchronous if from another thread, or synchronous if from the thread itself. The effect of the abort request (e.g. whether it is treated as an exception) and its immediacy (i.e., how long the thread may continue to execute before it is shut down) depend on language-specific rules. Immediate shutdown minimizes latency but may leave shared data structures in a corrupted state.

3.1.2.8
termination directing thread
thread (including the OS) that requests the abort of one or more threads

3.1.2.9
thread termination
completion and orderly shutdown of a thread, where the thread is permitted to make data objects consistent, release any acquired resources, and notify any dependent threads that it is terminating

Note 1: There are a number of steps in the termination of a thread as listed below, but depending upon the multithreading model, some of these steps may be combined, may be explicitly programmed, or may be missing.

- The termination of programmed execution of the thread, including termination of any synchronous communication;
- the finalization of the local objects of the thread;
- waiting for any threads that may depend on the thread to terminate;
- finalization of any state associated with dependent threads;
- notification that finalization is complete, including possible notification of the activating task;
- removal and cleanup of thread control blocks and any state accessible by the thread or by other threads in outer scopes.

3.1.2.10
terminated thread
thread that is being halted from any further execution

3.1.2.11
master thread
thread which must wait for a terminated thread before it can take further execution steps (including termination of itself)

3.1.2.12
process
single execution of a program, or portion of an application

Note 1: Processes do not normally share a common memory space, but often share

- processor,
- network,
- operating system,
- filing system,
- environment variables, or
- other resources.

Processes are usually started and stopped by an operating system and may or may not interact with other processes. A process may contain multiple threads.

3.1.3 Properties

3.1.3.1
software quality
degree to which software implements the requirements described by its specification and the degree to which the characteristics of a software product fulfill its requirements

3.1.3.2
predictable execution
property of the program such that all possible executions have results that can be predicted from the source code

3.1.4 Safety

3.1.4.1
safety hazard
potential source of harm

Note 1: IEC 61508–4: defines a “Hazard” as a “potential source of harm”, where “harm” is “physical injury or damage to the health of people either directly or indirectly as a result of damage to property or to the environment”. Some derived standards, such as UK Defence Standard 00-56, broaden the definition of “harm” to include material and environmental damage (not just harm to people caused by property and environmental damage).

3.1.4.2
safety-critical software
software for applications where failure can cause very serious consequences such as human injury or death

Note 1: IEC 61508–4: defines “Safety-related software” as “software that is used to implement safety functions in a safety-related system. Notwithstanding that in some domains a distinction is made between safety-related (can lead to any harm) and safety-critical (life threatening), this Technical Report uses the term *safety-critical* for all vulnerabilities that can result in safety hazards.

3.1.5 Vulnerabilities

3.1.5.1

application vulnerability

security vulnerability or safety hazard, or defect

3.1.5.2

language vulnerability

property (of a programming language) that can contribute to, or that is strongly correlated with, application vulnerabilities in programs written in that language

Note 1: The term "property" can mean the presence or the absence of a specific feature, used singly or in combination. As an example of the absence of a feature, encapsulation (control of where names can be referenced from) is generally considered beneficial since it narrows the interface between modules and can help prevent data corruption. The absence of encapsulation from a programming language can thus be regarded as a vulnerability. Note that a property together with its complement can both be considered language vulnerabilities. For example, automatic storage reclamation (garbage collection) can be a vulnerability since it can interfere with time predictability and result in a safety hazard. On the other hand, the absence of automatic storage reclamation can also be a vulnerability since programmers can mistakenly free storage prematurely, resulting in dangling references.

3.1.5.3

security vulnerability

weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat

3.2 Symbols and conventions

3.2.1 Symbols

For the purposes of this document, the symbols given in ISO/IEC 80000–2 apply. Other symbols are defined where they appear in this document.

3.2.2 Conventions

Programming language token and syntactic token appear in `courier` font.

4. Basic Concepts

4.1 Purpose of this Technical Report

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.

This Technical Report does not address software engineering and management issues such as how to design and implement programs, use configuration management tools, use managerial processes, and perform process improvement. Furthermore, the specification of properties and applications to be assured are not treated.

While this Technical Report does not discuss specification or design issues, there is recognition that boundaries among the various activities are not clear-cut. This Technical Report seeks to avoid the debate about where low-level design ends and implementation begins by treating selected issues that some might consider design issues rather than coding issues.

The body of this Technical Report provides users of programming languages with a language-independent overview of potential vulnerabilities in their usage. Annexes describe how the general observations apply to specific languages.

4.2 Intended Audience

The intended audience for this Technical Report are those who are concerned with assuring the predictable execution of the software of their system; that is, those who are developing, qualifying, or maintaining a software system and need to avoid language constructs that could cause the software to execute in a manner other than intended.

Developers of applications that have clear safety, security or mission criticality are expected to be aware of the risks associated with their code and could use this Technical Report to ensure that their development practices address the issues presented by the chosen programming languages, for example by subsetting or providing coding guidelines.

It should not be assumed, however, that other developers can ignore this Technical Report. A weakness in a non-critical application may provide the route by which an attacker gains control of a system or otherwise disrupts co-hosted applications that are critical. It is hoped that all developers would use this Technical Report to ensure that common vulnerabilities are removed or at least minimized from all applications.

Specific audiences for this International Technical Report include developers, maintainers and regulators of:

- Safety-critical applications that might cause loss of life, human injury, or damage to the environment.
- Security-critical applications that must ensure properties of confidentiality, integrity, and availability.
- Mission-critical applications that must avoid loss or damage to property or finance.
- Business-critical applications where correct operation is essential to the successful operation of the business.
- Scientific, modeling and simulation applications which require high confidence in the results of possibly complex, expensive and extended calculation.

4.3 How to Use This Document

This Technical Report gathers descriptions of programming language vulnerabilities, as well as selected application vulnerabilities, which have occurred in the past and are likely to occur again. Each vulnerability and its possible mitigations are described in the body of the report in a language-independent manner, though illustrative examples may be language specific. In addition, annexes for particular languages describe the vulnerabilities and their mitigations in a manner specific to the language.

Because new vulnerabilities are always being discovered, it is anticipated that this Technical Report will be revised and new descriptions added. For that reason, a scheme that is distinct from sub-clause numbering has been adopted to identify the vulnerability descriptions. Each description has been assigned an arbitrarily generated, unique three-letter code. These codes should be used in preference to sub-clause numbers when referencing descriptions because they will not change as additional descriptions are added to future editions of this Technical Report.

The main part of this Technical Report contains descriptions that are intended to be language-independent to the greatest possible extent. Annexes apply the generic guidance to particular programming languages.

This Technical Report has been written with several possible usages in mind:

- Programmers familiar with the vulnerabilities of a specific language can reference the guide for more generic descriptions and their manifestations in less familiar languages.
- Tool vendors can use the three-letter codes as a succinct way to “profile” the selection of vulnerabilities considered by their tools.
- Individual organizations may wish to write their own coding standards intended to reduce the number of vulnerabilities in their software products. The guide can assist in the selection of vulnerabilities to be addressed in those standards and the selection of coding guidelines to be enforced.
- Organizations or individuals selecting a language for use in a project may want to consider the vulnerabilities inherent in various candidate languages.

The descriptions include suggestions for ways of avoiding the vulnerabilities. Some are simply the avoidance of particular coding constructs, but others may involve increased review or other verification and validation methods. Source code checking tools can be used to automatically enforce some coding rules and standards.

Clause 2 provides Normative references, and Clause 3 provides Terms, definitions, symbols and conventions.

Clause 4 provides the basic concepts used for this Technical Report.

Clause 5, *Vulnerability Issues*, provides rationale for this Technical Report and explains how many of the vulnerabilities occur.

Clause 6, *Programming Language Vulnerabilities*, provides language-independent descriptions of vulnerabilities in programming languages that can lead to application vulnerabilities. Each description provides:

- a summary of the vulnerability,
- characteristics of languages where the vulnerability may be found,
- typical mechanisms of failure,

- techniques that programmers can use to avoid the vulnerability, and
- ways that language designers can modify language specifications in the future to help programmers mitigate the vulnerability.

Clause 7, *Application Vulnerabilities*, provides descriptions of selected application vulnerabilities which have been found and exploited in a number of applications and which have well known mitigation techniques, and which result from design decisions made by coders in the absence of suitable language library routines or other mechanisms. For these vulnerabilities, each description provides:

- a summary of the vulnerability,
- typical mechanisms of failure, and
- techniques that programmers can use to avoid the vulnerability.

Clause 8, *New Vulnerabilities*, provides new vulnerabilities that have not yet had corresponding programming language annex text developed.

Annex A, *Vulnerability Taxonomy and List*, is a categorization of the vulnerabilities of this report in the form of a hierarchical outline and a list of the vulnerabilities arranged in alphabetic order by their three letter code.

Annex B, *Language Specific Vulnerability Template*, is a template for the writing of programming language specific annexes that explain how the vulnerabilities from clause 6 are realized in that programming language (or show how they are absent), and how they might be mitigated in language-specific terms.

Additional annexes, each named for a particular programming language, list the vulnerabilities of Clauses 6 and 7 and describe how each vulnerability appears in the specific language and how it may be mitigated in that language, whenever possible. All of the language-dependent descriptions assume that the user adheres to the standard for the language as listed in the sub-clause of each annex.

5 Vulnerability issues

5.1 Predictable execution

There are many reasons why software might not execute as expected by its developers, its users or other stakeholders. Reasons include incorrect specifications, configuration management errors and a myriad of others. This Technical Report focuses on one cause—the usage of programming languages in ways that render the execution of the code less predictable.

Predictable execution is a property of a program such that all possible executions have results that can be predicted from examination of the source code. Achieving predictability is complicated by that fact that software may be used:

- on unanticipated platforms (for example, ported to a different processor)
- in unanticipated ways (as usage patterns change),
- in unanticipated contexts (for example, software reuse and system-of-system integrations), and
- by unanticipated users (for example, those seeking to exploit and penetrate a software system).

Furthermore, today's ubiquitous connectivity of software systems virtually guarantees that most software will be attacked—either because it is a target for penetration or because it offers a springboard for penetration of other software. Accordingly, today's programmers must take additional care to ensure predictable execution despite the new challenges.

Software vulnerabilities are unwanted characteristics of software that may allow software to execute in ways that are unexpected. Programmers introduce vulnerabilities into software by using language features that are inherently unpredictable in the variable circumstances outlined above or by using features in a manner that reduces what predictability they could offer. Of course, complete predictability is an ideal (particularly because new vulnerabilities are often discovered through experience), but any programmer can improve predictability by careful avoiding the introduction of known vulnerabilities into code.

This Technical Report focuses on a particular class of vulnerabilities, *language vulnerabilities*. These are properties of programming languages that can contribute to (or are strongly correlated with) *application vulnerabilities*—security weaknesses, safety hazards, or defects. An example may clarify the relationship. The programmer's use of a string copying function that does check length may be exploited by an attacker to place incorrect return values on the program stack, hence passing control of the execution to code provided by the attacker. The string copying function is the language vulnerability and the resulting weakness of the program in the face of the stack attack is the application vulnerability. The programming language vulnerability enables the application vulnerability. The language vulnerability can be avoided by using a string copying function that does set appropriate bounds on the length of the string to be copied. By using a bounded copy function the programmer improves the predictability of the code's execution.

The primary purpose of this Technical Report is to survey common programming language vulnerabilities; this is done in Clause 6. Each description explains how an application vulnerability can result. In Clause 7, a few additional application vulnerabilities are described. These are selected because they are associated with language weaknesses even if they do not directly result from language vulnerabilities. For example, a programmer might have stored a password in plaintext (see [XYM]) because the programming language did not provide a suitable library function for storing the password in a non-recoverable format.

In addition to considering the individual vulnerabilities, it is instructive to consider the sources of uncertainty that can decrease the predictability of software. These sources are briefly considered in the remainder of this clause.

5.2 Sources of unpredictability in language specification

5.2.1 Incomplete or evolving specification

The design and specification of a programming language involves considerations that are very different from the use of the language in programming. Language specifiers often need to maintain compatibility with older versions of the language—even to the extent of retaining inherently vulnerable features. Sometimes the semantics of new or complex features aren't completely known, especially when used in combination with other features.

5.2.2 Undefined behaviour

It's simply not possible for the specifier of a programming language to describe every possible behaviour. For example, the result of using a variable to which no value has been assigned is left undefined by most languages. In such cases, a program might do anything—including crashing with no diagnostic or executing with wrong data, leading to incorrect results.

5.2.3 Unspecified behaviour

The behaviour of some features may be incompletely defined. The language implementer would have to choose from finite set of choices, but the choice may not be apparent to the programmer. In such cases, different compilers may lead to different results.

5.2.4 Implementation-defined behaviour

In some cases, the results of execution may depend upon characteristics of the compiler that was used, the processor upon which the software is executed, or the other systems with which the software has interfaces. In principle, one could predict the execution with sufficient knowledge of the implementation, but such knowledge is sometimes difficult to obtain. Furthermore, dependence on a specific implementation-defined behaviour will lead to problems when a different processor or compiler is used—sometimes if different compiler switch settings are used.

5.2.5 Difficult features

Some language features may be difficult to understand or to use appropriately, either due to complicated semantics (for example, floating point in numerical analysis applications) or human limitations (for example, deeply nested program constructs or expressions). Sometimes simple typing errors can lead to major changes in behaviour without a diagnostic (for example, typing "=" for assignment when one really intended "==" for comparison).

5.2.6 Inadequate language support

No language is suitable for every possible application. Furthermore, programmers sometimes do not have the freedom to select the language that is most suitable for the task at hand. In many cases, libraries must be used to supplement the functionality of the language. Then, the library itself becomes a potential source of uncertainty reducing the predictability of execution.

5.3 Sources of unpredictability in language usage

5.3.1 Porting and interoperation

When a program is recompiled using a different compiler, recompiled using different switches, executed with different libraries, executed on a different platform, or even interfaced with different systems, its behaviour will change. Changes result from different choices for unspecified and implementation-defined behaviour, differences in library function, and differences in underlying hardware and operating system support. The

problem is far worse if the original programmer chose to use implementation-dependent extensions to the language rather than staying with the standardized language.

5.3.2 Compiler selection and usage

Nearly all software has bugs and compilers are no exception. They should be carefully selected from trusted sources and qualified prior to use. Perhaps less obvious, though, is the use of compiler switches. Different switch settings will result in differences in generated code. A careful selection of settings can improve the predictability of code, for example, a setting that causes the flagging of any usage of an implementation-defined extension.

6. Programming Language Vulnerabilities

6.1 General

This clause provides language-independent descriptions of vulnerabilities in programming languages that can lead to application vulnerabilities. Each description provides:

- a summary of the vulnerability,
- characteristics of languages where the vulnerability may be found,
- typical mechanisms of failure,
- techniques that programmers can use to avoid the vulnerability, and
- ways that language designers can modify language specifications in the future to help programmers mitigate the vulnerability.

Descriptions of how vulnerabilities are manifested in particular programming languages are provided in annexes of this Technical Report. In each case, the behaviour of the language is assumed to be as specified by the standard cited in the annex. Clearly, programs could have different vulnerabilities in a non-standard implementation. Examples of non-standard implementations include:

- compilers written to implement some specification other than the standard,
- use of non-standard vendor extensions to the language, and
- use of compiler switches providing alternative semantics.

6.2 Terminology

The following descriptions are written in a language-independent manner except when specific languages are used in examples. The annexes may be consulted for language specific descriptions.

This clause will, in general, use the terminology that is most natural to the description of each individual vulnerability. Hence terminology may differ from description to description.

6.3 Type System [IHN]

6.3.1 Description of application vulnerability

When data values are converted from one data type to another, even when done intentionally, unexpected results can occur.

6.3.2 Cross reference

JSF AV Rules: 148 and 183
MISRA C 2004: 6.1, 6.2, 6.3, 10.1, and 10.5
MISRA C++ 2008: 3-9-2, 5-0-3 to 5-0-14
CERT C guidelines: DCL07-C, DCL11-C, DCL35-C, EXP05-C and EXP32-C
Ada Quality and Style Guide: 3.4

6.3.3 Mechanism of failure

The *type* of a data object informs the compiler how values should be represented and which operations may be applied. The *type system* of a language is the set of rules used by the language to structure and organize its collection of types. Any attempt to manipulate data objects with inappropriate operations is a *type error*. A program is said to be *type safe* (or *type secure*) if it can be demonstrated that it has no type errors [27].

Every programming language has some sort of type system. A language is *statically typed* if the type of every expression is known at compile time. The type system is said to be *strong* if it guarantees type safety and *weak* if it does not. There are strongly typed languages that are not statically typed because they enforce type safety with run time checks [27].

In practical terms, nearly every language falls short of being strongly typed (in an ideal sense) because of the inclusion of mechanisms to bypass type safety in particular circumstances. For that reason and because every language has a different type system, this description will focus on taking advantage of whatever features for type safety may be available in the chosen language.

Sometimes it is appropriate for a data value to be converted from one type to another *compatible* one. For example, consider the following program fragment, written in no specific language:

```
float a;
integer i;
a := a + i;
```

The variable "`i`" is of integer type. It must be converted to the float type before it can be added to the data value. An implicit conversion, as shown, is called coercion. If, on the other hand, the conversion must be explicit, for example, "`a := a + float(i)`", then the conversion is called a *cast*.

Type *equivalence* is the strictest form of type compatibility; two types are equivalent if they are compatible without using coercion or casting. Type equivalence is usually characterized in terms of *name type equivalence*—two variables have the same type if they are declared in the same declaration or declarations that use the same type name—or *structure type equivalence*—two variables have the same type if they have identical structures.

There are variations of these approaches and most languages use different combinations of them [28]. Therefore, a programmer skilled in one language may very well code inadvertent type errors when using a different language.

It is desirable for a program to be type safe because the application of operations to operands of an inappropriate type may produce unexpected results. In addition, the presence of type errors can reduce the effectiveness of static analysis for other problems. Searching for type errors is a valuable exercise because their presence often reveals design errors as well as coding errors. Many languages check for type errors—some at compile-time, others at run-time. Obviously, compile-time checking is more valuable because it can catch errors that are not executed by a particular set of test cases.

Making the most use of the type system of a language is useful in two ways. First, data conversions always bear the risk of changing the value. For example, a conversion from integer to float risks the loss of significant digits while the inverse conversion risks the loss of any fractional value. Conversion of an integer value from a type with a longer representation to a type with a shorter representation risks the loss of significant digits. T his can produce particularly puzzling results if the value is used to index an array. Conversion of a floating-point value from a type with a longer representation to a type with a shorter representation risks the loss of precision. This can be particularly severe in computations where the number of calculations increases as a power of the problem size. (It should be noted that similar surprises can occur when an application is retargeted to a machine with different representations of numeric values.)

Second, a programmer can use the type system to increase the probability of catching design errors or coding blunders. For example, the following Ada fragment declares two distinct floating-point types:

```
type Celsius is new Float;
type Fahrenheit is new Float;
```

The declaration makes it impossible to add a value of type Celsius to a value of type Fahrenheit without explicit conversion.

6.3.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- Languages that support multiple types and allow conversions between types.

6.3.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Take advantage of any facility offered by the programming language to declare distinct types and use any mechanism provided by the language processor and related tools to check for or enforce type compatibility.
- Use available language and tools facilities to preclude or detect the occurrence of coercion. If it is not possible, use human review to assist in searching for coercions.
- Avoid casting data values except when there is no alternative. Document such occurrences so that the justification is made available to maintainers.

- Use the most restricted data type that suffices to accomplish the job. For example, use an enumeration type to select from a limited set of choices (such as, a switch statement or the discriminant of a union type) rather than a more general type, such as integer. This will make it possible for tooling to check if all possible choices have been covered.
- Treat every compiler, tool, or run-time diagnostic concerning type compatibility as a serious issue. Do not resolve the problem by modifying the code by inserting an explicit cast, without further analysis; instead examine the underlying design to determine if the type error is a symptom of a deeper problem.
- Never ignore instances of coercion; if the conversion is necessary, convert it to a cast and document the rationale for use by maintainers.
- Analyze the problem to be solved to learn the magnitudes and/or the precisions of the quantities needed as auxiliary variables, partial results and final results.

6.3.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specifiers should standardize on a common, uniform terminology to describe their type systems so that programmers experienced in other languages can reliably learn the type system of a language that is new to them.
- Provide a mechanism for selecting data types with sufficient capability for the problem at hand.
- Provide a way for the computation to determine the limits of the data types actually selected.
- Language implementers should consider providing compiler switches or other tools to provide the highest possible degree of checking for type errors.

6.4 Bit Representations [STR]

6.4.1 Description of application vulnerability

Interfacing with hardware, other systems and protocols often requires access to one or more bits in a single computer word, or access to bit fields that may cross computer words for the machine in question. Mistakes can be made as to what bits are to be accessed because of the “endianness” of the processor (see below) or because of miscalculations. Access to those specific bits may affect surrounding bits in ways that compromise their integrity. This can result in the wrong information being read from hardware, incorrect data or commands being given, or information being mangled, which can result in arbitrary effects on components attached to the system.

6.4.2 Cross reference

JSF AV Rules 147, 154 and 155
MISRA C 2004: 3.5, 6.4, 6.5, and 12.7
MISRA C++ 2008: 5-0-21, 5-2-4 to 5-2-9, and 9-5-1
CERT C guidelines: EXP38-C, INT00-C, INT07-C, INT12-C, INT13-C, and INT14-C
Ada Quality and Style Guide: 7.6.1 through 7.6.9, and 7.3.1

6.4.3 Mechanism of failure

Computer languages frequently provide a variety of sizes for integer variables. Languages may support short, integer, long, and even big integers. Interfacing with protocols, device drivers, embedded systems, low level graphics or other external constructs may require each bit or set of bits to have a particular meaning. Those bit sets may or may not coincide with the sizes supported by a particular language implementation. When they do not, it is common practice to pack all of the bits into one word. Masking and shifting of the word using powers of two to pick out individual bits or using sums of powers of 2 to pick out subsets of bits (for example, using $28=2^2+2^3+2^4$ to create the mask 11100 and then shifting 2 bits) provides a way of extracting those bits. Knowledge of the underlying bit storage is usually not necessary to accomplish simple extractions such as these. Problems can arise when programmers mix their techniques to reference the bits or output the bits. Problems can arise when programmers mix arithmetic and logical operations to reference the bits or output the bits. The storage ordering of the bits may not be what the programmer expects.

Packing of bits in an integer is not inherently problematic. However, an understanding of the intricacies of bit level programming must be known. Some computers or other devices store the bits left to right while others store them right to left. The type of storage can cause problems when interfacing with external devices that expect the bits in the opposite order. One problem arises when assumptions are made when interfacing with external constructs and the ordering of the bits or words are not the same as the receiving entity. Programmers may inadvertently use the sign bit in a bit field and then may not be aware that an arithmetic shift (sign extension) is being performed when right shifting causing the sign bit to be extended into other fields. Alternatively, a left shift can cause the sign bit to be one. Bit manipulations can also be problematic when the manipulations are done on binary encoded records that span multiple words. The storage and ordering of the bits must be considered when doing bitwise operations across multiple words as bytes may be stored in big-endian or little-endian format.

6.4.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that allow bit manipulations.

6.4.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Any assumption about bit ordering should be explicitly documented.
- The way bit ordering is done on the host system and on the systems with which the bit manipulations will be interfaced should be understood.
- Bit fields should be used in languages that support them.
- Bit operators should not be used on signed operands.
- Localize and document the code associated with explicit manipulation of bits and bit fields.

6.4.6 Implications for standardization

In future standardization activities, the following items should be considered:

- For languages that are commonly used for bit manipulations, an *API* (Application Programming Interface) for bit manipulations that is independent of word size and machine instruction set should be defined and standardized.

6.5 Floating-point Arithmetic [PLF]

6.5.1 Description of application vulnerability

Most real numbers cannot be represented exactly in a computer. To represent real numbers, most computers use IEC 60559 [47], or the US equivalent ANSI/IEEE Std 754 [35]. The bit representation for a floating-point number can vary from compiler to compiler and on different platforms. Relying on a particular representation can cause problems when a different compiler is used or the code is reused on another platform. Regardless of the representation, many real numbers can only be approximated since representing the real number using a binary representation would require an endlessly repeating string of bits or more binary digits than are available for representation. Therefore it should be assumed that a floating-point number is only an approximation, even though it may be an extremely good one. Floating-point representation of a real number or a conversion to floating-point can cause surprising results and unexpected consequences to those unaccustomed to the idiosyncrasies of floating-point arithmetic.

Algorithms that use floating point can have anomalous behaviour when used with certain values. The most common results are erroneous results or algorithms that never terminate for certain segments of the numeric domain, or for isolated values.

6.5.2 Cross reference

JSF AV Rules: 146, 147, 184, 197, and 202
MISRA C 2004: 1.5, 12.12, 13.3, and 13.4
MISRA C++ 2008: 0-4-3, 3-9-3, and 6-2-2
CERT C guidelines: FLP00-C, FP01-C, FLP02-C and FLP30-C
Ada Quality and Style Guide: 5.5.6 and 7.2.1 through 7.2.8

6.5.3 Mechanism of failure

Floating-point numbers are generally only an approximation of the actual value. In the base 10 world, the value of 1/3 is 0.333333… The same type of situation occurs in the binary world, but numbers that can be represented with a limited number of digits in base 10, such as 1/10=0.1 become endlessly repeating sequences in the binary world. So 1/10 represented as a binary number is:

0.0001100110011001100110011001100110011001100110011…

Which is 0*1/2 + 0*1/4 + 0*1/8 + 1*1/16 + 1*1/32 + 0*1/64… and no matter how many digits are used, the representation will still only be an approximation of 1/10. Therefore when adding 1/10 ten times, the final result may or may not be exactly 1.

Accumulating floating point values through the repeated addition of values, particularly relatively small values, can provide unexpected results. Using an accumulated value to terminate a loop can result in an unexpected number of iterations. Rounding and truncation can cause tests of floating-point numbers against other values to

yield unexpected results. Another cause of floating point errors is reliance upon comparisons of floating point values or the comparison of a floating point value with zero. Tests of equality/inequality can vary due to propagation or conversion errors. Differences in magnitudes of floating-point numbers can result in no change of a very large floating-point number when a relatively small number is added to or subtracted from it.

Manipulating bits in floating-point numbers is also very implementation dependent. Though IEC 60559 is a commonly used representation for floating-point data types, it is not universally used or required by all computer languages. Some languages predate IEC 60559 and make the support for the standard optional. One IEC 60559 representation uses a 24-bit mantissa (including the sign bit) and an 8-bit exponent, but the number of bits allocated to the mantissa and exponent can vary when using other representations, as can the particular representation used for the mantissa and exponent. Even within IEC 60559, various alternative representations are permitted for the "extended precision" format (from 80- to 128-bit representations, with or without a hidden bit). Typically special representations are specified for positive and negative zero and infinity. Relying on a particular bit representation is inherently problematic, especially when a new compiler is introduced or the code is reused on another platform. The uncertainties arising from floating-point can be divided into uncertainty about the actual bit representation of a given value (such as, big-endian or little-endian) and the uncertainty arising from the rounding of arithmetic operations (for example, the accumulation of errors when imprecise floating-point values are used as loop indices).

6.5.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- All languages with floating-point variables can be subject to rounding or truncation errors.

6.5.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Do not use a floating-point expression in a Boolean test for equality. Instead, use coding that determines the difference between the two values to determine whether the difference is acceptably small enough so that two values can be considered equal. Note that if the two values are very large, the "small enough" difference can be a very large number.
- Use library functions with known numerical characteristics whenever possible.
- Unless the use of floating-point is simple, an expert in numerical analysis should check the stability and accuracy of the algorithm employed.
- Avoid the use of a floating-point variable as a loop counter. If necessary to use a floating-point value as a loop control, use inequality to determine the loop control (that is, <, <=, > or >=).
- Understand the floating-point format used to represent the floating-point numbers. This will provide some understanding of the underlying idiosyncrasies of floating-point arithmetic.
- Manipulating the bit representation of a floating-point number should not be done except with built-in language operators and functions that are designed to extract the mantissa and exponent.
- Do not use floating-point for exact values such as monetary amounts. Use floating-point only when necessary such as for fundamentally inexact values such as measurements.
- Consider the use of decimal floating-point facilities when available.

6.5.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages that do not already adhere to or only adhere to a subset of IEC 60559 [47] should consider adhering completely to the standard. Examples of standardization that should be considered:
  - C should consider requiring IEC 60559 for floating-point arithmetic, rather than providing it as an option, as is the case in ISO/IEC 9899:2011[4].
  - Java should consider fully adhering to IEC 60559 instead of a subset.
- Languages should consider providing a means to generate diagnostics for code that attempts to test equality of two floating point values.
- Languages should consider standardizing their data type to ISO/IEC 10967-1:1994 and ISO/IEC 10967-2:2001.

6.6 Enumerator Issues [CCB]

6.6.1 Description of application vulnerability

Enumerations are a finite list of named entities that contain a fixed mapping from a set of names to a set of integral values (called the representation) and an order between the members of the set. In some languages there are no other operations available except order, equality, first, last, previous, and next; in others the full underlying representation operators are available, such as integer “+” and “-” and bit-wise operations.

Most languages that provide enumeration types also provide mechanisms to set non-default representations. If these mechanisms do not enforce whole-type operations and check for conflicts then some members of the set may not be properly specified or may have the wrong mappings. If the value-setting mechanisms are positional only, then there is a risk that improper counts or changes in relative order will result in an incorrect mapping.

For arrays indexed by enumerations with non-default representations, there is a risk of structures with holes, and if those indexes can be manipulated numerically, there is a risk of out-of-bound accesses of these arrays.

Most of these errors can be readily detected by static analysis tools with appropriate coding standards, restrictions and annotations. Similarly mismatches in enumeration value specification can be detected statically. Without such rules, errors in the use of enumeration types are computationally hard to detect statically as well as being difficult to detect by human review.

6.6.2 Cross reference

JSF AV Rule: 145
MISRA C 2004: 9.2 and 9.3
MISRA C++ 2008: 8-5-3
CERT C guidelines: INT09-C
Holzmann rule 6
Ada Quality and Style Guide: 3.4.2

6.6.3 Mechanism of failure

As a program is developed and maintained the list of items in an enumeration often changes in three basic ways: new elements are added to the list; order between the members of the set often changes; and representation (the map of values of the items) change. Expressions that depend on the full set or specific relationships between elements of the set can create value errors that could result in wrong results or in unbounded behaviours if used as array indices.

Improperly mapped representations can result in some enumeration values being unreachable, or may create "holes" in the representation where values that cannot be defined are propagated.

If arrays are indexed by enumerations containing non-default representations, some implementations may leave space for values that are unreachable using the enumeration, with a possibility of unnecessarily large memory allocations or a way to pass information undetected (hidden channel).

When enumerators are set and initialized explicitly and the language permits incomplete initializers, then changes to the order of enumerators or the addition or deletion of enumerators can result in the wrong values being assigned or default values being assigned improperly. Subsequent indexing can result in invalid accesses and possibly unbounded behaviours.

6.6.4 Applicable language Characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that permit incomplete mappings between enumerator specification and value assignment, or that provide a positional-only mapping require additional static analysis tools and annotations to help identify the complete mapping of every literal to its value.
- Languages that provide a trivial mapping to a type such as integer require additional static analysis tools to prevent mixed type errors. They also cannot prevent invalid values from being placed into variables of such enumerator types. For example:

```
enum Directions {back, forward, stop};
enum Directions a = forward, b = stop, c = a + b;
```

  In this example, `c` may have a value not defined by the enumeration, and any further use as that enumeration will lead to erroneous results.
- Some languages provide no enumeration capability, leaving it to the programmer to define named constants to represent the values and ranges.

6.6.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use static analysis tools that will detect inappropriate use of enumerators, such as using them as integers or bit maps, and that detect enumeration definition expressions that are incomplete or incorrect. For languages with a complete enumeration abstraction this is the compiler.

6.6.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages that currently permit arithmetic and logical operations on enumeration types could provide a mechanism to ban such operations program-wide.
- Languages that provide automatic defaults or that do not enforce static matching between enumerator definitions and initialization expressions could provide a mechanism to enforce such matching.

6.7 Numeric Conversion Errors [FLC]

6.7.1 Description of application vulnerability

Certain contexts in various languages may require exact matches with respect to types [32]:

```
aVar := anExpression
value1 + value2
foo(arg1, arg2, arg3, … , argN)
```

Type conversion seeks to follow these exact match rules while allowing programmers some flexibility in using values such as: structurally-equivalent types in a name-equivalent language, types whose value ranges may be distinct but intersect (for example, subranges), and distinct types with sensible/meaningful corresponding values (for example, integers and floats). Explicit conversions are called *type casts*. An implicit type conversion between compatible but not necessarily equivalent types is called *type coercion*.

Numeric conversions can lead to a loss of data, if the target representation is not capable of representing the original value. For example, converting from an integer type to a smaller integer type can result in truncation if the original value cannot be represented in the smaller size and converting a floating point to an integer can result in a loss of precision or an out-of-range value.

Type conversion errors can lead to erroneous data being generated, algorithms that fail to terminate, array bounds errors, and arbitrary program execution.

6.7.2 Cross reference

CWE:
192. Integer Coercion Error
MISRA C 2004: 10.1-10.6, 11.3-11.5, and 12.9
MISRA C++ 2008: 2-13-3, 5-0-3, 5-0-4, 5-0-5, 5-0-6, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-2-5, 5-2-9, and 5-3-2
CERT C guidelines: FLP34-C, INT02-C, INT08-C, INT31-C, and INT35-C

6.7.3 Mechanism of failure

Numeric conversion errors results in data integrity issues, but they may also result in a number of safety and security vulnerabilities.

Vulnerabilities typically occur when appropriate range checking is not performed, and unanticipated values are encountered. These can result in safety issues, for example, when the Ariane 5 launcher failure occurred due to an improperly handled conversion error resulting in the processor being shutdown [29].

Conversion errors can also result in security issues. An attacker may input a particular numeric value to exploit a flaw in the program logic. The resulting erroneous value may then be used as an array index, a loop iterator, a length, a size, state data, or in some other security critical manner. For example, a truncated integer value may be used to allocate memory, while the actual length is used to copy information to the newly allocated memory, resulting in a buffer overflow [30].

Numeric type conversion errors often lead to undefined states of execution resulting in infinite loops or crashes. In some cases, integer type conversion errors can lead to exploitable buffer overflow conditions, resulting in the execution of arbitrary code. Integer type conversion errors result in an incorrect value being stored for the variable in question.

6.7.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that perform implicit type conversion (coercion).
- Weakly typed languages that do not strictly enforce type rules.
- Languages that support logical, arithmetic, or circular shifts on integer values.
- Languages that do not generate exceptions on problematic conversions.

6.7.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. All integer values originating from a source that is not trusted should be validated for correctness. However, it is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program [30].
- An alternative or ancillary approach is to protect each operation. However, because of the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, this approach can be prohibitively labor intensive and expensive to implement.
- A language that generates exceptions on erroneous data conversions might be chosen. Design objects and program flow such that multiple or complex casts are unnecessary. Ensure that any data type casting that you must use is entirely understood to reduce the plausibility of error in use.
- The use of static analysis can often identify whether or not unacceptable numeric conversions will occur.

Verifiably in-range operations are often preferable to treating out of range values as an error condition because the handling of these errors has been repeatedly shown to cause denial-of-service problems in actual applications. Faced with a numeric conversion error, the underlying computer system may do one of two things: (a) signal some sort of error condition, or (b) produce a numeric value that is within the range of representable values on that system. The latter semantics may be preferable in some situations in that it allows the computation

to proceed, thus avoiding a denial-of-service attack. However, it raises the question of what numeric result to return to the user.

A recent innovation from ISO/IEC TR 24731-1 [13] is the definition of the `rsize_t` type for the C programming language. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. Also, some implementations do not support objects as large as the maximum value that can be represented by type `size_t`. For these reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define `RSIZE_MAX` as `SIZE_MAX`, which means that there is no object size that is considered a runtime-constraint violation.

6.7.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing means similar to the ISO/IEC TR 24731-1 definition of `rsize_t` type for C to restrict object sizes so as to expose programming errors.
- Languages should consider making all type conversions explicit or at least generating warnings for implicit conversions where loss of data might occur.

6.8 String Termination [CJM]

6.8.1 Description of application vulnerability

Some programming languages use a termination character to indicate the end of a string. Relying on the occurrence of the string termination character without verification can lead to either exploitation or unexpected behaviour.

6.8.2 Cross reference

CWE:

170. Improper Null Termination

CERT C guidelines: STR03-C, STR31-C, STR32-C, and STR36-C

6.8.3 Mechanism of failure

String termination errors occur when the termination character is solely relied upon to stop processing on the string and the termination character is not present. Continued processing on the string can cause an error or potentially be exploited as a buffer overflow. This may occur as a result of a programmer making an assumption that a string that is passed as input or generated by a library contains a string termination character when it does not.

Programmers may forget to allocate space for the string termination character and expect to be able to store an `n` length character string in an array that is `n` characters long. Doing so may work in some instances depending on what is stored after the array in memory, but it may fail or be exploited at some point.

6.8.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that use a termination character to indicate the end of a string.
- Languages that do not do bounds checking when accessing a string or array.

6.8.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Do not rely solely on the string termination character.
- Use library calls that do not rely on string termination characters such as `strncpy` instead of `strcpy` in the standard C library.

6.8.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Eliminating library calls that make assumptions about string termination characters.
- Checking bounds when an array or string is accessed.
- Specifying a string construct that does not need a string termination character.

6.9 Buffer Boundary Violation (Buffer Overflow) [HCB]

6.9.1 Description of application vulnerability

A buffer boundary violation arises when, due to unchecked array indexing or unchecked array copying, storage outside the buffer is accessed. Usually boundary violations describe the situation where such storage is then written. Depending on where the buffer is located, logically unrelated portions of the stack or the heap could be modified maliciously or unintentionally. Usually, buffer boundary violations are accesses to contiguous memory beyond either end of the buffer data, accessing before the beginning or beyond the end of the buffer data is equally possible, dangerous and maliciously exploitable.

6.9.2 Cross reference

CWE:

120. Buffer copy without Checking Size of Input (‘Classic Buffer Overflow’)
122. Heap-based Buffer Overflow
124. Boundary Beginning Violation (‘Buffer Underwrite’)
129. Unchecked Array Indexing
131. Incorrect Calculation of Buffer Size
787. Out-of-bounds Write
805. Buffer Access with Incorrect Length Value

JSF AV Rule: 15 and 25
MISRA C 2004: 21.1
MISRA C++ 2008: 5-0-15 to 5-0-18
CERT C guidelines: ARR30-C, ARR32-C, ARR33-C, ARR38-C, MEM35-C and STR31-C

6.9.3 Mechanism of failure

The program statements that cause buffer boundary violations are often difficult to find.

There are several kinds of failures (in all cases an exception may be raised if the accessed location is outside of some permitted range of the run-time environment):

- A read access will return a value that has no relationship to the intended value, such as, the value of another variable or uninitialized storage.
- An out-of-bounds read access may be used to obtain information that is intended to be confidential.
- A write access will not result in the intended value being updated and may result in the value of an unrelated object (that happens to exist at the given storage location) being modified, including the possibility of changes in external devices resulting from the memory location being hardware-mapped.
- When an array has been allocated storage on the stack an out-of-bounds write access may modify internal runtime housekeeping information (for example, a function's return address) which might change a program's control flow.
- An inadvertent or malicious overwrite of function pointers that may be in memory, causing them to point to an unexpected location or the attacker's code. Even in applications that do not explicitly use function pointers, the run-time will usually store pointers to functions in memory. For example, object methods in object-oriented languages are generally implemented using function pointers in a data structure or structures that are kept in memory. The consequence of a buffer boundary violation can be targeted to cause arbitrary code execution; this vulnerability may be used to subvert any security service.

6.9.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that do not detect and prevent an array being accessed outside of its declared bounds (either by means of an index or by pointer[1]).
- Languages that do not automatically allocate storage when accessing an array element for which storage has not already been allocated.
- Languages that provide bounds checking but permit the check to be suppressed.
- Languages that allow a copy or move operation without an automatic length check ensuring that source and target locations are of at least the same size. The destination target can be larger than the source being copied.

6.9.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use of implementation-provided functionality to automatically check array element accesses and prevent out-of-bounds accesses.

[1] Using the physical memory address to access the memory location.

- Use of static analysis to verify that all array accesses are within the permitted bounds. Such analysis may require that source code contain certain kinds of information, such as, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified.
- Sanity checks should be performed on all calculated expressions used as an array index or for pointer arithmetic.

Some guideline documents recommend only using variables having an unsigned data type when indexing an array, on the basis that an unsigned data type can never be negative. This recommendation simply converts an indexing underflow to an indexing overflow because the value of the variable will wrap to a large positive value rather than a negative one. Also some languages support arrays whose lower bound is greater than zero, so an index can be positive and be less than the lower bound.

In the past the implementation of array bound checking has sometimes incurred what has been considered to be a high runtime overhead (often because unnecessary checks were performed). It is now practical for translators to perform sophisticated analysis that significantly reduces the runtime overhead (because runtime checks are only made when it cannot be shown statically that no bound violations can occur).

6.9.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should provide safe copying of arrays as built-in operation.
- Languages should consider only providing array copy routines in libraries that perform checks on the parameters to ensure that no buffer overrun can occur.
- Languages should perform automatic bounds checking on accesses to array elements, unless the compiler can statically determine that the check is unnecessary. This capability may need to be optional for performance reasons.
- Languages that use pointer types should consider specifying a standardized feature for a pointer type that would enable array bounds checking.

6.10 Unchecked Array Indexing [XYZ]

6.10.1 Description of application vulnerability

Unchecked array indexing occurs when a value is used as an index into an array without checking that it falls within the acceptable index range.

6.10.2 Cross reference

CWE:
129. Unchecked Array Indexing
JSF AV Rules: 164 and 15
MISRA C 2004: 21.1
MISRA C++ 2008: 5-0-15 to 5-0-18
CERT C guidelines: ARR30-C, ARR32-C, ARR33-C, and ARR38-C
Ada Quality and Style Guide: 5.5.1, 5.5.2, 7.6.7, and 7.6.8

6.10.3 Mechanism of failure

A single fault could allow both an overflow and underflow of the array index. An index overflow exploit might use buffer overflow techniques, but this can often be exploited without having to provide "large inputs." Array index overflows can also trigger out-of-bounds read operations, or operations on the wrong objects; that is, "buffer overflows" are not always the result. Unchecked array indexing, depending on its instantiation, can be responsible for any number of related issues. Most prominent of these possible flaws is the buffer overflow condition, with consequences ranging from denial of service, and data corruption, to arbitrary code execution. The most common situation leading to unchecked array indexing is the use of loop index variables as buffer indexes. If the end condition for the loop is subject to a flaw, the index can grow or shrink unbounded, therefore causing a buffer overflow or underflow. Another common situation leading to this condition is the use of a function's return value, or the resulting value of a calculation directly as an index in to a buffer. Unchecked array indexing can result in the corruption of relevant memory and perhaps instructions, lead to the program halting, if the values are outside of the valid memory area. If the memory corrupted is data, rather than instructions, the system might continue to function with improper values. If the corrupted memory can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow.

Language implementations might or might not statically detect out of bound access and generate a compile-time diagnostic. At runtime the implementation might or might not detect the out-of-bounds access and provide a notification. The notification might be treatable by the program or it might not be. Accesses might violate the bounds of the entire array or violate the bounds of a particular index. It is possible that the former is checked and detected by the implementation while the latter is not. The information needed to detect the violation might or might not be available depending on the context of use. (For example, passing an array to a subroutine via a pointer might deprive the subroutine of information regarding the size of the array.)

Aside from bounds checking, some languages have ways of protecting against out-of-bounds accesses. Some languages automatically extend the bounds of an array to accommodate accesses that might otherwise have been beyond the bounds. However, this may or may not match the programmer's intent and can mask errors. Some languages provide for whole array operations that may obviate the need to access individual elements thus preventing unchecked array accesses.

6.10.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that do not automatically bounds check array accesses.
- Languages that do not automatically extend the bounds of an array to accommodate array accesses.

6.10.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Include sanity checks to ensure the validity of any values used as index variables.
- The choice could be made to use a language that is not susceptible to these issues.
- When available, use whole array operations whenever possible.

6.10.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing compiler switches or other tools to check the size and bounds of arrays and their extents that are statically determinable.
- Languages should consider providing whole array operations that may obviate the need to access individual elements.
- Languages should consider the capability to generate exceptions or automatically extend the bounds of an array to accommodate accesses that might otherwise have been beyond the bounds.

6.11 Unchecked Array Copying [XYW]

6.11.1 Description of application vulnerability

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.

6.11.2 Cross reference

CWE:
121. Stack-based Buffer Overflow
JSF AV Rule: 15
MISRA C 2004: 21.1
MISRA C++ 2008: 5-0-15 to 5-0-18
CERT C guidelines: ARR33-C and STR31-C
Ada Quality and Style Guide: 7.6.7 and 7.6.8

6.11.3 Mechanism of failure

Many languages and some third party libraries provide functions that efficiently copy the contents of one area of storage to another area of storage. Most of these libraries do not perform any checks to ensure that the copied from/to storage area is large enough to accommodate the amount of data being copied.

The arguments to these library functions include the addresses of the contents of the two storage areas and the number of bytes (or some other measure) to copy. Passing the appropriate combination of incorrect start addresses or number of bytes to copy makes it possible to read or write outside of the storage allocated to the source/destination area. When passed incorrect parameters the library function performs one or more unchecked array index accesses, as described in Unchecked Array Indexing [XYZ].

6.11.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that contain standard library functions for performing bulk copying of storage areas.
- The same range of languages having the characteristics listed in Unchecked Array Indexing [XYZ].

6.11.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Only use library functions that perform checks on the arguments to ensure no buffer overrun can occur (perhaps by writing a wrapper for the Standard provided functions). Perform checks on the argument expressions prior to calling the Standard library function to ensure that no buffer overrun will occur.
- Use static analysis to verify that the appropriate library functions are only called with arguments that do not result in a buffer overrun. Such analysis may require that source code contain certain kinds of information, for example, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified as annotations or language constructs.

6.11.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider only providing libraries that perform checks on the parameters to ensure that no buffer overrun can occur.
- Languages should consider providing full array assignment.

6.12 Pointer Casting and Pointer Type Changes [HFC]

6.12.1 Description of application vulnerability

The code produced for access via a data or function pointer requires that the type of the pointer is appropriate for the data or function being accessed. Otherwise undefined behaviour can occur. Specifically, “access via a data pointer” is defined to be “fetch or store indirectly through that pointer” and “access via a function pointer” is defined to be “invocation indirectly through that pointer.” The detailed requirements for what is meant by the “appropriate” type may vary among languages.

Even if the type of the pointer is appropriate for the access, erroneous pointer operations can still cause a fault.

6.12.2 Cross reference

CWE:
136. Type Errors
188. Reliance on Data/Memory Layout
JSF AV Rules: 182 and 183
MISRA C 2004: 11.1, 11.2, 11.3, 11.4, and 11.5
MISRA C++ 2008: 5-2-2 to 5-2-9
CERT C guidelines: INT11-C and EXP36-A
Hatton 13: Pointer casts
Ada Quality and Style Guide: 7.6.7 and 7.6.8

6.12.3 Mechanism of failure

If a pointer's type is not appropriate for the data or function being accessed, data can be corrupted or privacy can be broken by inappropriate read or write operation using the indirection provided by the pointer value. With a suitable type definition, large portions of memory can be maliciously or accidentally modified or read. Such modification of data objects will generally lead to value faults of the application. Modification of code elements such as function pointers or internal data structures for the support of object-orientation can affect control flow. This can make the code susceptible to targeted attacks by causing invocation via a pointer-to-function that has been manipulated to point to an attacker's payload.

6.12.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Pointers (and/or references) can be converted to different pointer types.
- Pointers to functions can be converted to pointers to data.

6.12.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Treat the compiler's pointer-conversion warnings as serious errors.
- Adopt programming guidelines (preferably augmented by static analysis) that restrict pointer conversions. For example, consider the rules itemized above from JSF AV [15], CERT C [11], Hatton [18], or MISRA C [12].
- Other means of assurance might include proofs of correctness, analysis with tools, verification techniques, or other methods.

6.12.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider creating a mode that provides a runtime check of the validity of all accessed objects before the object is read, written or executed.

6.13 Pointer Arithmetic [RVG]

6.13.1 Description of application vulnerability

Using pointer arithmetic incorrectly can result in addressing arbitrary locations, which in turn can cause a program to behave in unexpected ways.

6.13.2 Cross reference

JSF AV Rule: 215
MISRA C 2004: 17.1, 17.2, 17.3, and 17.4
MISRA C++ 2008: 5-0-15 to 5-0-18
CERT C guidelines: EXP08-C

6.13.3 Mechanism of failure

Pointer arithmetic used incorrectly can produce:

- Addressing arbitrary memory locations, including buffer underflow and overflow.
- Arbitrary code execution.
- Addressing memory outside the range of the program.

6.13.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that allow pointer arithmetic.

6.13.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Avoid using pointer arithmetic for accessing anything except composite types.
- Prefer indexing for accessing array elements rather than using pointer arithmetic.
- Limit pointer arithmetic calculations to the addition and subtraction of integers.

6.13.6 Implications for standardization

[None]

6.14 Null Pointer Dereference [XYH]

6.14.1 Description of application vulnerability

A null-pointer dereference takes place when a pointer with a value of `NULL` is used as though it pointed to a valid memory location. This is a special case of accessing storage via an invalid pointer.

6.14.2 Cross reference

CWE:
476. NULL Pointer Dereference
JSF AV Rule 174
CERT C guidelines: EXP34-C
Ada Quality and Style Guide: 5.4.5

6.14.3 Mechanism of failure

When a pointer with a value of `NULL` is used as though it pointed to a valid memory location, then a null-pointer dereference is said to take place. This can result in a segmentation fault, unhandled exception, or accessing unanticipated memory locations.

6.14.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that permit the use of pointers and that do not check the validity of the location being accessed prior to the access.
- Languages that allow the use of a `NULL` pointer.

6.14.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Before dereferencing a pointer, ensure it is not equal to `NULL`.

6.14.6 Implications for standardization

In future standardization activities, the following items should be considered:

- A language feature that would check a pointer value for `NULL` before performing an access should be considered.

6.15 Dangling Reference to Heap [XYK]

6.15.1 Description of application vulnerability

A dangling reference is a reference to an object whose lifetime has ended due to explicit deallocation or the stack frame in which the object resided has been freed due to exiting the dynamic scope. The memory for the object may be reused; therefore, any access through the dangling reference may affect an apparently arbitrary location of memory, corrupting data or code.

This description concerns the former case, dangling references to the heap. The description of dangling references to stack frames is [DCM]. In many languages references are called pointers; the issues are identical.

A notable special case of using a dangling reference is calling a deallocator, for example, `free()`, twice on the same pointer value. Such a "Double Free" may corrupt internal data structures of the heap administration, leading to faulty application behaviour (such as infinite loops within the allocator, returning the same memory repeatedly as the result of distinct subsequent allocations, or deallocating memory legitimately allocated to another request since the first `free()` call, to name but a few), or it may have no adverse effects at all.

Memory corruption through the use of a dangling reference is among the most difficult of errors to locate.

With sufficient knowledge about the heap management scheme (often provided by the *OS* (Operating System) or run-time system), use of dangling references is an exploitable vulnerability, since the dangling reference provides a method with which to read and modify valid data in the designated memory locations after freed memory has been re-allocated by subsequent allocations.

6.15.2 Cross reference

CWE:

415. Double Free (Note that Double Free (415) is a special case of Use After Free (416))

416. Use After Free

MISRA C 2004: 17.1-6

MISRA C++ 2008: 0-3-1, 7-5-1, 7-5-2, 7-5-3, and 18-4-1

CERT C guidelines: MEM01-C, MEM30-C, and MEM31.C

Ada Quality and Style Guide: 5.4.5, 7.3.3, and 7.6.6

6.15.3 Mechanism of failure

The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behaviour is undefined. Explicit deallocation of heap-allocated storage ends the lifetime of the object residing at this memory location (as does leaving the dynamic scope of a declared variable). The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime. Such pointers are called dangling references.

The use of dangling references to previously freed memory can have any number of adverse consequences — ranging from the corruption of valid data to the execution of arbitrary code, depending on the instantiation and timing of the deallocation causing all remaining copies of the reference to become dangling, of the system's reuse of the freed memory, and of the subsequent usage of a dangling reference.

Like memory leaks and errors due to double de-allocation, the use of dangling references has two common and sometimes overlapping causes:

- An error condition or other exceptional circumstances.
- Developer confusion over which part of the program is responsible for freeing the memory.

If a pointer to previously freed memory is used, it is possible that the referenced memory has been reallocated. Therefore, assignment using the original pointer has the effect of changing the value of an unrelated variable. This induces unexpected behaviour in the affected program. If the newly allocated data happens to hold a class description, in an object-oriented language for example, various function pointers may be scattered within the heap data. If one of these function pointers is overwritten with an address of malicious code, execution of arbitrary code can be achieved.

6.15.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that permit the use of pointers and that permit explicit deallocation by the developer or provide for alternative means to reallocate memory still pointed to by some pointer value.
- Languages that permit definitions of constructs that can be parameterized without enforcing the consistency of the use of parameter at compile time.

6.15.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use an implementation that checks whether a pointer is used that designates a memory location that has already been freed.
- Use a coding style that does not permit deallocation.
- In complicated error conditions, be sure that clean-up routines respect the state of allocation properly. If the language is object-oriented, ensure that object destructors delete each chunk of memory only once. Ensuring that all pointers are set to `NULL` once the memory they point to have been freed can be an effective strategy. The utilization of multiple or complex data structures may lower the usefulness of this strategy.
- Use a static analysis tool that is capable of detecting some situations when a pointer is used after the storage it refers to is no longer a pointer to valid memory location.
- Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or dereferencing `NULL` pointers or pointers that are not initialized. To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

6.15.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Implementations of the free function could tolerate multiple frees on the same reference/pointer or frees of memory that was never allocated.
- Language specifiers should design generics in such a way that any attempt to instantiate a generic with constructs that do not provide the required capabilities results in a compile-time error.
- For properties that cannot be checked at compile time, language specifiers should provide an assertion mechanism for checking properties at run-time. It should be possible to inhibit assertion checking if efficiency is a concern.
- A storage allocation interface should be provided that will allow the called function to set the pointer used to NULL after the referenced storage is deallocated.

6.16 Arithmetic Wrap-around Error [FIF]

6.16.1 Description of application vulnerability

Wrap-around errors can occur whenever a value is incremented past the maximum or decremented past the minimum value representable in its type and, depending upon

- whether the type is signed or unsigned,
- the specification of the language semantics and/or
- implementation choices,

"wraps around" to an unexpected value. This vulnerability is related to Using Shift Operations for Multiplication and Division [PIK][2].

6.16.2 Cross reference

CWE:

128. Wrap-around Error
190. Integer Overflow or Wraparound

JSF AV Rules: 164 and 15
MISRA C 2004: 10.1 to 10.6, 12.8 and 12.11
MISRA C++ 2008: 2-13-3, 5-0-3 to 5-0-10, and 5-19-1
CERT C guidelines: INT30-C, INT32-C, and INT34-C

6.16.3 Mechanism of failure

Due to how arithmetic is performed by computers, if a variable's value is increased past the maximum value representable in its type, the system may fail to provide an overflow indication to the program. One of the most common processor behaviour is to "wrap" to a very large negative value, or set a condition flag for overflow or underflow, or saturate at the largest representable value.

Wrap-around often generates an unexpected negative value; this unexpected value may cause a loop to continue for a long time (because the termination condition requires a value greater than some positive value) or an array bounds violation. A wrap-around can sometimes trigger buffer overflows that can be used to execute arbitrary code.

It should be noted that the precise consequences of wrap-around differ depending on:

- Whether the type is signed or unsigned.
- Whether the type is a modulus type.
- Whether the type's range is violated by exceeding the maximum representable value or falling short of the minimum representable value.
- The semantics of the language specification.
- Implementation decisions.

However, in all cases, the resulting problem is that the value yielded by the computation may be unexpected.

6.16.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that do not trigger an exception condition when a wrap-around error occurs.

6.16.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

[2] This description is derived from Wrap-Around Error [XYY], which appeared in Edition 1 of this international technical report.

- Determine applicable upper and lower bounds for the range of all variables and use language mechanisms or static analysis to determine that values are confined to the proper range.
- Analyze the software using static analysis looking for unexpected consequences of arithmetic operations.

6.16.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language standards developers should consider providing facilities to specify either an error, a saturated value, or a modulo result when numeric overflow occurs. Ideally, the selection among these alternatives could be made by the programmer.

6.17 Using Shift Operations for Multiplication and Division [PIK]

6.17.1 Description of application vulnerability

Using shift operations as a surrogate for multiply or divide may produce an unexpected value when the sign bit is changed or when value bits are lost. This vulnerability is related to Arithmetic Wrap-around Error [FIF][3].

6.17.2 Cross reference

CWE:

128. Wrap-around Error

190. Integer Overflow or Wraparound

JSF AV Rules: 164 and 15
MISRA C 2004: 10.1 to 10.6, 12.8 and 12.11
MISRA C++ 2008: 2-13-3, 5-0-3 to 5-0-10, and 5-19-1
CERT C guidelines: INT30-C, INT32-C, and INT34-C

6.17.3 Mechanism of failure

Shift operations intended to produce results equivalent to multiplication or division fail to produce correct results if the shift operation affects the sign bit or shifts significant bits from the value.

Such errors often generate an unexpected negative value; this unexpected value may cause a loop to continue for a long time (because the termination condition requires a value greater than some positive value) or an array bounds violation. The error can sometimes trigger buffer overflows that can be used to execute arbitrary code.

6.17.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that permit logical shift operations on variables of arithmetic type.

[3] This description is derived from Wrap-Around Error [XYY], which appeared in Edition 1 of this international technical report.

6.17.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Determine applicable upper and lower bounds for the range of all variables and use language mechanisms or static analysis to determine that values are confined to the proper range.
- Analyze the software using static analysis looking for unexpected consequences of shift operations.
- Avoid using shift operations as a surrogate for multiplication and division. Most compilers will use the correct operation in the appropriate fashion when it is applicable.

6.17.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Not providing logical shifting on arithmetic values or flagging it for reviewers.

6.18 Sign Extension Error [XZI]

6.18.1 Description of application vulnerability

Extending a signed variable that holds a negative value may produce an incorrect result.

6.18.2 Cross reference

CWE:
194. Incorrect Sign Extension
MISRA C++ 2008: 5-0-4
CERT C guidelines: INT13-C

6.18.3 Mechanism of failure

Converting a signed data type to a larger data type or pointer can cause unexpected behaviour due to the extension of the sign bit. A negative data element that is extended with an unsigned extension algorithm will produce an incorrect result. For instance, this can occur when a signed character is converted to a type short or a signed integer (32-bit) is converted to an integer type long (64-bit). Sign extension errors can lead to buffer overflows and other memory based problems. This can occur unexpectedly when moving software designed and tested on a 32-bit architecture to a 64-bit architecture computer.

6.18.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that are weakly typed due to their lack of enforcement of type classifications and interactions.
- Languages that explicitly or implicitly allow applying unsigned extension operations to signed entities or vice-versa.

6.18.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use a sign extension library, standard function, or appropriate language-specific coding methods to extend signed values.
- Use static analysis tools to help locate situations in which the conversion of variables might have unintended consequences.

6.18.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language definitions should define implicit and explicit conversions in a way that prevents alteration of the mathematical value beyond traditional rounding rules.

6.19 Choice of Clear Names [NAI]

6.19.1 Description of application vulnerability

Humans sometimes choose similar or identical names for objects, types, aggregates of types, subprograms and modules. They tend to use characteristics that are specific to the native language of the software developer to aid in this effort, such as use of mixed-casing, underscores and periods, or use of plural and singular forms to support the separation of items with similar names. Similarly, development conventions sometimes use casing for differentiation (for example, all uppercase for constants).

Human cognitive problems occur when different (but similar) objects, subprograms, types, or constants differ in name so little that human reviewers are unlikely to distinguish between them, or when the system maps such entities to a single entity.

Conventions such as the use of capitalization, and singular/plural distinctions may work in small and medium projects, but there are a number of significant issues to be considered:

- Large projects often have mixed languages and such conventions are often language-specific.
- Many implementations support identifiers that contain international character sets and some language character sets have different notions of casing and plurality.
- Different word-forms tend to be language and dialect specific, such as a pidgin, and may be meaningless to humans that speak other dialects.

An important general issue is the choice of names that differ from each other negligibly (in human terms), for example by differing by only underscores, (none, "_" "__"), plurals ("s"), visually similar characters (such as "l" and "1", "O" and "0"), or underscores/dashes ("-","_"). [There is also an issue where identifiers appear distinct to a human but identical to the computer, such as FOO, Foo, and foo in some computer languages.] Character sets extended with diacritical marks and non-Latin characters may offer additional problems. Some languages or their implementations may pay attention to only the first n characters of an identifier.

The problems described above are different from overloading or overriding where the same name is used intentionally (and documented) to access closely linked sets of subprograms. This is also different than using

reserved names which can lead to a conflict with the reserved use and the use of which may or may not be detected at compile time.

Name confusion can lead to the application executing different code or accessing different objects than the writer intended, or than the reviewers understood. This can lead to outright errors, or leave in place code that may execute sometime in the future with unacceptable consequences.

Although most such mistakes are unintentional, it is plausible that such usages can be intentional, if masking surreptitious behaviour is a goal.

6.19.2 Cross reference

JSF AV Rules: 48-56
MISRA C 2004: 1.4
CERT C guidelines: DCL02-C
Ada Quality and Style Guide: 3.2

6.19.3 Mechanism of Failure

Calls to the wrong subprogram or references to the wrong data element (that was missed by human review) can result in unintended behaviour. Language processors will not make a mistake in name translation, but human cognition limitations may cause humans to misunderstand, and therefore may be missed in human reviews.

6.19.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages with relatively flat name spaces will be more susceptible. Systems with modules, classes, packages can use qualification to disambiguate names that originate from different parents.
- Languages that provide preconditions, post conditions, invariances and assertions or redundant coding of subprogram signatures help to ensure that the subprograms in the module will behave as expected, but do nothing if different subprograms are called.
- Languages that treat letter case as significant. Some languages do not differentiate between names with differing case, while others do.

6.19.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Implementers can create coding standards that provide meaningful guidance on name selection and use. Good language specific guidelines could eliminate most problems.
- Use static analysis tools to show the target of calls and accesses and to produce alphabetical lists of names. Human review can then often spot the names that are sorted at an unexpected location or which look almost identical to an adjacent name in the list.
- Use static tools (often the compiler) to detect declarations that are unused.
- Use languages with a requirement to declare names before use or use available tool or compiler options to enforce such a requirement.

6.19.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages that do not require declarations of names should consider providing an option that does impose that requirement.

6.20 Dead Store [WXQ]

6.20.1 Description of application vulnerability

A variable's value is assigned but never subsequently used, either because the variable is not referenced again, or because a second value is assigned before the first is used. This may suggest that the design has been incompletely or inaccurately implemented, for example, a value has been created and then 'forgotten about'.

This vulnerability is very similar to Unused Variable [YZS].

6.20.2 Cross reference

CWE:
563. Unused Variable
MISRA C++ 2008: 0-1-4 and 0-1-6
CERT C guidelines: MSC13-C
See also Unused Variable [YZS]

6.20.3 Mechanism of failure

A variable is assigned a value but this is never subsequently used. Such an assignment is then generally referred to as a dead store.

A dead store may be indicative of careless programming or of a design or coding error; as either the use of the value was forgotten (almost certainly an error) or the assignment was performed even though it was not needed (at best inefficient). Dead stores may also arise as the result of mistyping the name of a variable, if the mistyped name matches the name of a variable in an enclosing scope.

There are legitimate uses for apparent dead stores. For example, the value of the variable might be intended to be read by another execution thread or an external device. In such cases, though, the variable should be marked as volatile. Common compiler optimization techniques will remove apparent dead stores if the variables are not marked as volatile, hence causing incorrect execution.

A dead store is justifiable if, for example:

- The code has been automatically generated, where it is commonplace to find dead stores introduced to keep the generation process simple and uniform.
- The code is initializing a sparse data set, where all members are cleared, and then selected values assigned a value.

6.20.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Any programming language that provides assignment.

6.20.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use static analysis to identify any dead stores in the program, and ensure that there is a justification for them.
- If variables are intended to be accessed by other execution threads or external devices, mark them as volatile.
- Avoid declaring variables of compatible types in nested scopes with similar names.

6.20.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing optional warning messages for dead store.

6.21 Unused Variable [YZS]

6.21.1 Description of application vulnerability

An unused variable is one that is declared but neither read nor written in the program. This type of error suggests that the design has been incompletely or inaccurately implemented.

Unused variables by themselves are innocuous, but they may provide memory space that attackers could use in combination with other techniques.

This vulnerability is similar to Dead Store [WXQ] if the variable is initialized but never used.

6.21.2 Cross reference

CWE:
563. Unused Variable
MISRA C++ 2008: 0-1-3
CERT C guidelines: MSC13-C
See also Dead Store [WXQ]

6.21.3 Mechanism of failure

A variable is declared, but never used. The existence of an unused variable may indicate a design or coding error.

Because compilers routinely diagnose unused local variables, their presence may be an indication that compiler warnings are either suppressed or are being ignored.

While unused variables are innocuous, they may provide available memory space to be used by attackers to exploit other vulnerabilities.

6.21.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that provide variable declarations.

6.21.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Enable detection of unused variables in the compiler.

6.21.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider requiring mandatory diagnostics for unused variables.

6.22 Identifier Name Reuse [YOW]

6.22.1 Description of application vulnerability

When distinct entities are defined in nested scopes using the same name it is possible that program logic will operate on an entity other than the one intended.

When it is not clear which identifier is used, the program could behave in ways that were not predicted by reading the source code. This can be found by testing, but circumstances can arise (such as the values of the same-named objects being mostly the same) where harmful consequences occur. This weakness can also lead to vulnerabilities such as hidden channels where humans believe that important objects are being rewritten or overwritten when in fact other objects are being manipulated.

For example, the innermost definition is deleted from the source, the program will continue to compile without a diagnostic being issued (but execution can produce unexpected results).

6.22.2 Cross reference

JSF AV Rules: 120 and 135-9
MISRA C 2004: 5.2, 5.5, 5.6, 5.7, 20.1, 20.2
MISRA C++ 2008: 2-10-2, 2-10-3, 2-10-4, 2-10-5, 2-10-6, 17-0-1, 17-0-2, and 17-0-3
CERT C guidelines: DCL01-C and DCL32-C
Ada Quality and Style Guide: 5.6.1 and 5.7.1

6.22.3 Mechanism of failure

Many languages support the concept of scope. One of the ideas behind the concept of scope is to provide a mechanism for the independent definition of identifiers that may share the same name.

For instance, in the following code fragment:

```
int some_var;
{
   int t_var;
   int some_var; /* definition in nested scope */

   t_var = 3;
   some_var = 2;
}
```

an identifier called `some_var` has been defined in different scopes.

If either the definition of `some_var` or `t_var` that occurs in the nested scope is deleted (for example, when the source is modified) it is necessary to delete all other references to the identifier's scope. If a developer deletes the definition of `t_var` but fails to delete the statement that references it, then most languages require a diagnostic to be issued (such as reference to undefined variable). However, if the nested definition of `some_var` is deleted but the reference to it in the nested scope is not deleted, then no diagnostic will be issued (because the reference resolves to the definition in the outer scope).

In some cases non-unique identifiers in the same scope can also be introduced through the use of identifiers whose common substring exceeds the length of characters the implementation considers to be distinct. For example, in the following code fragment:

```
extern int global_symbol_definition_lookup_table_a[100];
extern int global_symbol_definition_lookup_table_b[100];
```

the external identifiers are not unique on implementations where only the first 31 characters are significant. This situation only occurs in languages that allow multiple declarations of the same identifier (other languages require a diagnostic message to be issued).

A related problem exists in languages that allow overloading or overriding of keywords or standard library function identifiers. Such overloading can lead to confusion about which entity is intended to be referenced.

Definitions for new identifiers should not use a name that is already visible within the scope containing the new definition. Alternately, utilize language-specific facilities that check for and prevent inadvertent overloading of names should be used.

6.22.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- Languages that allow the same name to be used for identifiers defined in nested scopes.

- Languages where unique names can be transformed into non-unique names as part of the normal tool chain.

6.22.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and can be used in the same context. A language-specific project coding convention can be used to ensure that such errors are detectable with static analysis.
- Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and has a type that permits it to occur in at least one context where the first entity can occur.
- Use language features, if any, which explicitly mark definitions of entities that are intended to hide other definitions.
- Develop or use tools that identify name collisions or reuse when truncated versions of names cause conflicts.
- Ensure that all identifiers differ within the number of characters considered to be significant by the implementations that are likely to be used, and document all assumptions.

6.22.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should require mandatory diagnostics for variables with the same name in nested scopes.
- Languages should require mandatory diagnostics for variable names that exceed the length that the implementation considers unique.
- Languages should consider requiring mandatory diagnostics for overloading or overriding of keywords or standard library function identifiers.

6.23 Namespace Issues [BJL]

6.23.1 Description of Application Vulnerability

If a language provides separate, non-hierarchical namespaces, a user-controlled ordering of namespaces, and a means to make names declared in these name spaces directly visible to an application, the potential of unintentional and possible disastrous change in application behaviour can arise, when names are added to a namespace during maintenance.

Namespaces include constructs like packages, modules, libraries, classes or any other means of grouping declarations for import into other program units.

6.23.2 Cross references

MISRA C++ 2008: 7-3-1, 7-3-3, 7-3-5, 14-5-1, and 16-0-2

6.23.3 Mechanism of Failure

The failure is best illustrated by an example. Namespace N1 provides the name A but not B; Namespace N2 provides the name B but not A. The application wishes to use A from N1 and B from N2. At this point, there are no obvious issues. The application chooses (or needs to) import the namespaces to obtain names for direct usage, for an example.

Use N1, N2; – presumed to make all names in N1 and N2 directly visible

```
… X := A + B;
```

The semantics of the above example are intuitive and unambiguous.

Later, during maintenance, the name B is added to N1. The change to the namespace usually implies a recompilation of dependent units. At this point, two declarations of B are applicable for the use of B in the above example.

Some languages try to disambiguate the above situation by stating preference rules in case of such ambiguity among names provided by different name spaces. If, in the above example, N1 is preferred over N2, the meaning of the use of B changes silently, presuming that no typing error arises. Consequently the semantics of the program change silently and assuredly unintentionally, since the implementer of N1 cannot assume that all users of N1 would prefer to take any declaration of B from N1 rather than its previous namespace.

It does not matter what the preference rules actually are, as long as the namespaces are mutable. The above example is easily extended by adding A to N2 to show a symmetric error situation for a different precedence rule.

If a language supports overloading of subprograms, the notion of “same name” used in the above example is extended to mean not only the same name, but also the same signature of the subprogram. For vulnerabilities associated with overloading and overriding, see Identifier Name Reuse [YOW]. In the context of namespaces, however, adding signature matching to the name binding process, merely extends the described problem from simple names to full signatures, but does not alter the mechanism or quality of the described vulnerability. In particular, overloading does not introduce more ambiguity for binding to declarations in different name spaces. This vulnerability not only creates unintentional errors. It also can be exploited maliciously, if the source of the application and of the namespaces is known to the aggressor and one of the namespaces is mutable by the attacker.

6.23.4 Applicable Language Characteristics

The vulnerability is applicable to languages with the following characteristics:

- Languages that support non-hierarchical separate name-spaces, have means to import all names of a namespace “wholesale” for direct use, and have preference rules to choose among multiple imported direct homographs. All three conditions need to be satisfied for the vulnerability to arise.

6.23.5 Avoiding the Vulnerability or Mitigating its Effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Avoiding “wholesale” import directives
- Using only selective “single name” import directives or using fully qualified names (in both cases, provided that the language offers the respective capabilities)

6.23.6 Implications for Standardization

In future standardization activities, the following items should be considered:

- Languages should not have preference rules among mutable namespaces. Ambiguities should be invalid and avoidable by the user, for example, by using names qualified by their originating namespace.

6.24 Initialization of Variables [LAV]

6.24.1 Description of application vulnerability

Reading a variable that has not been assigned a value appropriate to its type can cause unpredictable execution in the block that uses the value of the variable, and has the potential to export bad values to callers, or cause out-of-bounds memory accesses.

Uninitialized variable usage is frequently not detected until after testing and often when the code in question is delivered and in use, because happenstance will provide variables with adequate values (such as default data settings or accidental left-over values) until some other change exposes the defect.

Variables that are declared during module construction (by a class constructor, instantiation, or elaboration) may have alternate paths that can read values before they are set. This can happen in straight sequential code but is more prevalent when concurrency or co-routines are present, with the same impacts described above.

Another vulnerability occurs when compound objects are initialized incompletely, as can happen when objects are incrementally built, or fields are added under maintenance.

When possible and supported by the language, whole-structure initialization is preferable to field-by-field initialization statements, and named association is preferable to positional, as it facilitates human review and is less susceptible to failures under maintenance. For classes, the declaration and initialization may occur in separate modules. In such cases it must be possible to show that every field that needs an initial value receives that value, and to document ones that do not require initial values.

6.24.2 Cross reference

CWE:
457. Use of Uninitialized Variable
JSF AV Rules: 71, 143, and 147
MISRA C 2004: 9.1, 9.2, and 9.3
MISRA C++ 2008: 8-5-1
CERT C guidelines: DCL14-C and EXP33-C
Ada Quality and Style Guide: 5.9.6

6.24.3 Mechanism of failure

Uninitialized objects may have invalid values, valid but wrong values, or valid and dangerous values. Wrong values could cause unbounded branches in conditionals or unbounded loop executions, or could simply cause wrong calculations and results.

There is a special case of pointers or access types. When such a type contains null values, a bound violation and hardware exception can result. When such a type contains plausible but meaningless values, random data reads and writes can collect erroneous data or can destroy data that is in use by another part of the program; when such a type is an access to a subprogram with a plausible (but wrong) value, then either a bad instruction trap may occur or a transfer to an unknown code fragment can occur. All of these scenarios can result in undefined behaviour.

Uninitialized variables are difficult to identify and use for attackers, but can be arbitrarily dangerous in safety situations.

6.24.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that permit variables to be read before they are assigned.

6.24.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- The general problem of showing that all objects are initialized is intractable; hence developers must carefully structure programs to show that all variables are set before first read on every path throughout the subprogram. Where objects are visible from many modules, it is difficult to determine where the first read occurs, and identify a module that must set the value before that read. When concurrency, interrupts and coroutines are present, it becomes especially imperative to identify where early initialization occurs and to show that the correct order is set via program structure, not by timing, OS precedence, or chance.
- The simplest method is to initialize each object at elaboration time, or immediately after subprogram execution commences and before any branches. If the subprogram must commence with conditional statements, then the programmer is responsible to show that every variable declared and not initialized earlier is initialized on each branch.
- Applications can consider defining or reserving fields or portions of the object to only be set when fully initialized. However, this approach has the effect of setting the variable to possibly mistaken values while defeating the use of static analysis to find the uninitialized variables.
- It should be possible to use static analysis tools to show that all objects are set before use in certain specific cases, but as the general problem is intractable, programmers should keep initialization algorithms simple so that they can be analyzed.
- When declaring and initializing the object together, if the language does not require that the compiler statically verify that the declarative structure and the initialization structure match, use static analysis tools to help detect any mismatches.

- When setting compound objects, if the language provides mechanisms to set all components together, use those in preference to a sequence of initializations as this helps coverage analysis; otherwise use tools that perform such coverage analysis and document the initialization. Do not perform partial initializations unless there is no choice, and document any deviations from 100% initialization.
- Where default assignments of multiple components are performed, explicit declaration of the component names and/or ranges helps static analysis and identification of component changes during maintenance.
- Some languages have named assignments that can be used to build reviewable assignment structures that can be analyzed by the language processor for completeness. Languages with positional notation only can use comments and secondary tools to help show correct assignment.

6.24.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Some languages have ways to determine if modules and regions are elaborated and initialized and to raise exceptions if this does not occur. Languages that do not could consider adding such capabilities.
- Languages could consider setting aside fields in all objects to identify if initialization has occurred, especially for security and safety domains.
- Languages that do not support whole-object initialization could consider adding this capability.

6.25 Operator Precedence/Order of Evaluation [JCW]

6.25.1 Description of application vulnerability

Each language provides rules of precedence and associativity, for each expression that operands bind to which operators. These rules are also known as “grouping” or “binding”.

Experience and experimental evidence shows that developers can have incorrect beliefs about the relative precedence of many binary operators. See, *Developer beliefs about binary operator precedence*. C Vu, 18(4):14-21, August 2006

6.25.2 Cross reference

JSF AV Rules: 204 and 213
MISRA C 2004: 12.1, 12.2, 12.5, 12.6, 13.2, 19.10, 19.12, and 19.13
MISRA C++ 2008: 4-5-1, 4-5-2, 4-5-3, 5-0-1, 5-0-2, 5-2-1, 5-3-1, 16-0-6, 16-3-1, and 16-3-2
CERT C guidelines: EXP00-C
Ada Quality and Style Guide: 7.1.8 and 7.1.9

6.25.3 Mechanism of failure

In C and C++, the bitwise operators (bitwise logical and bitwise shift) are sometimes thought of by the programmer having similar precedence to arithmetic operations, so just as one might correctly write “`x - 1 == 0`” (“`x` minus one is equal to zero”), a programmer might erroneously write “`x & 1 == 0`”, mentally thinking “`x` anded-with `1` is equal to zero”, whereas the operator precedence rules of C and C++ actually bind the expression

as "compute `1==0`, producing 'false' interpreted as zero, then bitwise-and the result with `x`", producing (a constant) zero, contrary to the programmer's intent.

Examples from an opposite extreme can be found in programs written in APL, which is noteworthy for the absence of *any* distinctions of precedence. One commonly made mistake is to write "`a * b + c`", intending to produce "`a` times `b` plus `c`", whereas APL's uniform right-to-left associativity produces "`b` plus `c`, times `a`".

6.25.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages whose precedence and associativity rules are sufficiently complex that developers do not remember them.

6.25.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Adopt programming guidelines (preferably augmented by static analysis). For example, consider the rules itemized above from JSF AV [15], CERT C [11] or MISRA C [12].
- Use parentheses around binary operator combinations that are known to be a source of error (for example, mixed arithmetic/bitwise and bitwise/relational operator combinations).
- Break up complex expressions and use temporary variables to make the order clearer.

6.25.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language definitions should avoid providing precedence or a particular associativity for operators that are not typically ordered with respect to one another in arithmetic, and instead require full parenthesization to avoid misinterpretation.

6.26 Side-effects and Order of Evaluation [SAM]

6.26.1 Description of application vulnerability

Some programming languages allow subexpressions to cause side-effects (such as assignment, increment, or decrement). For example, some programming languages permit such side-effects, and if, within one expression (such as "`i = v[i++]`"), two or more side-effects modify the same object, undefined behaviour results.

Some languages allow subexpressions to be evaluated in an unspecified ordering, or even removed during optimization. If these subexpressions contain side-effects, then the value of the full expression can be dependent upon the order of evaluation. Furthermore, the objects that are modified by the side-effects can receive values that are dependent upon the order of evaluation.

If a program contains these unspecified or undefined behaviours, testing the program and seeing that it yields the expected results may give the false impression that the expression will always yield the expected result.

6.26.2 Cross reference

JSF AV Rules: 157, 158, 166, 204, 204.1, and 213
MISRA C 2004: 12.1-12.5
MISRA C++ 2008: 5-0-1
CERT C guidelines: EXP10-C, EXP30-C
Ada Quality and Style Guide: 7.1.8 and 7.1.9

6.26.3 Mechanism of failure

When subexpressions with side effects are used within an expression, the unspecified order of evaluation can result in a program producing different results on different platforms, or even at different times on the same platform. For example, consider

```
a = f(b) + g(b);
```

where `f` and `g` both modify `b`. If `f(b)` is evaluated first, then the `b` used as a parameter to `g(b)` may be a different value than if `g(b)` is performed first. Likewise, if `g(b)` is performed first, `f(b)` may be called with a different value of `b`.

Other examples of unspecified order, or even undefined behaviour, can be manifested, such as

```
a = f(i) + i++;
```

or

```
a[i++] = b[i++];
```

Parentheses around expressions can assist in removing ambiguity about grouping, but the issues regarding side-effects and order of evaluation are not changed by the presence of parentheses; consider

```
j = i++ * i++;
```

where even if parentheses are placed around the `i++` subexpressions, undefined behaviour still remains. (All examples use the syntax of C or Java for brevity; the effects can be created in any language that allows functions with side-effects in the places where C allows the increment operations.)

The unpredictable nature of the calculation means that the program cannot be tested adequately to any degree of confidence. A knowledgeable attacker can take advantage of this characteristic to manipulate data values triggering execution that was not anticipated by the developer.

6.26.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that permit expressions to contain subexpressions with side effects.
- Languages whose subexpressions are computed in an unspecified ordering.

6.26.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Make use of one or more programming guidelines which (a) prohibit these unspecified or undefined behaviours, and (b) can be enforced by static analysis. (See JSF AV and MISRA rules in Cross reference clause [SAM])
- Keep expressions simple. Complicated code is prone to error and difficult to maintain.

6.26.6 Implications for standardization

In future standardization activities, the following items should be considered:

- In developing new or revised languages, give consideration to language features that will eliminate or mitigate this vulnerability, such as pure functions.

6.27 Likely Incorrect Expression [KOA]

6.27.1 Description of application vulnerability

Certain expressions are symptomatic of what is likely to be a mistake made by the programmer. The statement is not wrong, but it is unlikely to be right. The statement may have no effect and effectively is a null statement or may introduce an unintended side-effect. A common example is the use of = in an `if` expression in C where the programmer meant to do an equality test using the == operator. Other easily confused operators in C are the logical operators such as `&&` for the bitwise operator `&`, or vice versa. It is valid and possible that the programmer intended to do an assignment within the `if` expression, but due to this being a common error, a programmer doing so would be using a poor programming practice. A less likely occurrence, but still possible is the substitution of == for = in what is supposed to be an assignment statement, but which effectively becomes a null statement. These mistakes may survive testing only to manifest themselves in deployed code where they may be maliciously exploited.

6.27.2 Cross reference

CWE:

480. Use of Incorrect Operator
481. Assigning instead of Comparing
482. Comparing instead of Assigning
570. Expression is Always False
571. Expression is Always True

JSF AV Rules: 160 and 166
MISRA C 2004: 12.3, 12.4, 12.13, 13.1, 13.7, and 14.2
MISRA C++ 2008: 0-1-9, 5-0-1, 6-2-1, and 6-5-2
CERT C guidelines: MSC02-C and MSC03-C

6.27.3 Mechanism of failure

Some of the failures are simply a case of programmer carelessness. Substitution of = instead of == in a Boolean test is easy to do and most C and C++ programmers have made this mistake at one time or another. Other instances can be the result of intricacies of the language definition that specifies what part of an expression must be evaluated. For instance, having an assignment expression in a Boolean statement is likely making an assumption that the complete expression will be executed in all cases. However, this is not always the case as sometimes the truth-value of the Boolean expression can be determined after only executing some portion of the expression. For instance:

```
if ((a == b) | (c = (d-1)))
```

There is no guarantee which of the two subexpressions `(a == b)` or `(c=(d-1))` will be executed first. Should `(a==b)` be determined to be true, then there is no need for the subexpression `(c=(d-1))` to be executed and as such, the assignment `(c=(d-1))` will not occur.

Embedding expressions in other expressions can yield unexpected results. Increment and decrement operators (++ and −−) can also yield unexpected results when mixed into a complex expression.

Incorrectly calculated results can lead to a wide variety of erroneous program execution

6.27.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- All languages are susceptible to likely incorrect expressions.

6.27.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Simplify expressions.
- Do not use assignment expressions as function parameters. Sometimes the assignment may not be executed as expected. Instead, perform the assignment before the function call.
- Do not perform assignments within a Boolean expression. This is likely unintended, but if not, then move the assignment outside of the Boolean expression for clarity and robustness.
- On some rare occasions, some statements intentionally do not have side effects and do not cause control flow to change. These should be annotated through comments and made obvious that they are intentionally no-ops with a stated reason. If possible, such reliance on null statements should be avoided. In general, except for those rare instances, all statements should either have a side effect or cause control flow to change.

6.27.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing warnings for statements that are unlikely to be right such as statements without side effects. A null (no-op) statement may need to be added to the language for those rare instances where an intentional null statement is needed. Having a null statement as part of the language will reduce confusion as to why a statement with no side effects is present in code.
- Languages should consider not allowing assignments used as function parameters.
- Languages should consider not allowing assignments within a Boolean expression.
- Language definitions should avoid situations where easily confused symbols (such as = and ==, or `;` and `:`, or `!`= and `/`=) are valid in the same context. For example, = is not generally valid in an `if` statement in Java because it does not normally return a boolean value.

6.28 Dead and Deactivated Code [XYQ]

6.28.1 Description of application vulnerability

Dead and Deactivated code (the distinction is addressed in [XYQ]) is code that exists in the executable, but which can never be executed, either because there is no call path that leads to it (for example, a function that is never called), or the path is semantically infeasible (for example, its execution depends on the state of a conditional that can never be achieved).

Dead and Deactivated code may be undesirable because it may indicate the possibility of a coding error. A security issue is also possible if a "jump target" is injected. Many safety standards prohibit dead code because dead code is not traceable to a requirement.

Also covered in this vulnerability is code which is believed to be dead, but which is inadvertently executed.

Dead and Deactivated code is considered different than used data, used data is covered in a different vulnerability, see [YZS].

6.28.2 Cross reference

CWE:

561. Dead Code
570. Expression is Always False
571. Expression is Always True

JSF AV Rules: 127 and 186
MISRA C 2004: 2.4 and 14.1
MISRA C++ 2008: 0-1-1 to 0-1-10, 2-7-2, and 2-7-3
CERT C guidelines: MSC07-C and MSC12-C
DO-178B/C

6.28.3 Mechanism of failure

DO-178B defines Dead and Deactivated code as:

- Dead code – Executable object code (or data) which... cannot be executed (code) or used (data) in an operational configuration of the target computer environment and is not traceable to a system or software requirement.

- Deactivated code – Executable object code (or data) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options.

Dead code is code that exists in an application, but which can never be executed, either because there is no call path to the code (for example, a function that is never called) or because the execution path to the code is semantically infeasible, as in

```
integer i = 0;
if ( i == 0)
   then fun_a();
   else fun_b();
```

`fun_b` is dead code, as only `fun_a` can ever be executed.

Compilers that optimize sometimes generate and then remove dead code, including code placed there by the programmer. The deadness of code can also depend on the linking of separately compiled modules.

The presence of dead code is not in itself an error its presence may be an indication that the developer believed it to be necessary, but some error means it will never be executed? There may also be legitimate reasons for its presence, for example:

- Defensive code, only executed as the result of a hardware failure.
- Code that is part of a library not required in this application.
- Diagnostic code not executed in the operational environment.
- Code that is temporarily deactivated buy may be needed soon. This may occur as a way to make sure the code is still accepted by the language translator to reduce opportunities for errors when it is reactivated.
- Code that is made available so that it can be executed manually via a debugger

Such code may be referred to as “deactivated”. That is, dead code that is there by intent.

There is a secondary consideration for dead code in languages that permit overloading of functions and other constructs that use complex name resolution strategies. The developer may believe that some code is not going to be used (deactivated), but its existence in the program means that it appears in the namespace, and may be selected as the best match for some use that was intended to be of an overloading function. That is, although the developer believes it is never going to be used, in practice it is used in preference to the intended function.

6.28.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that allow code to exist in the executable that can never be executed.

6.28.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- The developer should endeavor to remove dead code from an application unless its presence serves a purpose.
- When a developer identifies code that is dead because a conditional always evaluates to the same value, this could be indicative of an earlier bug and additional testing may be needed to ascertain why the same value is occurring.
- The developer should identify any dead code in the application, and provide a justification (if only to themselves) as to why it is there.
- The developer should also ensure that any code that was expected to be unused is actually recognized as dead code.
- The developer should apply standard branch coverage measurement tools and ensure by 100% coverage that all branches are neither dead nor deactivated

6.28.6 Implications for standardization

[None]

6.29 Switch Statements and Static Analysis [CLL]

6.29.1 Description of application vulnerability

Many programming languages provide a construct, such as a `switch` statement, that chooses among multiple alternative control flows based upon the evaluated result of an expression. The use of such constructs may introduce application vulnerabilities if not all possible cases appear within the switch or if control unexpectedly flows from one alternative to another.

6.29.2 Cross reference

JSF AV Rules: 148, 193, 194, 195, and 196
MISRA C 2004: 15.2, 15.3, and 15.5
MISRA C++ 2008: 6-4-3, 6-4-5, 6-4-6, and 6-4-8
CERT C guidelines: MSC01-C
Ada Quality and Style Guide: 5.6.1 and 5.6.10

6.29.3 Mechanism of failure

The fundamental challenge when using a `switch` statement is to make sure that all possible cases are, in fact, treated correctly.

6.29.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that contain a construct, such as a `switch` statement, that provides a selection among alternative control flows based on the evaluation of an expression.
- Languages that do not require full coverage of a `switch` statement.
- Languages that provide a default case (choice) in a `switch` statement.

6.29.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Switch on an expression that has a small number of potential values that can be statically enumerated. In languages that provide them, a variable of an enumerated type is to be preferred because a possible set of values is known statically and is small in number (as compared, for example, to the value set of an integer variable). Where it is practical to statically enumerate the switched type, it is preferable to omit the default case, because the static analysis is simplified and because maintainers can better understand the intent of the original programmer. When one must switch on some other form of type, it is necessary to have a default case, preferably to be regarded as a serious error condition.
- Avoid "flowing through" from one case to another. Even if correctly implemented, it is difficult for reviewers and maintainers to distinguish whether the construct was intended or is an error of omission[4]. In cases where flow-through is necessary and intended, an explicitly coded branch may be preferable to clearly mark the intent. Providing comments regarding intention can be helpful to reviewers and maintainers.
- Perform static analysis to determine if all cases are, in fact, covered by the code. (Note that the use of a default case can hamper the effectiveness of static analysis since the tool cannot determine if omitted alternatives were or were not intended for default treatment.)
- Other means of mitigation include manual review, bounds testing, tool analysis, verification techniques, and proofs of correctness.

6.29.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specifications could require compilers to ensure that a complete set of alternatives is provided in cases where the value set of the switch variable can be statically determined.

6.30 Demarcation of Control Flow [EOJ]

6.30.1 Description of application vulnerability

Some programming languages explicitly mark the end of an `if` statement or a loop, whereas other languages mark only the end of a block of statements. Languages of the latter category are prone to oversights by the programmer, causing unintended sequences of control flow.

6.30.2 Cross reference

JSF AV Rules: 59 and 192
MISRA C 2004: 14.8, 14.9, 14.10, and 19.5
MISRA C++ 2008: 6-3-1, 6-4-1, 6-4-2, 6-4-3, 6-4-8, 6-5-1, 6-5-6, 6-6-1 to 6-6-5, and16-0-2
Hatton 18: Control flow – `if` structure
Ada Quality and Style Guide: 3, 5.6.1 through 5.6.10

[4] Using multiple labels on individual alternatives is not a violation of this recommendation, though.

6.30.3 Mechanism of failure

Programmers may rely on indentation to determine inclusion of statements within constructs. Testing of the software may not reveal that statements thought to be included in an `if-then`, `if-then-else`, or loops that are not in reality a part of the `if` statement. Moreover, for a nested `if-then-else` statement the programmer may be confused about which `if` statement controls the `else` part directly. This can lead to unexpected results.

6.30.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that contain loops and conditional statements that are not explicitly terminated by an "end" construct.

6.30.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Adopt a convention for marking the closing of a construct that can be checked by a tool, to ensure that program structure is apparent.
- Adopt programming guidelines (preferably augmented by static analysis). For example, consider the rules itemized above from JSF AV, MISRA C, MISRA C++ or Hatton.
- Other means of assurance might include proofs of correctness, analysis with tools, verification techniques, or other methods.
- Pretty-printers and syntax-aware editors may be helpful in finding such problems, but sometimes disguise them.
- Include a final else statement at the end of `if-…-else-if` constructs to avoid confusion.
- Always enclose the body of statements of an `if`, `while`, `for`, or other statements potentially introducing a block of code in braces ("`{}`") or other demarcation indicators appropriate to the language used.

6.30.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Specifiers of languages should consider adding a mode that strictly enforces compound conditional and looping constructs with explicit termination, such as "end if" or a closing bracket.
- Specifiers of languages might consider explicit termination of loops and conditional statements.
- Specifiers might consider features to terminate named loops and conditionals and determine if the structure as named matches the structure as inferred.

6.31 Loop Control Variables [TEX]

6.31.1 Description of application vulnerability

Many languages support a looping construct whose number of iterations is controlled by the value of a loop control variable. Looping constructs provide a method of specifying an initial value for this loop control variable, a

test that terminates the loop and the quantity by which it should be decremented or incremented on each loop iteration.

In some languages it is possible to modify the value of the loop control variable within the body of the loop. Experience shows that such value modifications are sometimes overlooked by readers of the source code, resulting in faults being introduced.

6.31.2 Cross reference

JSF AV Rule: 201
MISRA C 2004: 13.6
MISRA C++ 2008: 6-5-1 to 6-5-6

6.31.3 Mechanism of failure

Readers of source code often make assumptions about what has been written. A common assumption is that a loop control variable is a constant since such variables are not usually modified in the body of the associated loop. A reader of the source may incorrectly assume that a loop control variable is not modified in the body of its loop and write (incorrect) code based on this assumption.

6.31.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that permit a loop control variable to be modified in the body of its associated loop.

6.31.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Not modifying a loop control variable in the body of its associated loop body.
- Some languages, such as C and C++ do not explicitly specify which of the variables appearing in a loop header is the control variable for the loop. MISRA C [12] and MISRA C++ [16] have proposed algorithms for deducing which, if any, of these variables is the loop control variable in the programming languages C and C++ (these algorithms could also be applied to other languages that support a C-like for-loop).

6.31.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language designers should consider the addition of an identifier type for loop control that cannot be modified by anything other than the loop control construct.

6.32 Off-by-one Error [XZH]

6.32.1 Description of application vulnerability

A program uses an incorrect maximum or minimum value that is 1 more or 1 less than the correct value. This usually arises from one of a number of situations where the bounds as understood by the developer differ from the design, such as:

- Confusion between the need for < and <= or > and >= in a test.
- Confusion as to the index range of an algorithm, such as: beginning an algorithm at 1 when the underlying structure is indexed from 0; beginning an algorithm at 0 when the underlying structure is indexed from 1 (or some other start point); or using the length of a structure as its bound instead of the sentinel values.
- Failing to allow for storage of a sentinel value, such as the `NULL` string terminator that is used in the C and C++ programming languages.

These issues arise from mistakes in mapping the design into a particular language, in moving between languages (such as between languages where all arrays start at 0 and other languages where arrays start at 1), and when exchanging data between languages with different default array bounds.

The issue also can arise in algorithms where relationships exist between components, and the existence of a bounds value changes the conditions of the test.

The existence of this possible flaw can also be a serious security hole as it can permit someone to surreptitiously provide an unused location (such as 0 or the last element) that can be used for undocumented features or hidden channels.

6.32.2 Cross reference

CWE:

193. Off-by-one Error

6.32.3 Mechanism of failure

An off-by-one error could lead to:

- an out-of bounds access to an array (buffer overflow),
- incomplete comparisons or calculation mistakes,
- a read from the wrong memory location, or
- an incorrect conditional.

Such incorrect accesses can cause cascading errors or references to invalid locations, resulting in potentially unbounded behaviour.

Off-by-one errors are not often exploited in attacks because they are difficult to identify and exploit externally, but the cascading errors and boundary-condition errors can be severe.

6.32.4 Applicable language characteristics

As this vulnerability arises because of an algorithmic error by the developer, it can in principle arise in any language; however, it is most likely to occur when:

- The language relies on the developer having implicit knowledge of structure start and end indices (for example, knowing whether arrays start at 0 or 1 – or indeed some other value).
- Where the language relies upon explicit bounds values to terminate variable length arrays.

6.32.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- A systematic development process, use of development/analysis tools and thorough testing are all common ways of preventing errors, and in this case, off-by-one errors.
- Where references are being made to structure indices and the languages provide ways to specify the whole structure or the starting and ending indices explicitly (for example, Ada provides xxx'First and xxx'Last for each dimension), these should be used always. Where the language doesn't provide these, constants can be declared and used in preference to numeric literals.
- Where the language doesn't encapsulate variable length arrays, encapsulation should be provided through library objects and a coding standard developed that requires such arrays to only be used via those library objects, so the developer does not need to be explicitly concerned with managing bounds values.

6.32.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should provide encapsulations for arrays that:
  - Prevent the need for the developer to be concerned with explicit bounds values.
  - Provide the developer with symbolic access to the array start, end and iterators.

6.33 Structured Programming [EWD]

6.33.1 Description of application vulnerability

Programs that have a convoluted control structure are likely to be more difficult to be human readable, less understandable, harder to maintain, more difficult to modify, harder to statically analyze, more difficult to match the allocation and release of resources, and more likely to be incorrect.

6.33.2 Cross reference

JSF AV Rules: 20, 113, 189, 190, and 191
MISRA C 2004: 14.4, 14.5, and 20.7
MISRA C++ 2008: 6-6-1, 6-6-2, 6-6-3, and 17-0-5
CERT C guidelines: SIG32-C
Ada Quality and Style Guide: 3, 4, 5.4, 5.6, and 5.7

6.33.3 Mechanism of failure

Lack of structured programming can lead to:

- Memory or resource leaks.
- Error prone maintenance.
- Design that is difficult or impossible to validate.
- Source code that is difficult or impossible to statically analyze.

6.33.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that allow leaving a loop without consideration for the loop control.
- Languages that allow local jumps (`goto` statement).
- Languages that allow non-local jumps (`setjmp`/`longjmp` in the C programming language).
- Languages that support multiple entry and exit points from a function, procedure, subroutine or method.

6.33.5 Avoiding the vulnerability or mitigating its effects

Use only those features of the programming language that enforce a logical structure on the program. The program flow follows a simple hierarchical model that employs looping constructs such as `for`, `repeat`, `do`, and `while`.

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Avoid using language features such as `goto`.
- Avoid using language features such as `continue` and `break` in the middle of loops.
- Avoid using language features that transfer control of the program flow via a jump.
- Avoid multiple exit points to a function/procedure/method/subroutine.
- Avoid multiple entry points to a function/procedure/method/subroutine.

6.33.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should support and favor structured programming through their constructs to the extent possible.

6.34 Passing Parameters and Return Values [CSJ]

6.34.1 Description of application vulnerability

Nearly every procedural language provides some method of process abstraction permitting decomposition of the flow of control into routines, functions, subprograms, or methods. (For the purpose of this description, the term subprogram will be used.) To have any effect on the computation, the subprogram must change data visible to the calling program. It can do this by changing the value of a non-local variable, changing the value of a parameter, or, in the case of a function, providing a return value. Because different languages use different

mechanisms with different semantics for passing parameters, a programmer using an unfamiliar language may obtain unexpected results.

6.34.2 Cross reference

JSF AV Rules: 116, 117, and 118
MISRA C 2004: 16.1, 16.2, 16.3, 16.4, 16.5, 16.6, 16.7, and 16.9
MISRA C++ 2008: 0-3-2, 7-1-2, 8-4-1, 8-4-2, 8-4-3, and 8-4-4
CERT C guidelines: EXP12-C and DCL33-C
Ada Quality and Style Guide: 5.2 and 8.3

6.34.3 Mechanism of failure

The mechanisms for parameter passing include: *call by reference*, *call by copy*, and *call by name*. The last is so specialized and supported by so few programming languages that it will not be treated in this description.

In call by reference, the calling program passes the addresses of the arguments to the called subprogram. When the subprogram references the corresponding formal parameter, it is actually sharing data with the calling program. If the subprogram changes a formal parameter, then the corresponding actual argument is also changed. If the actual argument is an expression or a constant, then the address of a temporary location is passed to the subprogram; this may be an error in some languages.

In call by copy, the called subprogram does not share data with the calling program. Instead, formal parameters act as local variables. Values are passed between the actual arguments and the formal parameters by copying. Some languages may control changes to formal parameters based on labels such as `in`, `out`, or `inout`. There are three cases to consider: *call by value* for `in` parameters; *call by result* for `out` parameters and function return values; and *call by value-result* for `inout` parameters. For call by value, the calling program evaluates the actual arguments and copies the result to the corresponding formal parameters that are then treated as local variables by the subprogram. For call by result, the values of the locals corresponding to formal parameters are copied to the corresponding actual arguments. For call by value-result, the values are copied in from the actual arguments at the beginning of the subprogram's execution and back out to the actual arguments at its termination.

The obvious disadvantage of call by copy is that extra copy operations are needed and execution time is required to produce the copies. Particularly if parameters represent sizable objects, such as large arrays, the cost of call by copy can be high. For this reason, many languages also provide the call by reference mechanism. The disadvantage of call by reference is that the calling program cannot be assured that the subprogram hasn't changed data that was intended to be unchanged. For example, if an array is passed by reference to a subprogram intended to sum its elements, the subprogram could also change the values of one or more elements of the array. However, some languages enforce the subprogram's access to the shared data based on the labeling of actual arguments with modes—such as `in`, `out`, or `inout` or by constant pointers.

Another problem with call by reference is unintended aliasing. It is possible that the address of one actual argument is the same as another actual argument or that two arguments overlap in storage. A subprogram, assuming the two formal parameters to be distinct, may treat them inappropriately. For example, if one codes a subprogram to swap two values using the exclusive-or method, then a call to `swap(x,x)` will zero the value of `x`. Aliasing can also occur between arguments and non-local objects. For example, if a subprogram modifies a

non-local object as a side-effect of its execution, referencing that object by a formal parameter will result in aliasing and, possibly, unintended results.

Some languages provide only simple mechanisms for passing data to subprograms, leaving it to the programmer to synthesize appropriate mechanisms. Often, the only available mechanism is to use call by copy to pass small scalar values or pointer values containing addresses of data structures. Of course, the latter amounts to using call by reference with no checking by the language processor. In such cases, subprograms can pass back pointers to anything whatsoever, including data that is corrupted or absent.

Some languages use call by copy for small objects, such as scalars, and call by reference for large objects, such as arrays. The choice of mechanism may even be implementation-defined. Because the two mechanisms produce different results in the presence of aliasing, it is very important to avoid aliasing.

An additional problem may occur if the called subprogram fails to assign a value to a formal parameter that the caller expects as an output from the subprogram. In the case of call by reference, the result may be an uninitialized variable in the calling program. In the case of call by copy, the result may be that a legitimate initialization value provided by the caller is overwritten by an uninitialized value because the called program did not make an assignment to the parameter. This error may be difficult to detect through review because the failure to initialize is hidden in the subprogram.

An additional complication with subprograms occurs when one or more of the arguments are expressions. In such cases, the evaluation of one argument might have side-effects that result in a change to the value of another or unintended aliasing. Implementation choices regarding order of evaluation could affect the result of the computation. This particular problem is described in Side-effects and Order of Evaluation clause [SAM].

6.34.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that provide mechanisms for defining subprograms where the data passes between the calling program and the subprogram via parameters and return values. This includes methods in many popular object-oriented languages.

6.34.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use available mechanisms to label parameters as constants or with modes like `in`, `out`, or `inout`.
- When a choice of mechanisms is available, pass small simple objects using call by copy.
- When a choice of mechanisms is available and the computational cost of copying is tolerable, pass larger objects using call by copy.
- When the choice of language or the computational cost of copying forbids using call by copy, then take safeguards to prevent aliasing:
    - Minimize side-effects of subprograms on non-local objects; when side-effects are coded, ensure that the affected non-local objects are not passed as parameters using call by reference.
    - To avoid unintentional aliasing, avoid using expressions or functions as actual arguments; instead assign the result of the expression to a temporary local and pass the local.

- Utilize tooling or other forms of analysis to ensure that non-obvious instances of aliasing are absent.
- Perform reviews or analysis to determine that called subprograms fulfill their responsibilities to assign values to all output parameters.

6.34.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Programming language specifications could provide labels—such as `in`, `out`, and `inout`—that control the subprogram's access to its formal parameters, and enforce the access.

6.35 Dangling References to Stack Frames [DCM]

6.35.1 Description of application vulnerability

Many languages allow treating the address of a local variable as a value stored in other variables. Examples are the application of the address operator in C or C++, or of the 'Access or 'Address attributes in Ada. In some languages, this facility is also used to model the call-by-reference mechanism by passing the address of the actual parameter by-value. An obvious safety requirement is that the stored address shall not be used after the lifetime of the local variable has expired. This situation can be described as a "dangling reference to the stack".

6.35.2 Cross reference

CWE:
562. Return of Stack Variable Address
JSF AV Rule: 173
MISRA C 2004: 17.6 and 21.1
MISRA C++ 2008: 0-3-1, 7-5-1, 7-5-2, and 7-5-3
CERT C guidelines: EXP35-C and DCL30-C
Ada Quality and Style Guide: 7.6.7, 7.6.8, and 10.7.6

6.35.3 Mechanism of failure

The consequences of dangling references to the stack come in two variants: a deterministically predictable variant, which therefore can be exploited, and an intermittent, non-deterministic variant, which is next to impossible to elicit during testing. The following code sample illustrates the two variants; the behaviour is not language-specific:

```
struct s {  … };
typedef struct s array_type[1000];
array_type* ptr;
array_type* F()
{
  struct s Arr[1000];
  ptr = &Arr;     // Risk of variant 1;
  return &Arr;    // Risk of variant 2;
```

```
  }
  …
    struct s secret;
    array_type* ptr2;
    ptr2 = F();
    secret = (*ptr2)[10];   // Fault of variant 2
   …
secret = (*ptr)[10];    // Fault of variant 1
```

The risk of variant 1 is the assignment of the address of `Arr` to a pointer variable that survives the lifetime of `Arr`. The fault is the subsequent use of the dangling reference to the stack, which references memory since altered by other calls and possibly validly owned by other routines. As part of a call-back, the fault allows systematic examination of portions of the stack contents without triggering an array-bounds-checking violation. Thus, this vulnerability is easily exploitable. As a fault, the effects can be most astounding, as memory gets corrupted by completely unrelated code portions. (A life-time check as part of pointer assignment can prevent the risk. In many cases, such as the situations above, the check is statically decidable by a compiler. However, for the general case, a dynamic check is needed to ensure that the copied pointer value lives no longer than the designated object.)

The risk of variant 2 is an idiom "seen in the wild" to return the address of a local variable to avoid an expensive copy of a function result, as long as it is consumed before the next routine call occurs. The idiom is based on the ill-founded assumption that the stack will not be affected by anything until this next call is issued. The assumption is false, however, if an interrupt occurs and interrupt handling employs a strategy called "stack stealing", which is, using the current stack to satisfy its memory requirements. Thus, the value of `Arr` can be overwritten before it can be retrieved after the call on `F`. As this fault will only occur if the interrupt arrives after the call has returned but before the returned result is consumed, the fault is highly intermittent and next to impossible to re-create during testing. Thus, it is unlikely to be exploitable, but also exceedingly hard to find by testing. It can begin to occur after a completely unrelated interrupt handler has been coded or altered. Only static analysis can relatively easily detect the danger (unless the code combines it with risks of variant 1). Some compilers issue warnings for this situation; such warnings need to be heeded, and some forms of static analysis are effective in identifying such problems.

6.35.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- The address of a local entity (or formal parameter) of a routine can be obtained and stored in a variable or can be returned by this routine as a result.
- No check is made that the lifetime of the variable receiving the address is no larger than the lifetime of the designated entity.

6.35.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Do not use the address of locally declared entities as storable, assignable or returnable value (except where idioms of the language make it unavoidable).

- Where unavoidable, ensure that the lifetime of the variable containing the address is completely enclosed by the lifetime of the designated object.
- Never return the address of a local variable as the result of a function call.

6.35.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Do not provide means to obtain the address of a locally declared entity as a storable value; or
- Define implicit checks to implement the assurance of enclosed lifetime expressed in sub-clause 5 of this vulnerability. Note that, in many cases, the check is statically decidable, for example, when the address of a local entity is taken as part of a return statement or expression.

6.36 Subprogram Signature Mismatch [OTR]

6.36.1 Description of application vulnerability

If a subprogram is called with a different number of parameters than it expects, or with parameters of different types than it expects, then the results will be incorrect. Depending on the language, the operating environment, and the implementation, the error might be as benign as a diagnostic message or as extreme as a program continuing to execute with a corrupted stack. The possibility of a corrupted stack provides opportunities for penetration.

6.36.2 Cross reference

CWE:

628. Function Call with Incorrectly Specified Arguments
686. Function Call with Incorrect Argument Type
683. Function Call with Incorrect Order of Arguments

JSF AV Rule: 108
MISRA C 2004: 8.1, 8.2, 8.3, 16.1, 16.3, 16.4, 16.5, and 16.6
MISRA C++ 2008: 0-3-2, 3-2-1, 3-2-2, 3-2-3, 3-2-4, 3-3-1, 3-9-1, 8-3-1, 8-4-1, and 8-4-2
CERT C guidelines: DCL31-C, and DCL35-C

6.36.3 Mechanism of failure

When a subprogram is called, the actual arguments of the call are pushed on to the execution stack. When the subprogram terminates, the formal parameters are popped off the stack. If the number and type of the actual arguments do not match the number and type of the formal parameters, then the push and the pop will not be commensurable and the stack will be corrupted. Stack corruption can lead to unpredictable execution of the program and can provide opportunities for execution of unintended or malicious code.

The compilation systems for many languages and implementations can check to ensure that the list of actual parameters and any expected return match the declared set of formal parameters and return value (the *subprogram signature*) in both number and type. (In some cases, programmers should observe a set of conventions to ensure that this is true.) However, when the call is being made to an externally compiled

subprogram, an object-code library, or a module compiled in a different language, the programmer must take additional steps to ensure a match between the expectations of the caller and the called subprogram.

6.36.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that do not require their implementations to ensure that the number and types of actual arguments are equal to the number and types of the formal parameters.
- Implementations that permit programs to call subprograms that have been externally compiled (without a means to check for a matching subprogram signature), subprograms in object code libraries, and any subprograms compiled in other languages.

6.36.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Take advantage of any mechanism provided by the language to ensure that subprogram signatures match.
- Avoid any language features that permit variable numbers of actual arguments without a method of enforcing a match for any instance of a subprogram call.
- Take advantage of any language or implementation feature that would guarantee matching the subprogram signature in linking to other languages or to separately compiled modules.
- Intensively review subprogram calls where the match is not guaranteed by tooling.

6.36.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specifiers could ensure that the signatures of subprograms match within a single compilation unit and could provide features for asserting and checking the match with externally compiled subprograms.

6.37 Recursion [GDL]

6.37.1 Description of application vulnerability

Recursion is an elegant mathematical mechanism for defining the values of some functions. It is tempting to write code that mirrors the mathematics. However, the use of recursion in a computer can have a profound effect on the consumption of finite resources, leading to denial of service.

6.37.2 Cross reference

CWE:
674. Uncontrolled Recursion
JSF AV Rule: 119
MISRA C 2004: 16.2

MISRA C++ 2008: 7-5-4
CERT C guidelines: MEM05-C
Ada Quality and Style Guide: 5.6.6

6.37.3 Mechanism of failure

Recursion provides for the economical definition of some mathematical functions. However, economical definition and economical calculation are two different subjects. It is tempting to calculate the value of a recursive function using recursive subprograms because the expression in the programming language is straightforward and easy to understand. However, the impact on finite computing resources can be profound. Each invocation of a recursive subprogram may result in the creation of a new stack frame, complete with local variables. If stack space is limited and the calculation of some values will lead to an exhaustion of resources resulting in the program terminating.

In calculating the values of mathematical functions the use of recursion in a program is usually obvious, but this is not true in the general case. For example, finalization of a computing context after treating an error condition might result in recursion (such as attempting to "clean up" by closing a file after an error was encountered in closing the same file). Although such situations may have other problems, they typically do not result in exhaustion of resources but may otherwise result in a denial of service.

6.37.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Any language that permits the recursive invocation of subprograms.

6.37.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Minimize the use of recursion.
- Converting recursive calculations to the corresponding iterative calculation. In principle, any recursive calculation can be remodeled as an iterative calculation which will have a smaller impact on some computing resources but which may be harder for a human to comprehend. The cost to human understanding must be weighed against the practical limits of computing resource.
- In cases where the depth of recursion can be shown to be statically bounded by a tolerable number, then recursion may be acceptable, but should be documented for the use of maintainers.

It should be noted that some languages or implementations provide special (more economical) treatment of a form of recursion known as *tail-recursion*. In this case, the impact on computing economy is reduced. When using such a language, tail recursion may be preferred to an iterative calculation.

6.37.6 Implications for standardization

[None]

6.38 Ignored Error Status and Unhandled Exceptions [OYB]

6.38.1 Description of application vulnerability

Unpredicted faults and exceptional situations arise during the execution of code, preventing the intended functioning of the code. They are detected and reported by the language implementation or by explicit code written by the user. Different strategies and language constructs are used to report such errors and to take remedial action. Serious vulnerabilities arise when detected errors are reported but ignored or not properly handled.

6.38.2 Cross reference

CWE:
754. Improper Check for Unusual or Exceptional Conditions
JSF AV Rules: 115 and 208
MISRA C 2004: 16.10
MISRA C++ 2008: 15-3-2 and 19-3-1
CERT C guidelines: DCL09-C, ERR00-C, and ERR02-C

6.38.3 Mechanism of failure

The fundamental mechanism of failure is that the program does not react to a detected error or reacts inappropriately to it. Execution may continue outside the envelope provided by its specification, making additional errors or serious malfunction of the software likely. Alternatively, execution may terminate. The mechanism can be easily exploited to perform denial-of-service attacks.

The specific mechanism of failure depends on the error reporting and handling scheme provided by a language or applied idiomatically by its users.

In languages that expect routines to report errors via status variables, return codes, or thread-local error indicators, the error indications need to be checked after each call. As these frequent checks cost execution time and clutter the code immensely to deal with situations that may occur rarely, programmers are reluctant to apply the scheme systematically and consistently. Failure to check for and handle an arising error condition continues execution as if the error never occurred. In most cases, this continued execution in an ill-defined program state will sooner or later fail, possibly catastrophically.

The raising and handling of exceptions was introduced into languages to address these problems. They bundle the exceptional code in exception handlers, they need not cost execution time if no error is present, and they will not allow the program to continue execution by default when an error occurs, since upon raising the exception, control of execution is automatically transferred to a handler for the exception found on the call stack. The risk and the failure mechanism is that there is no such handler (unless the language enforces restrictions that guarantees its existence), resulting in the termination of the current thread of control. Also, a handler that is found might not be geared to handle the multitude of error situations that are vectored to it. Exception handling is therefore in practice more complex for the programmer than, for example, the use of status parameters. Furthermore, different languages provide exception-handling mechanisms that differ in details of their design, which in turn may lead to misunderstandings by the programmer.

The cause for the failure might be simply laziness or ignorance on the part of the programmer, or, more commonly, a mismatch in the expectations of where fault detection and fault recovery is to be done. Particularly when components meet that employ different fault detection and reporting strategies, the opportunity for mishandling recognized errors increases and creates vulnerabilities.

Another cause of the failure is the scant attention that many library providers pay to describe all error situations that calls on their routines might encounter and report. In this case, the caller cannot possibly react sensibly to all error situations that might arise. As yet another cause, the error information provided when the error occurs may be insufficiently complete to allow recovery from the error.

6.38.4 Applicable language characteristics

Whether supported by the language or not, error reporting and handling is idiomatically present in all languages. Of course, vulnerabilities caused by exceptions require a language that supports exceptions.

6.38.5 Avoiding the vulnerability or mitigating its effects

Given the variety of error handling mechanisms, it is difficult to provide general guidelines. However, dealing with exception handling in some languages can stress the capabilities of static analysis tools and can, in some cases, reduce the effectiveness of their analysis. Inversely, the use of error status variables can lead to confusingly complicated control structures, particularly when recovery is not possible locally. Therefore, for situations where the highest of reliability is required, the decision for or against exception handling deserves careful thought. In any case, exception-handling mechanisms should be reserved for truly unexpected situations and other situations where no local recovery is possible. Situations which are merely unusual, like the end of file condition, should be treated by explicit testing—either prior to the call which might raise the error or immediately afterward. In general, error detection, reporting, correction, and recovery should not be a late opportunistic add-on, but should be an integral part of a system design.

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Checking error return values or auxiliary status variables following a call to a subprogram is mandatory unless it can be demonstrated that the error condition is impossible.
- Equally, exceptions need to be handled by the exception handlers of an enclosing construct as close as possible to the origin of the exception but as far out as necessary to be able to deal with the error.
- For each routine, all error conditions need to be documented and matching error detection and reporting needs to be implemented, providing sufficient information for handling the error situation.
- When execution within a particular context is abandoned due to an exception or error condition, it is important to finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.
- It is often not appropriate to repair an error situation and retry the operation. It is usually a better solution to finalize and terminate the current context and retreat to a context where the fault can be handled completely.
- Error checking provided by the language, the software system, or the hardware should never be disabled in the absence of a conclusive analysis that the error condition is rendered impossible.
- Because of the complexity of error handling, careful review of all error handling mechanisms is appropriate.

- In applications with the highest requirements for reliability, defense-in-depth approaches are often appropriate, for example, checking and handling errors even if thought to be impossible.

6.38.6 Implications for standardization

In future standardization activities, the following items should be considered:

- A standardized set of mechanisms for detecting and treating error conditions should be developed so that all languages to the extent possible could use them. This does not mean that all languages should use the same mechanisms as there should be a variety, but each of the mechanisms should be standardized.

6.39 Termination Strategy [REU]

6.39.1 Description of application vulnerability

Expectations that a system will be dependable are based on the confidence that the system will operate as expected and not fail in normal use. The dependability of a system and its fault tolerance can be measured through the component part's reliability, availability, safety and security. Reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time [IEEE 1990 glossary]. Availability is how timely and reliable the system is to its intended users. Both of these factors matter highly in systems used for safety and security. In spite of the best intentions, systems may encounter a failure, either from internally poorly written software or external forces such as power outages/variations, floods, or other natural disasters. The reaction to a fault can affect the performance of a system and in particular, the safety and security of the system and its users.

When the software does not terminate in the planned mechanism, safety or security is compromised, as failing in an unspecified way interferes with the alternative recovery features. In safety-related systems the results can be catastrophic: for other systems the result can mean failure of the complete system.

6.39.2 Cross reference

JSF AV Rule: 24
MISRA C 2004: 20.11
MISRA C++ 2008: 0-3-2, 15-5-2, 15-5-3, and 18-0-3
CERT C guidelines: ERR04-C, ERR06-C and ENV32-C
Ada Quality and Style Guide: 5.8 and 7.5

6.39.3 Mechanism of failure

The reaction to a fault in a system can depend on the criticality of the part in which the fault originates. When a program consists of several tasks, each task may be critical, or not. If a task is critical, it may or may not be restartable by the rest of the program. Ideally, a task that detects a fault within itself should be able to halt leaving its resources available for use by the rest of the program, halt clearing away its resources, or halt the entire program. The latency of task termination and whether tasks can ignore termination signals should be clearly specified. Having inconsistent reactions to a fault can potentially be a vulnerability.

When a fault is detected, there are many ways in which a system can react. The quickest and most noticeable way is to fail hard, also known as fail fast or fail stop. The reaction to a detected fault is to immediately halt the system. Alternatively, the reaction to a detected fault could be to fail soft. The system would keep working with the faults present, but the performance of the system would be degraded. Systems used in a high availability environment such as telephone switching centers, e-commerce, or other "always available" applications would likely use a fail soft approach. What is actually done in a fail soft approach can vary depending on whether the system is used for safety critical or security critical purposes. For fail-safe systems, such as flight controllers, traffic signals, or medical monitoring systems, there would be no effort to meet normal operational requirements, but rather to limit the damage or danger caused by the fault. A system that fails securely, such as cryptologic systems, would maintain maximum security when a fault is detected, possibly through a denial of service.

6.39.4 Applicable language characteristics

This vulnerability description is intended to be applicable to all languages.

6.39.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- A strategy for fault handling should be decided. Consistency in fault handling should be the same with respect to critically similar parts.
- A multi-tiered approach of fault prevention, fault detection and fault reaction should be used.
- System-defined components that assist in uniformity of fault handling should be used when available. For one example, designing a "runtime constraint handler" (as described in ISO/IEC TR 24731-1 [13]) permits the application to intercept various erroneous situations and perform one consistent response, such as flushing a previous transaction and re-starting at the next one.
- When there are multiple tasks, a fault-handling policy should be specified whereby a task may
    - Halt, and keep its resources available for other tasks (perhaps permitting restarting of the faulting task).
    - Halt, and remove its resources (perhaps to allow other tasks to use the resources so freed, or to allow a recreation of the task).
    - Halt, and signal the rest of the program to likewise halt.

6.39.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing a means to perform fault handling. Terminology and the means should be coordinated with other languages.

6.40 Type-breaking Reinterpretation of Data [AMV]

6.40.1 Description of application vulnerability

In most cases, objects in programs are assigned locations in processor storage to hold their value. If the same storage space is assigned to more than one object—either statically or temporarily—then a change in the value of

one object will have an effect on the value of the other. Furthermore, if the representation of the value of an object is reinterpreted as being the representation of the value of an object with a different type, unexpected results may occur.

6.40.2 Cross reference

JSF AV Rules 153 and183
MISRA 2004: 18.2, 18.3, and 18.4
MISRA C++ 2008: 4-5-1 to 4-5-3, 4-10-1, 4-10-2, and 5-0-3 to 5-0-9
CERT C guidelines: MEM08-C
Ada Quality and Style Guide: 7.6.7 and 7.6.8

6.40.3 Mechanism of failure

Sometimes there is a legitimate need for applications to place different interpretations upon the same stored representation of data. The most fundamental example is a program loader that treats a binary image of a program as data by loading it, and then treats it as a program by invoking it. Most programming languages permit type-breaking reinterpretation of data, however, some offer less error prone alternatives for commonly encountered situations.

Type-breaking reinterpretation of representation presents obstacles to human understanding of the code, the ability of tools to perform effective static analysis, and the ability of code optimizers to do their job.

Examples include:

- Providing alternative mappings of objects into blocks of storage performed either statically (such as Fortran `common`) or dynamically (such as pointers).
- Union types, particularly unions that do not have a discriminant stored as part of the data structure.
- Operations that permit a stored value to be interpreted as a different type (such as treating the representation of a pointer as an integer).

In all of these cases accessing the value of an object may produce an unanticipated result.

A related problem, the aliasing of parameters, occurs in languages that permit call by reference because supposedly distinct parameters might refer to the same storage area, or a parameter and a non-local object might refer to the same storage area. That vulnerability is described in Passing Parameters and Return Values [CSJ].

6.40.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- A programming language that permits multiple interpretations of the same bit pattern.

6.40.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Programmers should avoid reinterpretation performed as a matter of convenience; for example, using an integer pointer to manipulate character string data should be avoided. When type-breaking reinterpretation is necessary, it should be carefully documented in the code. However this vulnerability cannot be completely avoided because some applications view stored data in alternative ways.
- When using union types it is preferable to use discriminated unions. This is a type of a union where a stored value indicates which interpretation is to be placed upon the data. Some languages (such as variant records in Ada) enforce the view of data indicated by the value of the discriminant. If the language does not enforce the interpretation (for example, equivalence in Fortran and union in C and C++), then the code should implement an explicit discriminant and check its value before accessing the data in the union, or use some other mechanism to ensure that correct interpretation is placed upon the data value.
- Operations that reinterpret the same stored value as representing a different type should be avoided. It is easier to avoid such operations when the language clearly identifies them. For example, the name of Ada's `Unchecked_Conversion` function explicitly warns of the problem. A much more difficult situation occurs when pointers are used to achieve type reinterpretation. Some languages perform type-checking of pointers and place restrictions on the ability of pointers to access arbitrary locations in storage. Others permit the free use of pointers. In such cases, code must be carefully reviewed in a search for unintended reinterpretation of stored values. Therefore it is important to explicitly comment the source code where *intended* reinterpretations occur.
- Static analysis tools may be helpful in locating situations where unintended reinterpretation occurs. On the other hand, the presence of reinterpretation greatly complicates static analysis for other problems, so it may be appropriate to segregate intended reinterpretation operations into distinct subprograms.

6.40.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Because the ability to perform reinterpretation is sometimes necessary, but the need for it is rare, programming language designers might consider putting caution labels on operations that permit reinterpretation. For example, the operation in Ada that permits unconstrained reinterpretation is called `Unchecked_Conversion`.
- Because of the difficulties with undiscriminated unions, programming language designers might consider offering union types that include distinct discriminants with appropriate enforcement of access to objects.

6.41 Memory Leak [XYL]

6.41.1 Description of application vulnerability

A memory leak occurs when software does not release allocated memory after it ceases to be used. Repeated occurrences of a memory leak can consume considerable amounts of available memory. A memory leak can be exploited by attackers to generate denial-of-service by causing the program to execute repeatedly a sequence that triggers the leak. Moreover, a memory leak can cause any long-running critical program to shutdown prematurely.

6.41.2 Cross reference

CWE:
401. Failure to Release Memory Before Removing Last Reference (aka 'Memory Leak')
JSF AV Rule: 206
MISRA C 2004: 20.4
CERT C guidelines: MEM00-C and MEM31-C
Ada Quality and Style Guide: 5.4.5, 5.9.2, and 7.3.3

6.41.3 Mechanism of failure

As a process or system runs, any memory taken from dynamic memory and not returned or reclaimed (by the runtime system or a garbage collector) after it ceases to be used, may result in future memory allocation requests failing for lack of free space. Alternatively, memory claimed and returned can cause the heap to fragment, which will eventually result in an inability to take the necessary size storage. Either condition will result in a memory exhaustion exception, and program termination or a system crash.

If an attacker can determine the cause of an existing memory leak, the attacker may be able to cause the application to leak quickly and therefore cause the application to crash.

6.41.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that support mechanisms to dynamically allocate memory and reclaim memory under program control.

6.41.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use of garbage collectors that reclaim memory that will never be used by the application again. Some garbage collectors are part of the language while others are add-ons.
- Allocating and freeing memory in different modules and levels of abstraction may make it difficult for developers to match requests to free storage with the appropriate storage allocation request. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to memory leaks. To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.
- Storage pools are a specialized memory mechanism where all of the memory associated with a class of objects is allocated from a specific bounded region. When used with strong typing one can ensure a strong relationship between pointers and the space accessed such that storage exhaustion in one pool does not affect the code operating on other memory.
- Memory leaks can be eliminated by avoiding the use of dynamically allocated storage entirely, or by doing initial allocation exclusively and never allocating once the main execution commences. For safety-critical systems and long running systems, the use of dynamic memory is almost always prohibited, or restricted to the initialization phase of execution.

- Use static analysis, which can sometimes detect when allocated storage is no longer used and has not been freed.

6.41.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages can provide syntax and semantics to guarantee program-wide that dynamic memory is not used (such as the configuration `pragmas` feature offered by some programming languages).
- Languages can document or specify that implementations must document choices for dynamic memory management algorithms, to hope designers decide on appropriate usage patterns and recovery techniques as necessary

6.42 Templates and Generics [SYM]

6.42.1 Description of application vulnerability

Many languages provide a mechanism that allows objects and/or functions to be defined parameterized by type and then instantiated for specific types. In C++ and related languages, these are referred to as “templates”, and in Ada and Java, “generics”. To avoid having to keep writing ‘templates/generics’, in this clause these will simply be referred to collectively as generics.

Used well, generics can make code clearer, more predictable and easier to maintain. Used badly, they can have the reverse effect, making code difficult to review and maintain, leading to the possibility of program error.

6.42.2 Cross reference

JSF AV Rules: 101, 102, 103, 104, and 105
MISRA C++ 2008: 14-6-1, 14-6-2, 14-7-1 to 14-7-3, 14-8-1, and 14-8-2
Ada Quality and Style Guide: 8.3.1 through 8.3.8, and 8.4.2

6.42.3 Mechanism of failure

The value of generics comes from having a single piece of code that supports some behaviour in a type independent manner. This simplifies development and maintenance of the code. It should also assist in the understanding of the code during review and maintenance, by providing the same behaviour for all types with which it is instantiated.

Problems arise when the use of a generic actually makes the code harder to understand during review and maintenance, by not providing consistent behaviour.

In most cases, the generic definition will have to make assumptions about the types it can legally be instantiated with. For example, a sort function requires that the elements to be sorted can be copied and compared. If these assumptions are not met, the result is likely to be a compiler error. For example if the sort function is instantiated with a user defined type that doesn’t have a relational operator. Where ‘misuse’ of a generic leads to a compiler error, this can be regarded as a development issue, and not a software vulnerability.

Confusion, and hence potential vulnerability, can arise where the instantiated code is apparently invalid, but doesn't result in a compiler error. For example, a generic class defines a set of members, a subset of which rely on a particular property of the instantiation type (such as a generic container class with a sort member function, only the sort function relies on the instantiating type having a defined relational operator). In some languages, such as C++, if the generic is instantiated with a type that doesn't meet all the requirements but the program never subsequently makes use of the subset of members that rely on the property of the instantiating type, the code will compile and execute (for example, the generic container is instantiated with a user defined class that doesn't define a relational operator, but the program never calls the sort member of this instantiation). When the code is reviewed the generic class will appear to reference a member of the instantiating type that doesn't exist.

The problem as described in the two prior paragraphs can be reduced by a language feature (such as the *concepts* language feature being designed by the C++ committee).

Similar confusion can arise if the language permits specific elements of a generic to be explicitly defined, rather than using the common code, so that behaviour is not consistent for all instantiations. For example, for the same generic container class, the sort member normally sorts the elements of the container into ascending order. In languages such as C++, a 'special case' can be created for the instantiation of the generic with a particular type. For example, the sort member for a 'float' container may be explicitly defined to provide different behaviour, say sorting the elements into descending order. Specialization that doesn't affect the apparent behaviour of the instantiation is not an issue. Again, for C++, there are some irregularities in the semantics of arrays and pointers that can lead to the generic having different behaviour for different, but apparently very similar, types. In such cases, specialization can be used to enforce consistent behaviour.

6.42.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- Languages that permit definitions of objects or functions to be parameterized by type, for later instantiation with specific types, such as:
    - Templates in C++
    - Generics in Ada, Java.

6.42.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Document the properties of an instantiating type necessary for a generic to be valid.
- If an instantiating type has the required properties, the whole of the generic should be ensured to be valid, whether actually used in the program or not.
- Preferably avoid, but at least carefully document, any 'special cases' where a generic is instantiated with a specific type doesn't behave as it does for other types.

6.42.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specifiers should standardize on a common, uniform terminology to describe generics/templates so that programmers experienced in one language can reliably learn and refer to the type system of another language that has the same concept, but with a different name.
- Language specifiers should design generics in such a way that any attempt to instantiate a generic with constructs that do not provide the required capabilities results in a compile-time error.
- Language specifiers should provide an assertion mechanism for checking properties at run-time, for those properties that cannot be checked at compile time. It should be possible to inhibit assertion checking if efficiency is a concern.

6.43 Inheritance [RIP]

6.43.1 Description of application vulnerability

Inheritance, the ability to create enhanced and/or restricted object classes based on existing object classes can introduce a number of vulnerabilities, both inadvertent and malicious. Because Inheritance allows the overriding of methods of the parent class and because object oriented systems are designed to separate and encapsulate code and data, it can be difficult to determine where in the hierarchy an invoked method is actually defined. Also, since an overriding method does not need to call the method in the parent class that has been overridden, essential initialization and manipulation of class data may be bypassed. This can be especially dangerous during constructor and destructor methods.

Languages that allow multiple inheritance add additional complexities to the resolution of method invocations. Different object brokerage systems may resolve the method identity to different classes, based on how the inheritance tree is traversed.

6.43.2 Cross reference

JSF AV Rules: 86 to 97
MISRA C++ 2008: 0-1-12, 8-3-1, 10-1-1 to 10-1-3, and 10-3-1 to 10-3-3
Ada Quality and Style Guide: 9 (complete clause)

6.43.3 Mechanism of failure

The use of inheritance can lead to an exploitable application vulnerability or negatively impact system safety in several ways:

- Execution of malicious redefinitions, this can occur through the insertion of a class into the class hierarchy that overrides commonly called methods in the parent classes.
- Accidental redefinition, where a method is defined that inadvertently overrides a method that has already been defined in a parent class.
- Accidental failure of redefinition, when a method is incorrectly named or the parameters are not defined properly, and thus does not override a method in a parent class.
- Breaking of class invariants, this can be caused by redefining methods that initialize or validate class data without including that initialization or validation in the overriding methods.

These vulnerabilities can increase dramatically as the complexity of the hierarchy increases, especially in the use of multiple inheritance.

6.43.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that allow single and multiple inheritances.

6.43.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Avoid the use of multiple inheritance whenever possible.
- Provide complete documentation of all encapsulated data, and how each method affects that data for each object in the hierarchy.
- Inherit only from trusted sources, and, whenever possible, check the version of the parent classes during compilation and/or initialization.
- Provide a method that provides versioning information for each class.

6.43.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language specification should include the definition of a common versioning method.
- Compilers should provide an option to report the class in which a resolved method resides.
- Runtime environments should provide a trace of all runtime method resolutions.

6.44 Extra Intrinsics [LRM]

6.44.1 Description of application vulnerability

Most languages define intrinsic procedures, which are easily available, or always "simply available", to any translation unit. If a translator extends the set of intrinsics beyond those defined by the standard, and the standard specifies that intrinsics are selected before procedures of the same signature defined by the application, a different procedure may be unexpectedly used when switching between translators.

6.44.2 Cross reference

[None]

6.44.3 Mechanism of failure

Most standard programming languages define a set of intrinsic procedures which may be used in any application. Some language standards allow a translator to extend this set of intrinsic procedures. Some language standards specify that intrinsic procedures are selected ahead of an application procedure of the same signature. This may cause a different procedure to be used when switching between translators.

For example, most languages provide a routine to calculate the square root of a number, usually named `sqrt()`. If a translator also provided, as an extension, a cube root routine, say named `cbrt()`, that extension may override an application defined procedure of the same signature. If the two different `cbrt()` routines chose different branch cuts when applied to complex arguments, the application could unpredictably go wrong.

If the language standard specifies that application defined procedures are selected ahead of intrinsic procedures of the same signature, the use of the wrong procedure may mask a linking error.

6.44.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Any language where translators may extend the set of intrinsic procedures and where intrinsic procedures are selected ahead of application defined (or external library defined) procedures of the same signature.

6.44.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use whatever language features are available to mark a procedure as language defined or application defined.
- Be aware of the documentation for every translator in use and avoid using procedure signatures matching those defined by the translator as extending the standard set.

6.44.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Clearly state whether translators can extend the set of intrinsic procedures or not.
- Clearly state what the precedence is for resolving collisions.
- Clearly provide ways to mark a procedure signature as being the intrinsic or an application provided procedure.
- Require that a diagnostic is issued when an application procedure matches the signature of an intrinsic procedure.

6.45 Argument Passing to Library Functions [TRJ]

6.45.1 Description of application vulnerability

Libraries that supply objects or functions are in most cases not required to check the validity of parameters passed to them. In those cases where parameter validation is required there might not be adequate parameter validation.

6.45.2 Cross reference

CWE:

114. Process Control
JSF AV Rules 16, 18, 19, 20, 21, 22, 23, 24, and 25
MISRA C 2004: 20.2, 20.3, 20.4, 20.6, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12
MISRA C++ 2008: 17-0-1, 17-0-5, 18-0-2, 18-0-3, 18-0-4, 18-2-1, 18-7-1 and 27-0-1
CERT C guidelines: INT03-C and STR07-C

6.45.3 Mechanism of failure

When calling a library, either the calling function or the library may make assumptions about parameters. For example, it may be assumed by a library that a parameter is non-zero so division by that parameter is performed without checking the value. Sometimes some validation is performed by the calling function, but the library may use the parameters in ways that were unanticipated by the calling function resulting in a potential vulnerability. Even when libraries do validate parameters, their response to an invalid parameter is usually undefined and can cause unanticipated results.

6.45.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages providing or using libraries that do not validate the parameters accepted by functions, methods and objects.

6.45.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Libraries should be defined so that as many parameters as possible are validated.
- Libraries should be defined to validate any values passed to the library before the value is used.
- Develop wrappers around library functions that check the parameters before calling the function.
- Demonstrate statically that the parameters are never invalid.
- Use only libraries known to have been developed with consistent and validated interface requirements.

It is noted that several approaches can be taken, some work best if used in conjunction with each other.

6.45.6 Implications for standardization

In future standardization activities, the following items should be considered:

- All languages that define a support library should consider removing most if not all cases of undefined behaviour from the library clauses.
- Languages should define libraries that provide the capability to validate parameters during compilation, during execution or by static analysis.

6.46 Inter-language Calling [DJS]

6.46.1 Description of application vulnerability

When an application is developed using more than one programming language, complications arise. The calling conventions, data layout, error handing and return conventions all differ between languages; if these are not addressed correctly, stack overflow/underflow, data corruption, and memory corruption are possible.

In multi-language development environments it is also difficult to reuse data structures and object code across the languages.

6.46.2 Cross reference

[None**]**

6.46.3 Mechanism of failure

When calling a function that has been developed using a language different from the calling language, the call convention and the return convention used must be taken into account. If these conventions are not handled correctly, there is a good chance the calling stack will be corrupted, see [OTR]. The call convention covers how the language invokes the call, see [CJS], and how the parameters are handled.

Many languages restrict the length of identifiers, the type of characters that can be used as the first character, and the case of the characters used. All of these need to be taken into account when invoking a routine written in a language other than the calling language. Otherwise the identifiers might bind in a manner different than intended.

Character and aggregate data types require special treatment in a multi-language development environment. The data layout of all languages that are to be used must be taken into consideration; this includes padding and alignment. If these data types are not handled correctly, the data could be corrupted, the memory could be corrupted, or both may become corrupt. This can happen by writing/reading past either end of the data structure, see [HCB]. For example, a Pascal `STRING` data type

```
VAR str: STRING(10);
```

corresponds to a C structure

```
struct {
  int length;
  char str [10];
};
```

and **not** to the C structure

```
char str [10]
```

where `length` contains the actual length of `STRING`. The second C construct is implemented with a physical length that is different from physical length of the Pascal `STRING` and assumes a null terminator.

Most numeric data types have counterparts across languages, but again the layout should be understood, and only those types that match the languages should be used. For example, in some implementations of C++ a

```
signed char
```

would match a Fortran

```
integer(1)
```

and would match a Pascal

```
PACKED -128..127
```

These correspondences can be implementation-defined and should be verified.

6.46.4 Applicable language characteristics

The vulnerability is applicable to languages with the following characteristics:

- All high level programming languages and low level programming languages are susceptible to this vulnerability when used in a multi-language development environment.

6.46.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use the inter-language methods and syntax specified by the applicable language standard(s). For example, Fortran and Ada specify how to call C functions.
- Understand the calling conventions of all languages used.
- For items comprising the inter-language interface:
    - Understand the data layout of all data types used.
    - Understand the return conventions of all languages used.
    - Ensure that the language in which error check occurs is the one that handles the error.
    - Avoid assuming that the language makes a distinction between upper case and lower case letters in identifiers.
    - Avoid using a special character as the first character in identifiers.
    - Avoid using long identifier names.

6.46.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Standards committees should consider developing standard provisions for inter-language calling with languages most often used with their programming language.

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

6.47.1 Description of application vulnerability

Code that is dynamically linked may be different from the code that was tested. This may be the result of replacing a library with another of the same name or by altering an environment variable such as `LD_LIBRARY_PATH` on UNIX platforms so that a different directory is searched for the library file. Executing code that is different than that which was tested may lead to unanticipated errors or intentional malicious activity.

On some platforms, and in some languages, instructions can modify other instructions in the code space. Historically self-modifying code was needed for software that was required to run on a platform with very limited memory. It is now primarily used (or misused) to hide functionality of software and make it more difficult to reverse engineer or for specialty applications such as graphics where the algorithm is tuned at runtime to give better performance. Self-modifying code can be difficult to write correctly and even more difficult to test and maintain correctly leading to unanticipated errors.

6.47.2 Cross reference

JSF AV Rule: 2

6.47.3 Mechanism of failure

Through the alteration of a library file or environment variable, the code that is dynamically linked may be different from the code which was tested resulting in different functionality.

On some platforms, a pointer-to-data can erroneously be given an address value that designates a location in the instruction space. If subsequently a modification is made through that pointer, then an unanticipated behaviour can result.

6.47.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that allow a pointer-to-data to be assigned an address value that designates a location in the instruction space.
- Languages that allow execution of code that exists in data space.
- Languages that permit the use of dynamically linked or shared libraries.
- Languages that execute on an OS that permits program memory to be both writable and executable.

6.47.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Verify that the dynamically linked or shared code being used is the same as that which was tested.
- Do not write self-modifying code except in extremely rare instances. Most software applications should never have a requirement for self-modifying code.

- In those extremely rare instances where its use is justified, self-modifying code should be very limited and heavily documented.

6.47.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should consider providing a means so that a program can either automatically or manually check that the digital signature of a library matches the one in the compile/test environment.

6.48 Library Signature [NSQ]

6.48.1 Description of application vulnerability

Programs written in modern languages may use libraries written in other languages than the program implementation language. If the library is large, the effort of adding signatures for all of the functions use by hand may be tedious and error-prone. Portable cross-language signatures will require detailed understanding of both languages, which a programmer may lack.

Integrating two or more programming languages into a single executable relies upon knowing how to interface the function calls, argument list and global data structures so the symbols match in the object code during linking.

Byte alignment can be a source of data corruption if memory boundaries between the programming languages are different. Each language may also align structure data differently.

6.48.2 Cross reference

MISRA C 2004: 1.3
MISRA C++ 2008: 1-0-2

6.48.3 Mechanism of failure

When the library and the application in which it is to be used are written in different languages, the specification of signatures is complicated by inter-language issues.

As used in this vulnerability description, the term library includes the interface to the operating system, which may be specified only for the language used to code the operating system itself. In this case, any program written in any other language faces the inter-language interoperability issue of creating a fully-functional signature.

When the application language and the library language are different, then the ability to specify signatures according to either standard may not exist, or be very difficult. Thus, a translator-by-translator solution may be needed, which maximizes the probability of incorrect signatures (since the solution must be recreated for each translator pair). Incorrect signatures may or may not be caught during the linking phase.

6.48.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that do not specify how to describe signatures for subprograms written in other languages.

6.48.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use tools to create the signatures.
- Avoid using translator options or language features to reference library subprograms without proper signatures.

6.48.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Provide correct linkage even in the absence of correctly specified procedure signatures. (Note that this may be very difficult where the original source code is unavailable.)
- Provide specified means to describe the signatures of subprograms.

6.49 Unanticipated Exceptions from Library Routines [HJW]

6.49.1 Description of application vulnerability

A library in this context is taken to mean a set of software routines produced outside the control of the main application developer, usually by a third party, and where the application developer may not have access to the source. In such circumstances the application developer has limited knowledge of the library functions, other than from their behavioural interface.

Whilst the use of libraries can present a number of vulnerabilities, the focus of this vulnerability is any undesirable behaviour that a library routine may exhibit, in particular the generation of unexpected exceptions.

6.49.2 Cross reference

JSF AV Rule: 208
MISRA C 2004: 3.6, 20.3
MISRA C++ 2008: 15-3-1, 15-3-2, 17-0-4
Ada Quality and Style Guide: 5.8 and 7.5

6.49.3 Mechanism of failure

In some languages, unhandled exceptions lead to implementation-defined behaviour. This can include immediate termination, without for example, releasing previously allocated resources. If a library routine raises an unanticipated exception, this undesirable behaviour may result.

It should be noted that the considerations of [OYB], Ignored Error Status and Unhandled Exceptions, are also relevant here.

6.49.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that can link previously developed library code (where the developer and compiler don't have access to the library source).
- Languages that permit exceptions to be thrown but do not require handlers for them.

6.49.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- All library calls should be wrapped within a 'catch-all' exception handler (if the language supports such a construct), so that any unanticipated exceptions can be caught and handled appropriately. This wrapping may be done for each library function call or for the entire behaviour of the program, for example, having the exception handler in main for C++. However, note that the latter isn't a complete solution, as static objects are constructed before main is entered and are destroyed after it has been exited. Consequently, MISRA C++ [16] bars class constructors and destructors from throwing exceptions (unless handled locally).
- An alternative approach would be to use only library routines for which all possible exceptions are specified.

6.49.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages that provide exceptions should provide a mechanism for catching all possible exceptions (for example, a 'catch-all' handler). The behaviour of the program when encountering an unhandled exception should be fully defined.
- Languages should provide a mechanism to determine which exceptions might be thrown by a called library routine.

6.50 Pre-processor Directives [NMP]

6.50.1 Description of application vulnerability

Pre-processor replacements happen before any source code syntax check, therefore there is no type checking – this is especially important in function-like macro parameters.

If great care is not taken in the writing of macros, the expanded macro can have an unexpected meaning. In many cases if explicit delimiters are not added around the macro text and around all macro arguments within the macro text, unexpected expansion is the result.

Source code that relies heavily on complicated pre-processor directives may result in obscure and hard to maintain code since the syntax they expect may be different from the expressions programmers regularly expect in a given programming language.

6.50.2 Cross reference

Holzmann-8
JSF AV Rules: 26, 27, 28, 29, 30, 31, and 32
MISRA C 2004: 19.6, 19.7, 19.8, and 19.9
MISRA C++ 2008: 16-0-3, 16-0-4, and 16-0-5
CERT C guidelines: PRE01-C, PRE02-C, PRE10-C, and PRE31-C

6.50.3 Mechanism of failure

Readability and maintainability may be greatly decreased if pre-processing directives are used instead of language features.

While static analysis can identify many problems early; heavy use of the pre-processor can limit the effectiveness of many static analysis tools, which typically work on the pre-processed source code.

In many cases where complicated macros are used, the program does not do what is intended. For example:

define a macro as follows,

```
#define CD(x, y) (x + y - 1) / y
```

whose purpose is to divide. Then suppose it is used as follows

```
a = CD (b & c, sizeof (int));
```

which expands into

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which most times will not do what is intended. Defining the macro as

```
#define CD(x, y) ((x) + (y) - 1) / (y)
```

will provide the desired result.

6.50.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that have a lexical-level pre-processor.
- Languages that allow unintended groupings of arithmetic statements.
- Languages that allow cascading macros.
- Languages that allow duplication of side effects.
- Languages that allow macros that reference themselves.
- Languages that allow nested macro calls.
- Languages that allow complicated macros.

6.50.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Where it is possible to achieve the desired functionality without the use of pre-processor directives, this should be done in preference to the use of pre-processor directives.

6.50.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Standards should reduce or eliminate dependence on lexical-level pre-processors for essential functionality (such as conditional compilation).
- Standards should consider providing capabilities to inline functions and procedure calls, to reduce the need for pre-processor macros.

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

6.51.1 Description of application vulnerability

Some languages include the provision for runtime checking to prevent vulnerabilities to arise. Canonical examples are bounds or length checks on array operations or null-value checks upon dereferencing pointers or references. In most cases, the reaction to a failed check is the raising of a language-defined exception.

As run-time checking requires execution time and as some project guidelines exclude the use of exceptions, languages may define a way to optionally suppress such checking for regions of the code or for the entire program. Analogously, compiler options may be used to achieve this effect.

6.51.2 Cross reference

[None]

6.51.3 Mechanism of Failure

Vulnerabilities that could have been prevented by the run-time checks are undetected, resulting in memory corruption, propagation of incorrect values or unintended execution paths.

6.51.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that define runtime checks to prevent certain vulnerabilities and
- Languages that allow the above checks to be suppressed,
- Languages or compilers that suppress checking by default, or whose compilers or interpreters provide options to omit the above checks

6.51.5 Avoiding the vulnerability

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Do not suppress checks at all or restrict the suppression of checks to regions of the code that have been proved to be performance-critical.
- If the default behaviour of the compiler or the language is to suppress checks, then enable them.
- Where checks are suppressed, verify that the suppressed checks could not have failed.
- Clearly identify code sections where checks are suppressed.
- Do not assume that checks in code verified to satisfy all checks could not fail nevertheless due to hardware faults.

6.51.6 Implications for standardization

[None]

6.52 Provision of Inherently Unsafe Operations [SKL]

6.52.1 Description of application vulnerability

Languages define semantic rules to be obeyed by conforming programs. Compilers enforce these rules and diagnose violating programs.

A canonical example are the rules of type checking, intended among other reasons to prevent semantically incorrect assignments, such as characters to pointers, meter to feet, euro to dollar, real numbers to booleans, or complex numbers to two-dimensional coordinates.

Occasionally there arises a need to step outside the rules of the type model to achieve needed functionally. One such situation is the casting of memory as part of the implementation of a heap allocator to the type of object for which the memory is allocated. A type-safe assignment is impossible for this functionality. Thus, a capability for unchecked “type casting” between arbitrary types to interpret the bits in a different fashion is a necessary but inherently unsafe operation, without which the type-safe allocator cannot be programmed.

Another example is the provision of operations known to be inherently unsafe, such as the deallocation of heap memory without prevention of dangling references.

A third example is any interfacing with another language, since the checks ensuring type-safeness rarely extend across language boundaries.

These inherently unsafe operations constitute a vulnerability, since they can (and will) be used by programmers in situations where their use is neither necessary nor appropriate.

The vulnerability is eminently exploitable to violate program security.

6.52.2 Cross reference

[None]

6.52.3 Mechanism of Failure

The use of inherently unsafe operations or the suppression of checking circumvents the features that are normally applied to ensure safe execution. Control flow, data values, and memory accesses can be corrupted as a consequence. See the respective vulnerabilities resulting from such corruption.

6.52.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that allow compile-time checks for the prevention of vulnerabilities to be suppressed by compiler or interpreter options or by language constructs, or
- Languages that provide inherently unsafe operations

6.52.5 Avoiding the vulnerability

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Restrict the suppression of compile-time checks to where the suppression is functionally essential.
- Use inherently unsafe operations only when they are functionally essential.
- Clearly identify program code that suppresses checks or uses unsafe operations. This permits the focusing of review effort to examine whether the function could be performed in a safer manner.

6.53 Obscure Language Features [BRS]

6.53.1 Description of application vulnerability

Every programming language has features that are obscure, difficult to understand or difficult to use correctly. The problem is compounded if a software design must be reviewed by people who may not be language experts, such as, hardware engineers, human-factors engineers, or safety officers. Even if the design and code are initially correct, maintainers of the software may not fully understand the intent. The consequences of the problem are more severe if the software is to be used in trusted applications, such as safety or mission critical ones.

Misunderstood language features or misunderstood code sequences can lead to application vulnerabilities in development or in maintenance.

6.53.2 Cross reference

JSF AV Rules: 84, 86, 88, and 97
MISRA C 2004: 3.2, 10.2, 13.1, 17.5, 20.6-20.12, and 12.10
MISRA C++ 2008: 0-2-1, 2-3-1, and 12-1-1
CERT C guidelines: FIO03-C, MSC05-C, MSC30-C, and MSC31-C.
ISO/IEC TR 15942:2000: 5.4.2, 5.6.2 and 5.9.3

6.53.3 Mechanism of failure

The use of obscure language features can lead to an application vulnerability in several ways:

- The original programmer may misunderstand the correct usage of the feature and could utilize it incorrectly in the design or code it incorrectly.
- Reviewers of the design and code may misunderstand the intent or the usage and overlook problems.
- Maintainers of the code cannot fully understand the intent or the usage and could introduce problems during maintenance.

6.53.4 Applicable language characteristics

This vulnerability description is intended to be applicable to any language.

6.53.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Individual programmers should avoid the use of language features that are obscure or difficult to use, especially in combination with other difficult language features. Organizations should adopt coding standards that discourage use of such features or show how to use them correctly.
- Organizations developing software with critically important requirements should adopt a mechanism to monitor which language features are correlated with failures during the development process and during deployment.
- Organizations should adopt or develop stereotypical idioms for the use of difficult language features, codify them in organizational standards, and enforce them via review processes.
- Avoid the use of complicated features of a language.
- Avoid the use of rarely used constructs that could be difficult for entry-level maintenance personnel to understand.
- Static analysis can be used to find incorrect usage of some language features.

It should be noted that consistency in coding is desirable for each of review and maintenance. Therefore, the desirability of the particular alternatives chosen for inclusion in a coding standard does not need to be empirically proven.

6.53.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language designers should consider removing or deprecating obscure, difficult to understand, or difficult to use features.
- Language designers should provide language directives that optionally disable obscure language features.

6.54 Unspecified Behaviour [BQF]

6.54.1 Description of application vulnerability

The external behaviour of a program whose source code contains one or more instances of constructs having unspecified behaviour may not be fully predictable when the source code is (re)compiled or (re)linked.

6.54.2 Cross reference

JSF AV Rules: 17-25
MISRA C 2004: 1.3, 1.5, 3.1 3.3, 3.4, 17.3, 1.2, 5.1, 18.2, 19.2, and 19.14
MISRA C++ 2008: 5-0-1, 5-2-6, 7-2-1, and 16-3-1
CERT C guidelines: MSC15-C
See: Undefined Behaviour [EWF] and Implementation-defined Behaviour [FAB].

6.54.3 Mechanism of failure

Language specifications do not always uniquely define the behaviour of a construct. When an instance of a construct that is not uniquely defined is encountered (this might be at any of compile, link, or run time) implementations are permitted to choose from the set of behaviours allowed by the language specification. The term 'unspecified behaviour' is sometimes applied to such behaviours, (language specific guidelines need to analyze and document the terms used by their respective language).

A developer may use a construct in a way that depends on a subset of the possible behaviours occurring. The behaviour of a program containing such a usage is dependent on the translator used to build it always selecting the 'expected' behaviour.

Many language constructs may have unspecified behaviour and unconditionally recommending against any use of these constructs may be impractical. For instance, in many languages the order of evaluation of the operands appearing on the left- and right-hand side of an assignment is unspecified, but in most cases the set of possible behaviours always produce the same result.

The appearance of unspecified behaviour in a language specification is recognition by the language designers that in some cases flexibility is needed by software developers and provides a worthwhile benefit for language translators; this usage is not a defect in the language.

The important characteristic is not the internal behaviour exhibited by a construct (such as the sequence of machine code generated by a translator) but its external behaviour (that is, the one visible to a user of a program). If the set of possible unspecified behaviours permitted for a specific use of a construct all produce the same external effect when the program containing them is executed, then rebuilding the program cannot result in a change of behaviour for that specific usage of the construct.

For instance, while the following assignment statement contains unspecified behaviour in many languages (that is, it is possible to evaluate either the `A` or `B` operand first, followed by the other operand):

```
A = B;
```

in most cases the order in which `A` and `B` are evaluated does not affect the external behaviour of a program containing this statement.

6.54.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- Languages whose specification allows a finite set of more than one behaviour for how a translator handles some construct, where two or more of the behaviours can result in differences in external program behaviour.

6.54.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use language constructs that have specified behaviour.
- Ensure that a specific use of a construct having unspecified behaviour produces a result that is the same for all of the possible behaviours permitted by the language specification.
- When developing coding guidelines for a specific language all constructs that have unspecified behaviour should be documented and for each construct the situations where the set of possible behaviours can vary should be enumerated.

6.54.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should minimize the amount of unspecified behaviours, minimize the number of possible behaviours for any given "unspecified" choice, and document what might be the difference in external effect associated with different choices.

6.55 Undefined Behaviour [EWF]

6.55.1 Description of application vulnerability

The external behaviour of a program containing an instance of a construct having undefined behaviour, as defined by the language specification, is not predictable.

6.55.2 Cross reference

JSF AV Rules: 17-25
MISRA C 2004: 1.3, 1.5, 3.1, 3.3, 3.4, 17.3, 1.2, 5.1, 18.2, 19.2, and 19.14
MISRA C++ 2008: 2-13-1, 5-2-2, 16-2-4, and 16-2-5
CERT C guidelines: MSC15-C
See: Unspecified Behaviour [BQF] and Implementation-defined Behaviour [FAB].

6.55.3 Mechanism of failure

Language specifications may categorize the behaviour of a language construct as undefined rather than as a semantic violation (that is, an erroneous use of the language) because of the potentially high implementation cost of detecting and diagnosing all occurrences of it. In this case no specific behaviour is required and the translator or runtime system is at liberty to do anything it pleases (which may include issuing a diagnostic).

The behaviour of a program built from successfully translated source code containing a construct having undefined behaviour is not predictable. For example, in some languages the value of a variable is undefined before it is initialized.

6.55.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- Languages that do not fully define the extent to which the use of a particular construct is a violation of the language specification.
- Languages that do not fully define the behaviour of constructs during compile, link and program execution.

6.55.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Ensuring that undefined language constructs are not used.
- Ensuring that a use of a construct having undefined behaviour does not operate within the domain in which the behaviour is undefined. When it is not possible to completely verify the domain of operation during translation a runtime check may need to be performed.
- When developing coding guidelines for a specific language all constructs that have undefined behaviour should be documented. The items on this list might be classified by the extent to which the behaviour is likely to have some critical impact on the external behaviour of a program (the criticality may vary between different implementations, for example, whether conversion between object and function pointers has well defined behaviour).

6.55.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Language designers should minimize the amount of undefined behaviour to the extent possible and practical.
- Language designers should enumerate all the cases of undefined behaviour.
- Language designers should provide mechanisms that permit the disabling or diagnosing of constructs that may produce undefined behaviour.

6.56 Implementation-defined Behaviour [FAB]

6.56.1 Description of application vulnerability

Some constructs in programming languages are not fully defined (see Unspecified Behaviour [BQF]) and thus leave compiler implementations to decide how the construct will operate. The behaviour of a program, whose source code contains one or more instances of constructs having implementation-defined behaviour, can change when the source code is recompiled or relinked.

6.56.2 Cross reference

JSF AV Rules: 17-25
MISRA C 2004: 1.3, 1.5, 3.1 3.3, 3.4, 17.3, 1.2, 5.1, 18.2, 19.2, and 19.14
MISRA C++ 2008: 5-2-9, 5-3-3, 7-3-2, and 9-5-1
CERT C guidelines: MSC15-C
ISO/IEC TR 15942:2000: 5.9
Ada Quality and Style Guide: 7.1.5 and 7.1.6
See: Unspecified Behaviour [BQF] and Undefined Behaviour [EWF].

6.56.3 Mechanism of failure

Language specifications do not always uniquely define the behaviour of a construct. When an instance of a construct that is not uniquely defined is encountered (this might be at any of translation, link-time, or program execution) implementations are permitted to choose from a set of behaviours. The only difference from unspecified behaviour is that implementations are required to document how they behave.

A developer may use a construct in a way that depends on a particular implementation-defined behaviour occurring. The behaviour of a program containing such a usage is dependent on the translator used to build it always selecting the 'expected' behaviour.

Some implementations provide a mechanism for changing an implementation's implementation-defined behaviour (for example, use of `pragmas` in source code). Use of such a change mechanism creates the potential for additional human error in that a developer may be unaware that a change of behaviour was requested earlier in the source code and may write code that depends on the implementation-defined behaviour that occurred prior to that explicit change of behaviour.

Many language constructs may have implementation-defined behaviour and unconditionally recommending against any use of these constructs may be completely impractical. For instance, in many languages the number of significant characters in an identifier is implementation-defined. Developers need to choose a minimum number of characters and require that only translators supporting at least that number, N, of characters be used.

The appearance of implementation-defined behaviour in a language specification is recognition by the language designers that in some cases implementation flexibility provides a worthwhile benefit for language translators; this usage is not a defect in the language.

6.56.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- Languages whose specification allows some variation in how a translator handles some construct, where reliance on one form of this variation can result in differences in external program behaviour.
- Language implementations may not be required to provide a mechanism for controlling implementation-defined behaviour.

6.55.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Document the set of implementation-defined features an application depends upon, so that upon a change of translator, development tools, or target configuration it can be ensured that those dependencies are still met.
- Ensure that a specific use of a construct having implementation-defined behaviour produces an external behaviour that is the same for all of the possible behaviours permitted by the language specification.
- Only use a language implementation whose implementation-defined behaviours are within a known subset of implementation-defined behaviours. The known subset should be chosen so that the 'same external behaviour' condition described above is met.
- Create highly visible documentation (perhaps at the start of a source file) that the default implementation-defined behaviour is changed within the current file.
- When developing coding guidelines for a specific language all constructs that have implementation-defined behaviour shall be documented and for each construct, the situations where the set of possible behaviours can vary shall be enumerated.
- When applying this guideline on a project the functionality provided by and for changing its implementation-defined behaviour shall be documented.
- Verify code behaviour using at least two different compilers with two different technologies.

6.56.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Portability guidelines for a specific language should provide a list of common implementation-defined behaviours.
- Language specifiers should enumerate all the cases of implementation-defined behaviour.
- Language designers should provide language directives that optionally disable obscure language features.

6.57 Deprecated Language Features [MEM]

6.57.1 Description of application vulnerability

All code should conform to the current standard for the respective language. In reality though, a language standard may change during the creation of a software system or suitable compilers and development environments may not be available for the new standard for some period of time after the standard is published. To smooth the process of evolution, features that are no longer needed or which serve as the root cause of or contributing factor for safety or security problems are often deprecated to temporarily allow their continued use but to indicate that those features may be removed in the future. The deprecation of a feature is a strong indication that it should not be used. Other features, although not formally deprecated, are rarely used and there exist other more common ways of expressing the same function. Use of these rarely used features can lead to problems when others are assigned the task of debugging or modifying the code containing those features.

6.57.2 Cross reference

JSF AV Rules: 8 and 11
MISRA C 2004: 1.1, 4.2, and 20.10
MISRA C++ 2008: 1-0-1, 2-3-1, 2-5-1, 2-7-1, 5-2-4, and 18-0-2
Ada Quality and Style Guide: 7.1.1

6.57.3 Mechanism of failure

Most languages evolve over time. Sometimes new features are added making other features extraneous. Languages may have features that are frequently the basis for security or safety problems. The deprecation of these features indicates that there is a better way of accomplishing the desired functionality. However, there is always a time lag between the acknowledgement that a particular feature is the source of safety or security problems, the decision to remove or replace the feature and the generation of warnings or error messages by compilers that the feature shouldn't be used. Given that software systems can take many years to develop, it is possible and even likely that a language standard will change causing some of the features used to be suddenly deprecated. Modifying the software can be costly and time consuming to remove the deprecated features. However, if the schedule and resources permit, this would be prudent as future vulnerabilities may result from leaving the deprecated features in the code. Ultimately the deprecated features will likely need to be removed when the features are removed.

6.57.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- All languages that have standards, though some only have defacto standards.
- All languages that evolve over time and as such could potentially have deprecated features at some point.

6.57.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Adhere to the latest published standard for which a suitable complier and development environment is available.
- Avoid the use of deprecated features of a language.
- Stay abreast of language discussions in language user groups and standards groups on the Internet. Discussions and meeting notes will give an indication of problem prone features that should not be used or should be used with caution.

6.57.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Obscure language features for which there are commonly used alternatives should be considered for removal from the language standard.

- Obscure language features that have routinely been found to be the root cause of safety or security vulnerabilities, or that are routinely disallowed in software guidance documents should be considered for removal from the language standard.
- Language designers should provide language mechanisms that optionally disable deprecated language features.

7. Application Vulnerabilities

7.1 General

This clause provides descriptions of selected application vulnerabilities which have been found and exploited in a number of applications and which have well known mitigation techniques, and which result from design decisions made by coders in the absence of suitable language library routines or other mechanisms. For these vulnerabilities, each description provides:

- a summary of the vulnerability,
- typical mechanisms of failure, and
- techniques that programmers can use to avoid the vulnerability

7.2 Terminology

These vulnerabilities are application-related rather than language-related. They are written in a language-independent manner, and there are no corresponding sections in the annexes.

7.3 Unspecified Functionality [BVQ]

7.3.1 Description of application vulnerability

Unspecified functionality is code that may be executed, but whose behaviour does not contribute to the requirements of the application. While this may be no more than an amusing 'Easter Egg', like the flight simulator in a spreadsheet, it does raise questions about the level of control of the development process.

In a security-critical environment particularly, the developer of an application could include a 'trap-door' to allow illegitimate access to the system on which it is eventually executed, irrespective of whether the application has obvious security requirements.

7.3.2 Cross reference

JSF AV Rule: 127
MISRA C 2004: 2.2, 2.3, 2.4, and 14.1
XYQ: Dead and Deactivated code.

7.3.3 Mechanism of failure

Unspecified functionality is not a software vulnerability per se, but more a development issue. In some cases, unspecified functionality may be added by a developer without the knowledge of the development organization.

In other cases, typically Easter Eggs, the functionality is unspecified as far as the user is concerned (nobody buys a spreadsheet expecting to find it includes a flight simulator), but is specified by the development organization. In effect they only reveal a subset of the program's behaviour to the users.

In the first case, one would expect a well managed development environment to discover the additional functionality during validation and verification. In the second case, the user is relying on the supplier not to release harmful code.

In effect, a program's requirements are 'the program should behave in the following manner and do nothing else'. The 'and do nothing else' clause is often not explicitly stated, and can be difficult to demonstrate.

7.3.4 Avoiding the vulnerability or mitigating its effects

End users can avoid the vulnerability or mitigate its ill effects in the following ways:

- Programs and development tools that are to be used in critical applications should come from a developer who uses a recognized and audited development process for the development of those programs and tools. For example: ISO 9001 or CMMI®.
- The development process should generate documentation showing traceability from source code to requirements, in effect answering 'why is this unit of code in this program?'. Where unspecified functionality is there for a legitimate reason (such as diagnostics required for developer maintenance or enhancement), the documentation should also record this. It is not unreasonable for customers of bespoke critical code to ask to see such traceability as part of their acceptance of the application.

7.4 Distinguished Values in Data Types [KLK]

7.4.1 Description of application vulnerability

Sometimes, in a type representation, certain values are distinguished as not being members of the type, but rather as providing auxiliary information. Examples include special characters used as string terminators, distinguished values used to indicate out of type entries in *SQL* (Structured Query Language) database fields, and sentinels used to indicate the bounds of queues or other data structures. When the usage pattern of code containing distinguished values is changed, it may happen that the distinguished value happens to coincide with a legitimate in-type value. In such a case, the value is no longer distinguishable from an in-type value and the software will no longer produce the intended results.

7.4.2 Cross reference

CWE:

20. Improper input validation
137. Representation errors

JSF AV Rule: 151

7.4.3 Mechanism of failure

A "distinguished value" or a "magic number" in the representation of a data type might be used to represent out-of-type information. Some examples include the following:

- The use of a special code, such as “00”, to indicate the termination of a coded character string.
- The use of a special value, such as “999…9”, as the indication that the actual value is either not known or is invalid.

If the use of the software is later generalized, the once-special value can become indistinguishable from valid data. Note that the problem may occur simply if the pattern of usage of the software is changed from that anticipated by the software’s designers. It may also occur if the software is reused in other circumstances.

An example of a change in the pattern of usage is this: An organization logs visitors to its buildings by recording their names and national identity numbers or social security numbers in a database. Of course, some visitors legitimately don’t have or don’t know their social security number, so the receptionists enter numbers to “make the computer happy.” Receptionists at one building have adopted the convention of using the code “555-55-5555” to designate children of employees. Receptionists at another building have used the same code to designate foreign nationals. When the databases are merged, the children are reclassified as foreign nationals or vice-versa depending on which set of receptionists are using the newly merged database.

An example of an unanticipated change due to reuse is this: Suppose a software component analyzes radar data, recording data every degree of azimuth from 0 to 359. Packets of data are sent to other components for processing, updating displays, recording, and so on. Since all degree values are non-negative, a distinguished value of -1 is used as a signal to stop processing, compute summary data, close files, and so on. Many of the components are to be reused in a new system with a new radar analysis component. However the new component represents direction by numbers in the range -180 degrees to 179 degrees. When an azimuth value of -1 is provided, the downstream components will interpret that as the indication to stop processing. If the magic value is changed to, say, -999, the software is still at risk of failing when future enhancements (say, counting accumulated degrees on complete revolutions) bring -999 into the range of valid data.

Distinguished values should be avoided. Instead, the software should be designed to use distinct variables to encode the desired out-of-type information. For example, the length of a character string might be encoded in a dope vector and validity of data entries might be encoded in distinct Boolean values.

7.4.4 Avoiding the vulnerability or mitigating its effects

End users can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use auxiliary variables (perhaps enclosed in variant records) to encode out-of-type information.
- Use enumeration types to convey category information. Do not rely upon large ranges of integers, with distinguished values having special meanings.
- Use named constants to make it easier to change distinguished values.

7.5 Adherence to Least Privilege [XYN]

7.5.1 Description of application vulnerability

Failure to adhere to the principle of least privilege amplifies the risk posed by other vulnerabilities.

7.5.2 Cross reference

CWE:

250. Design Principle Violation: Failure to Use Least Privilege

CERT C guidelines: POS02-C

7.5.3 Mechanism of failure

This vulnerability type refers to cases in which an application grants greater access rights than necessary. Depending on the level of access granted, this may allow a user to access confidential information. For example, programs that run with root privileges have caused innumerable UNIX security disasters. It is imperative that you carefully review privileged programs for all kinds of security problems, but it is equally important that privileged programs drop back to an unprivileged state as quickly as possible to limit the amount of damage that an overlooked vulnerability might be able to cause. Privilege management functions can behave in some less-than-obvious ways, and they have different quirks on different platforms. These inconsistencies are particularly pronounced if you are transitioning from one non-root user to another. Signal handlers and spawned processes run at the privilege of the owning process, so if a process is running as root when a signal fires or a sub-process is executed, the signal handler or sub-process will operate with root privileges. An attacker may be able to leverage these elevated privileges to do further damage. To grant the minimum access level necessary, first identify the different permissions that an application or user of that application will need to perform their actions, such as file read and write permissions, network socket permissions, and so forth. Then explicitly allow those actions while denying all else.

7.5.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Very carefully manage the setting, management and handling of privileges. Explicitly manage trust zones in the software.
- Follow the principle of least privilege when assigning access rights to entities in a software system.

7.6 Privilege Sandbox Issues [XYO]

7.6.1 Description of application vulnerability

A variety of vulnerabilities occur with improper handling, assignment, or management of privileges. These are especially present in sandbox environments, although it could be argued that any privilege problem occurs within the context of some sort of sandbox.

7.6.2 Cross reference

CWE:

266. Incorrect Privilege Assignment
267. Privilege Defined With Unsafe Actions
268. Privilege Chaining
269. Privilege Management Error

270. Privilege Context Switching Error
272. Least Privilege Violation
273. Failure to Check Whether Privileges were Dropped Successfully
274. Failure to Handle Insufficient Privileges
276. Insecure Default Permissions
732. Incorrect Permission Assignment for Critical Resource

CERT C guidelines: POS36-C

7.6.3 Mechanism of failure

The failure to drop system privileges when it is reasonable to do so is not an application vulnerability by itself. It does, however, serve to significantly increase the severity of other vulnerabilities. According to the principle of least privilege, access should be allowed only when it is absolutely necessary to the function of a given system, and only for the minimal necessary amount of time. Any further allowance of privilege widens the window of time during which a successful exploitation of the system will provide an attacker with that same privilege.

Many situations could lead to a mechanism of failure:

- A product could incorrectly assign a privilege to a particular entity.
- A particular privilege, role, capability, or right could be used to perform unsafe actions that were not intended, even when it is assigned to the correct entity. (Note that there are two separate sub-categories here: privilege incorrectly allows entities to perform certain actions; and the object is incorrectly accessible to entities with a given privilege.)
- Two distinct privileges, roles, capabilities, or rights could be combined in a way that allows an entity to perform unsafe actions that would not be allowed without that combination.
- The software may not properly manage privileges while it is switching between different contexts that cross privilege boundaries.
- A product may not properly track, modify, record, or reset privileges.
- In some contexts, a system executing with elevated permissions will hand off a process/file or other object to another process/user. If the privileges of an entity are not reduced, then elevated privileges are spread throughout a system and possibly to an attacker.
- The software may not properly handle the situation in which it has insufficient privileges to perform an operation.
- A program, upon installation, may set insecure permissions for an object.

7.6.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- The principle of least privilege when assigning access rights to entities in a software system should be followed. The setting, management and handling of privileges should be managed very carefully. Upon changing security privileges, one should ensure that the change was successful.
- Consider following the principle of separation of privilege. Require multiple conditions to be met before permitting access to a system resource.

- Trust zones in the software should be explicitly managed. If at all possible, limit the allowance of system privilege to small, simple sections of code that may be called atomically.
- As soon as possible after acquiring elevated privilege to call a privileged function such as `chroot()`, the program should drop root privilege and return to the privilege level of the invoking user.
- In newer Windows implementations, make sure that the process token has the SeImpersonatePrivilege.

7.7 Executing or Loading Untrusted Code [XYS]

7.7.1 Description of application vulnerability

Executing commands or loading libraries from an untrusted source or in an untrusted environment can cause an application to execute malicious commands (and payloads) on behalf of an attacker.

7.7.2 Cross reference

CWE:

114. Process Control

306. Missing Authentication for Critical Function

CERT C guidelines: PRE09-C, ENV02-C, and ENV03-C

7.7.3 Mechanism of failure

Process control vulnerabilities take two forms:

- An attacker can change the command that the program executes so that the attacker explicitly controls what the command is.
- An attacker can change the environment in which the command executes so that the attacker implicitly controls what the command means.

Considering only the first scenario, the possibility that an attacker may be able to control the command that is executed, process control vulnerabilities occur when:

- Data enters the application from a source that is not trusted.
- The data is used as or as part of a string representing a command that is executed by the application.
- By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

7.7.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Libraries that are loaded should be well understood and come from a trusted source with a digital signature. The application can execute code contained in native libraries, which often contain calls that are susceptible to other security problems, such as buffer overflows or command injection.
- All native libraries should be validated.
- Determine if the application requires the use of the native library. It can be very difficult to determine what these libraries actually do, and the potential for malicious code is high.

- To help prevent buffer overflow attacks, validate all input to native calls for content and length.
- If the native library does not come from a trusted source, review the source code of the library. The library should be built from the reviewed source before using it.

7.7.5 Implications for standardization

In future standardization activities, the following items should be considered:

- Language independent APIs for code signing and data signing should be defined, allowing each Programming Language to define a binding.

7.8 Memory Locking [XZX]

7.8.1 Description of application vulnerability

Sensitive data stored in memory that was not locked or that has been improperly locked may be written to swap files on disk by the virtual memory manager.

7.8.2 Cross reference

CWE:

591. Sensitive Data Storage in Improperly Locked Memory

CERT C guidelines: MEM06-C

7.8.3 Mechanism of failure

Sensitive data that is not kept cryptographically secure may become visible to an attacker by any of several mechanisms. Some operating systems may write memory to swap or page files that may be visible to an attacker. Some operating systems may provide mechanisms to examine the physical memory of the system or the virtual memory of another application. Application debuggers may be able to stop the target application and examine or alter memory.

7.8.4 Avoiding the vulnerability or mitigating its effects

In almost all cases, these attacks require elevated or appropriate privilege.

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Remove debugging tools from production systems.
- Log and audit all privileged operations.
- Identify data that needs to be protected and use appropriate cryptographic and other data obfuscation techniques to avoid keeping plaintext versions of this data in memory or on disk.
- If the operating system allows, clear the swap file on shutdown.

Note: Several implementations of the POSIX `mlock()` and the Microsoft Windows `VirtualLock()` functions will prevent the named memory region from being written to a swap or page file. However, such usage is not portable.

Systems that provide a "hibernate" facility (such as laptops) will write all of physical memory to a file that may be visible to an attacker on resume.

7.8.5 Implications for standardization

In future standardization activities, the following items should be considered:

- Language independent APIs for memory locking should be defined, allowing each Programming Language to define a binding.

7.9 Resource Exhaustion [XZP]

7.9.1 Description of application vulnerability

The application is susceptible to generating and/or accepting an excessive number of requests that could potentially exhaust limited resources, such as memory, file system storage, database connection pool entries, or CPU. This could ultimately lead to a denial of service that could prevent any other applications from accessing these resources.

7.9.2 Cross reference

CWE:

400. Resource Exhaustion

7.9.3 Mechanism of failure

There are two primary failures associated with resource exhaustion. The most common result of resource exhaustion is denial of service. In some cases an attacker or a defect may cause a system to fail in an unsafe or insecure fashion by causing an application to exhaust the available resources.

Resource exhaustion issues are generally understood but are far more difficult to prevent. Taking advantage of various entry points, an attacker could craft a wide variety of requests that would cause the site to consume resources. Database queries that take a long time to process are good *DoS* (Denial of Service) targets. An attacker would only have to write a few lines of Perl code to generate enough traffic to exceed the site's ability to keep up. This would effectively prevent authorized users from using the site at all.

Resources can be exhausted simply by ensuring that the target machine must do much more work and consume more resources to service a request than the attacker must do to initiate a request. Prevention of these attacks requires either that the target system either recognizes the attack and denies that user further access for a given amount of time or uniformly throttles all requests to make it more difficult to consume resources more quickly than they can again be freed. The first of these solutions is an issue in itself though, since it may allow attackers to prevent the use of the system by a particular valid user. If the attacker impersonates the valid user, he may be able to prevent the user from accessing the server in question. The second solution is simply difficult to effectively institute and even when properly done, it does not provide a full solution. It simply makes the attack require more resources on the part of the attacker.

The final concern that must be discussed about issues of resource exhaustion is that of systems which "fail open." This means that in the event of resource consumption, the system fails in such a way that the state of the system — and possibly the security functionality of the system — are compromised. A prime example of this can be found in old switches that were vulnerable to "macof" attacks (so named for a tool developed by Dugsong). These attacks flooded a switch with random IP(Internet Protocol) and MAC(Media Access Control) address combinations, therefore exhausting the switch's cache, which held the information of which port corresponded to which MAC addresses. Once this cache was exhausted, the switch would fail in an insecure way and would begin to act simply as a hub, broadcasting all traffic on all ports and allowing for basic sniffing attacks.

7.9.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Implement throttling mechanisms into the system architecture. The best protection is to limit the amount of resources that an application can cause to be expended. A strong authentication and access control model will help prevent such attacks from occurring in the first place. The authentication application should be protected against denial of service attacks as much as possible. Limiting the database access, perhaps by caching result sets, can help minimize the resources expended. To further limit the potential for a denial of service attack, consider tracking the rate of requests received from users and blocking requests that exceed a defined rate threshold.
- Ensure that applications have specific limits of scale placed on them, and ensure that all failures in resource allocation cause the application to fail safely.

7.10 Unrestricted File Upload [CBF]

7.10.1 Description of application vulnerability

A first step often used to attack is to get an executable on the system to be attacked. Then the attack only needs to execute this code. Many times this first step is accomplished by unrestricted file upload. In many of these attacks, the malicious code can obtain the same privilege of access as the application, or even administrator privilege.

7.10.2 Cross reference

CWE:

434. Unrestricted Upload of File with Dangerous Type

7.10.3 Mechanism of failure

There are several failures associated with an uploaded file:

- Executing arbitrary code.
- Phishing page added to a website.
- Defacing a website.
- Creating a vulnerability for other attacks.
- Browsing the file system.

- Creating a denial of service.
- Uploading a malicious executable to a server, which could be executed with administrator privilege.

7.10.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Allow only certain file extensions, commonly known as a *white-list*.
- Disallow certain file extensions, commonly known as a *black-list*.
- Use a utility to check the type of the file.
- Check the content-type in the header information of all files that are uploaded. The purpose of the content-type field is to describe the data contained in the body completely enough that the receiving agent can pick an appropriate agent or mechanism to present the data to the user, or otherwise deal with the data in an appropriate manner.
- Use a dedicated location, which does not have execution privileges, to store and validate uploaded files, and then serve these files dynamically.
- Require a unique file extension (named by the application developer), so only the intended type of the file is used for further processing. Each upload facility of an application could handle a unique file type.
- Remove all Unicode characters and all control characters[5] from the filename and the extensions.
- Set a limit for the filename length; including the file extension. In an *NTFS* (New Technology File System) partition, usually a limit of 255 characters, without path information will suffice.
- Set upper and lower limits on file size. Setting these limits can help in denial of service attacks.

All of the above have some short comings, for example, a GIF (.gif) file may contain a free-form comment field, and therefore a sanity check of the files contents is not always possible. An attacker can hide code in a file segment that will still be executed by the application or server. In many cases it will take a combination of the techniques from the above list to avoid this vulnerability.

7.10.5 Implications for standardization

In future standardization activities, the following items should be considered:

- Language independent APIs for file identification should be defined, allowing each Programming Language to define a binding.

7.11 Resource Names [HTS]

7.11.1 Description of application vulnerability

Interfacing with the directory structure or other external identifiers on a system on which software executes is very common. Differences in the conventions used by operating systems can result in significant changes in behaviour when the same program is executed under different operating systems. For instance, the directory structure, permissible characters, case sensitivity, and so forth can vary among operating systems and even

[5] See http://www.ascii.cl/control-characters.htm

among variations of the same operating system. For example, Microsoft prohibits "/?:&*"<>|#%"; but UNIX, Linux, and OS X operating systems allow any character except for the reserved character '/' to be used in a filename.

Some operating systems are case sensitive while others are not. On non-case sensitive operating systems, depending on the software being used, the same filename could be displayed, as "filename", "Filename" or "FILENAME" and all would refer to the same file.

Some operating systems, particularly older ones, only rely on the significance of the first `n` characters of the file name. `n` can be unexpectedly small, such as the first 8 characters in the case of Win16 architectures which would cause "filename1", "filename2" and "filename3" to all map to the same file.

Variations in the filename, named resource or external identifier being referenced can be the basis for various kinds of problems. Such mistakes or ambiguity can be unintentional, or intentional, and in either case they can be potentially exploited, if surreptitious behaviour is a goal.

7.11.2 Cross reference

JSF AV Rules: 46, 51, 53, 54, 55, and 56
MISRA C 2004: 1.4 and 5.1
CERT C guidelines: MSC09-C and MSC10-C

7.11.3 Mechanism of Failure

The wrong named resource, such as a file, may be used within a program in a form that provides access to a resource that was not intended to be accessed. Attackers could exploit this situation to intentionally misdirect access of a named resource to another named resource.

7.11.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Where possible, use an API that provides a known common set of conventions for naming and accessing external resources, such as POSIX, ISO/IEC 9945:2003 (IEEE Std 1003.1-2001).
- Analyze the range of intended target systems, develop a suitable API for dealing with them, and document the analysis.
- Ensure that programs adapt their behaviour to the platform on which they are executing, so that only the intended resources are accessed. The means that information on such characteristics as the directory separator string and methods of accessing parent directories need to be parameterized and not exist as fixed strings within a program.
- Avoid creating resource names that are longer than the guaranteed unique length of all potential target platforms.
- Avoid creating resources, which are differentiated only by the case in their names.
- Avoid all Unicode characters and all control characters[6] in filenames and the extensions.

[6] See http://www.ascii.cl/control-characters.htm

7.11.5 Implications for standardization

In future standardization activities, the following items should be considered:

- Language independent APIs for interfacing with external identifiers should be defined, allowing each Programming Language to define a binding.

7.12 Injection [RST]

7.12.1 Description of application vulnerability

Injection problems span a wide range of instantiations. The basic form of this weakness involves the software allowing injection of additional data in input data to alter the control flow of the process. Command injection problems are a subset of injection problem, in which the process can be tricked into calling external processes of an attacker's choice through the injection of command syntax into the input data. Multiple leading/internal/trailing special elements injected into an application through input can be used to compromise a system. As data is parsed, improperly handled multiple leading special elements may cause the process to take unexpected actions that result in an attack. Software may allow the injection of special elements that are non-typical but equivalent to typical special elements with control implications. This frequently occurs when the product has protected itself against special element injection. Software may allow inputs to be fed directly into an output file that is later processed as code, such as a library file or template. Line or section delimiters injected into an application can be used to compromise a system.

Many injection attacks involve the disclosure of important information — in terms of both data sensitivity and usefulness in further exploitation. In some cases injectable code controls authentication; this may lead to a remote vulnerability. Injection attacks are characterized by the ability to significantly change the flow of a given process, and in some cases, to the execution of arbitrary code. Data injection attacks lead to loss of data integrity in nearly all cases as the control-plane data injected is always incidental to data recall or writing. Often the actions performed by injected control code are not logged.

SQL injection attacks are a common instantiation of injection attack, in which SQL commands are injected into input to effect the execution of predefined SQL commands. Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities. If poorly implemented SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password. If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of the SQL injection vulnerability. Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL injection attack.

Injection problems encompass a wide variety of issues — all mitigated in very different ways. The most important issue to note is that all injection problems share one thing in common — they allow for the injection of control data into the user controlled data. This means that the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism. While buffer overflows and many other flaws involve the use of some further issue to gain execution, injection problems need only for the data to be parsed. Many injection attacks involve the disclosure of important information in terms of both data sensitivity and

usefulness in further exploitation. In some cases injectable code controls authentication, this may lead to a remote vulnerability.

7.12.2 Cross reference

CWE:

74. Failure to Sanitize Data into a Different Plane ('Injection')
76. Failure to Resolve Equivalent Special Elements into a Different Plane
78. Failure to Sanitize Data into an OS Command (aka 'OS Command Injection')
89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
90. Failure to Sanitize Data into LDAP Queries (aka 'LDAP Injection')
91. XML Injection (aka Blind XPath Injection)
92. Custom Special Character Injection
95. Insufficient Control of Directives in Dynamically Code Evaluated Code (aka 'Eval Injection')
97. Failure to Sanitize Server-Side Includes (SSI) Within a Web Page
98. Insufficient Control of Filename for Include/Require Statement in PHP Program (aka 'PHP File Inclusion')
99. Insufficient Control of Resource Identifiers (aka 'Resource Injection')
144. Failure to Sanitize Line Delimiters
145. Failure to Sanitize Section Delimiters
161. Failure to Sanitize Multiple Leading Special Elements
163. Failure to Sanitize Multiple Trailing Special Elements
165. Failure to Sanitize Multiple Internal Special Elements
166. Failure to Handle Missing Special Element
167. Failure to Handle Additional Special Element
168. Failure to Resolve Inconsistent Special Elements
564. SQL Injection: Hibernate

CERT C guidelines: FIO30-C

7.12.3 Mechanism of failure

A software system that accepts and executes input in the form of operating system commands (such as `system()`, `exec()`, `open()`) could allow an attacker with lesser privileges than the target software to execute commands with the elevated privileges of the executing process. Command injection is a common problem with wrapper programs. Often, parts of the command to be run are controllable by the end user. If a malicious user injects a character (such as a semi-colon) that delimits the end of one command and the beginning of another, he may then be able to insert an entirely new and unrelated command to do whatever he pleases.

Dynamically generating operating system commands that include user input as parameters can lead to command injection attacks. An attacker can insert operating system commands or modifiers in the user input that can cause the request to behave in an unsafe manner. Such vulnerabilities can be very dangerous and lead to data and system compromise. If no validation of the parameter to the exec command exists, an attacker can execute any command on the system the application has the privilege to access.

There are two forms of command injection vulnerabilities. An attacker can change the command that the program executes (the attacker explicitly controls what the command is). Alternatively, an attacker can change

the environment in which the command executes (the attacker implicitly controls what the command means). The first scenario where an attacker explicitly controls the command that is executed can occur when:

- Data enters the application from an untrusted source.
- The data is part of a string that is executed as a command by the application.
- By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

Eval injection occurs when the software allows inputs to be fed directly into a function (such as "eval") that dynamically evaluates and executes the input as code, usually in the same interpreted language that the product uses. Eval injection is prevalent in handler/dispatch procedures that might want to invoke a large number of functions, or set a large number of variables.

A PHP file inclusion occurs when a PHP product uses `require` or `include` statements, or equivalent statements, that use attacker-controlled data to identify code or *HTML* (HyperText Markup Language) to be directly processed by the PHP interpreter before inclusion in the script.

A resource injection issue occurs when the following two conditions are met:

- An attacker can specify the identifier used to access a system resource. For example, an attacker might be able to specify part of the name of a file to be opened or a port number to be used.
- By specifying the resource, the attacker gains a capability that would not otherwise be permitted. For example, the program may give the attacker the ability to overwrite the specified file, run with a configuration controlled by the attacker, or transmit sensitive information to a third-party server. Note: Resource injection that involves resources stored on the file system goes by the name path manipulation and is reported in separate category. See Path Traversal [EWR] description for further details of this vulnerability. Allowing user input to control resource identifiers may enable an attacker to access or modify otherwise protected system resources.

Line or section delimiters injected into an application can be used to compromise a system. As data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions that result in an attack. One example of a section delimiter is the boundary string in a multipart *MIME* (Multipurpose Internet Mail Extensions) message. In many cases, doubled line delimiters can serve as a section delimiter.

7.12.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Assume all input is malicious. Use an appropriate combination of black-lists and white-lists to ensure only valid, expected and appropriate input is processed by the system.
- Narrowly define the set of safe characters based on the expected values of the parameter in the request.
- Developers should anticipate that delimiters and special elements would be injected/removed/manipulated in the input vectors of their software system and appropriate mechanisms should be put in place to handle them.
- Implement SQL strings using prepared statements that bind variables. Prepared statements that do not bind variables can be vulnerable to attack.

- Use vigorous white-list style checking on any user input that may be used in a SQL command. Rather than escape meta-characters, it is safest to disallow them entirely since the later use of data that have been entered in the database may neglect to escape meta-characters before use.
- Follow the principle of least privilege when creating user accounts to a SQL database. Users should only have the minimum privileges necessary to use their account. If the requirements of the system indicate that a user can read and modify their own data, then limit their privileges so they cannot read/write others' data.
- Assign permissions to the software system that prevents the user from accessing/opening privileged files.
- Restructure code so that there is not a need to use the `eval()` utility.

7.13 Cross-site Scripting [XYT]

7.13.1 Description of application vulnerability

Cross-site scripting (XSS) occurs when dynamically generated web pages display input, such as login information that is not properly validated, allowing an attacker to embed malicious scripts into the generated page and then execute the script on the machine of any user that views the site. If successful, cross-site scripting vulnerabilities can be exploited to manipulate or steal cookies, create requests that can be mistaken for those of a valid user, compromise confidential information, or execute malicious code on the end user systems for a variety of nefarious purposes.

7.13.2 Cross reference

CWE:

79. Failure to Preserve Web Page Structure ('Cross-site Scripting')
80. Failure to Sanitize Script-Related HTML Tags in a Web Page (Basic XSS)
81. Failure to Sanitize Directives in an Error Message Web Page
82. Failure to Sanitize Script in Attributes of IMG Tags in a Web Page
83. Failure to Sanitize Script in Attributes in a Web Page
84. Failure to Resolve Encoded URI Schemes in a Web Page
85. Doubled Character XSS Manipulations
86. Invalid Characters in Identifiers
87. Alternate XSS Syntax

7.13.3 Mechanism of failure

Cross-site scripting (XSS) vulnerabilities occur when an attacker uses a web application to send malicious code, generally JavaScript, to a different end user. When a web application uses input from a user in the output it generates without filtering it, an attacker can insert an attack in that input and the web application sends the attack to other users. The end user trusts the web application, and the attacks exploit that trust to do things that would not normally be allowed. Attackers frequently use a variety of methods to encode the malicious portion of the tag, such as using Unicode, so the request looks less suspicious to the user.

XSS attacks can generally be categorized into two categories: stored and reflected. Stored attacks are those where the injected code is permanently stored on the target servers in a database, message forum, visitor log,

and so forth. Reflected attacks are those where the injected code takes another route to the victim, such as in an email message, or on some other server. When a user is tricked into clicking a link or submitting a form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. The browser then executes the code because it came from a 'trusted' server. For a reflected XSS attack to work, the victim must submit the attack to the server. This is still a very dangerous attack given the number of possible ways to trick a victim into submitting such a malicious request, including clicking a link on a malicious Web site, in an email, or in an inter-office posting.

XSS flaws are very common in web applications, as they require a great deal of developer discipline to avoid them in most applications. It is relatively easy for an attacker to find XSS vulnerabilities. Some of these vulnerabilities can be found using scanners, and some exist in older web application servers. The consequence of an XSS attack is the same regardless of whether it is stored or reflected.

The difference is in how the payload arrives at the server. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. The most severe XSS attacks involve disclosure of the user's session cookie, which allows an attacker to hijack the user's session and take over their account. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirecting the user to some other page or site, and modifying presentation of content.

Cross-site scripting (XSS) vulnerabilities occur when:

- Data enters a Web application through an untrusted source, most frequently a web request. The data is included in dynamic content that is sent to a web user without being validated for malicious code.
- The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Cross-site scripting attacks can occur wherever an untrusted user has the ability to publish content to a trusted web site. Typically, a malicious user will craft a client-side script, which — when parsed by a web browser — performs some activity (such as sending all site cookies to a given e–mail address). If the input is unchecked, this script will be loaded and run by each user visiting the web site. Since the site requesting to run the script has access to the cookies in question, the malicious script does also. There are several other possible attacks, such as running "Active X" controls (under Microsoft Internet Explorer) from sites that a user perceives as trustworthy; cookie theft is however by far the most common. All of these attacks are easily prevented by ensuring that no script tags — or for good measure, HTML tags at all — are allowed in data to be posted publicly.

Specific instances of XSS are:

- 'Basic' XSS involves a complete lack of cleansing of any special characters, including the most fundamental XSS elements such as "<", ">", and "&".
- A web developer displays input on an error page (such as a customized 403 Forbidden page). If an attacker can influence a victim to view/request a web page that causes an error, then the attack may be successful.

- A Web application that trusts input in the form of HTML IMG tags is potentially vulnerable to XSS attacks. Attackers can embed XSS exploits into the values for IMG attributes (such as SRC) that is streamed and then executed in a victim's browser. Note that when the page is loaded into a user's browser, the exploit will automatically execute.
- The software does not filter "JavaScript:" or other *URI*'s (Uniform Resource Identifier) from dangerous attributes within tags, such as `onmouseover`, `onload`, `onerror`, or `style`.
- The web application fails to filter input for executable script disguised with URI encodings.
- The web application fails to filter input for executable script disguised using doubling of the involved characters.
- The software does not strip out invalid characters in the middle of tag names, schemes, and other identifiers, which are still rendered by some web browsers that ignore the characters.
- The software fails to filter alternate script syntax provided by the attacker.

Cross-site scripting attacks may occur anywhere that possibly malicious users are allowed to post unregulated material to a trusted web site for the consumption of other valid users. The most common example can be found in bulletin-board web sites that provide web based mailing list-style functionality. The most common attack performed with cross-site scripting involves the disclosure of information stored in user cookies. In some circumstances it may be possible to run arbitrary code on a victim's computer when cross-site scripting is combined with other flaws.

7.13.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Carefully check each input parameter against a rigorous positive specification (white-list) defining the specific characters and format allowed.
- All input should be sanitized, not just parameters that the user is supposed to specify, but all data in the request, including hidden fields, cookies, headers, the *URL* (Uniform Resource Locator) itself, and so forth.
- A common mistake that leads to continuing XSS vulnerabilities is to validate only fields that are expected to be redisplayed by the site.
- Data is frequently encountered from the request that is reflected by the application server or the application that the development team did not anticipate. Also, a field that is not currently reflected may be used by a future developer. Therefore, validating ALL parts of the *HTTP* (Hypertext Transfer Protocol) request is recommended.

7.14 Unquoted Search Path or Element[XZQ]

7.14.1 Description of application vulnerability

Strings injected into a software system that are not quoted can permit an attacker to execute arbitrary commands.

7.14.2 Cross reference

CWE:

428. Unquoted Search Path or Element

CERT C guidelines: ENV04-C

7.14.3 Mechanism of failure

The mechanism of failure stems from missing quoting of strings injected into a software system. By allowing white-spaces in identifiers, an attacker could potentially execute arbitrary commands. This vulnerability covers "`C:\Program Files`" and space-in-search-path issues. Theoretically this could apply to other operating systems besides Windows, especially those that make it easy for spaces to be in filenames or folders names.

7.14.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Software should quote the input data that can be potentially executed on a system.
- Use a programming language that enforces the quoting of strings.

7.15 Improperly Verified Signature [XZR]

7.15.1 Description of application vulnerability

The software does not verify, or improperly verifies, the cryptographic signature for data. By not adequately performing the verification step, the data being received should not be trusted and may be corrupted or made intentionally incorrect by an adversary.

7.15.2 Cross reference

CWE:

347. Improperly Verified Signature

7.15.3 Mechanism of failure

Data is signed using techniques that assure the integrity of the data. There are two ways that the integrity can be intentionally compromised. The exchange of the cryptologic keys may have been compromised so that an attacker could provide encrypted data that has been altered. Alternatively, the cryptologic verification could be flawed so that the encryption of the data is flawed which again allows an attacker to alter the data.

7.15.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use data signatures to the extent possible to help ensure trust in data.
- Use built-in verifications for data.

7.15.5 Implications for standardization

In future standardization activities, the following items should be considered:

- Language independent APIs for data signing should be defined, allowing each Programming Language to define a binding.

7.16 Discrepancy Information Leak [XZL]

7.16.1 Description of application vulnerability

A discrepancy information leak is an information leak in which the product behaves differently, or sends different responses, in a way that reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

7.16.2 Cross reference

CWE:

203. Discrepancy Information Leaks
204. Response Discrepancy Information Leak
206. Internal Behavioural Inconsistency Information Leak
207. External Behavorial Inconsistency Information Leak
208. Timing Discrepancy Information Leak

7.16.3 Mechanism of failure

A response discrepancy information leak occurs when the product sends different messages in direct response to an attacker's request, in a way that allows the attacker to learn about the inner state of the product. The leaks can be inadvertent (bug) or intentional (design).

A behavioural discrepancy information leak occurs when the product's actions indicate important differences based on (1) the internal state of the product or (2) differences from other products in the same class. Attacks such as OS fingerprinting rely heavily on both behavioural and response discrepancies. An internal behavioural inconsistency information leak is the situation where two separate operations in a product cause the product to behave differently in a way that is observable to an attacker and reveals security-relevant information about the internal state of the product, such as whether a particular operation was successful or not. An external behavioural inconsistency information leak is the situation where the software behaves differently than other products like it, in a way that is observable to an attacker and reveals security-relevant information about which product is being used, or its operating state.

A timing discrepancy information leak occurs when two separate operations in a product require different amounts of time to complete, in a way that is observable to an attacker and reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

7.16.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Compartmentalize the system to have "safe" areas where trust boundaries can be unambiguously drawn.
- Do not allow sensitive data to go outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area.

7.17 Sensitive Information Uncleared Before Use [XZK]

7.17.1 Description of application vulnerability

The software does not fully clear previously used information in a data structure, file, or other resource, before making that resource available to another party that did not have access to the original information.

7.17.2 Cross reference

CWE:

226. Sensitive Information Uncleared Before Release

CERT C guidelines: MEM03-C

7.17.3 Mechanism of failure

This typically involves memory in which the new data occupies less memory than the old data, which leaves portions of the old data still available ("memory disclosure"). However, equivalent errors can occur in other situations where the length of data is variable but the associated data structure is not. This can overlap with cryptographic errors and cross-boundary cleansing information leaks.

Dynamic memory managers are not required to clear freed memory and generally do not because of the additional runtime overhead. Furthermore, dynamic memory managers are free to reallocate this same memory. As a result, it is possible to accidentally leak sensitive information if it is not cleared before calling a function that frees dynamic memory. Programmers should not and can't rely on memory being cleared during allocation.

7.17.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use library functions and or programming language features (such as destructors or finalization procedures) that provide automatic clearing of freed buffers or the functionality to clear buffers.

7.18 Path Traversal [EWR]

7.18.1 Description of application vulnerability

The software constructs a path that contains relative traversal sequence such as ".." or an absolute path sequence such as "/path/here." Attackers run the software in a particular directory so that the hard link or symbolic link used by the software accesses a file that the attacker has under their control. In doing this, the attacker may be able to escalate their privilege level to that of the running process.

7.18.2 Cross reference

CWE:

22. Path Traversal
24. Path Traversal: - '../filedir'
25. Path Traversal: '/../filedir'

26. Path Traversal: '/dir/../filename’
27. Path Traversal: 'dir/../../filename'
28. Path Traversal: '..\filename'
29. Path Traversal: '\..\filename'
30. Path Traversal: '\dir\..\filename'
31. Path Traversal: 'dir\..\filename'
32. Path Traversal: '...' (Triple Dot)
33. Path Traversal: '....' (Multiple Dot)
34. Path Traversal: '....//'
35. Path Traversal: '.../...//'
37. Path Traversal: ‘/absolute/pathname/here’
38. Path Traversal: ‘ \absolute\pathname\here’
39. Path Traversal: 'C:dirname'
40. Path Traversal: '\\UNC\share\name\' (Windows UNC Share)
61. UNIX Symbolic Link (Symlink) Following
62. UNIX Hard Link
64. Windows Shortcut Following (.LNK)
65. Windows Hard Link

CERT C guidelines: FIO02-C

7.18.3 Mechanism of failure

There are two primary ways that an attacker can orchestrate an attack using path traversal. In the first, the attacker alters the path being used by the software to point to a location that the attacker has control over. Alternatively, the attacker has no control over the path, but can alter the directory structure so that the path points to a location that the attacker does have control over.

For instance, a software system that accepts input in the form of: '..\filename', '\..\filename', '/directory/../filename', 'directory/../../filename', '..\filename', '\..\filename', '\directory\..\filename', 'directory\..\..\filename', '...', '....' (multiple dots), '....//', or '.../...//' without appropriate validation can allow an attacker to traverse the file system to access an arbitrary file. Note that '..' is ignored if the current working directory is the root directory. Some of these input forms can be used to cause problems for systems that strip out '..' from input in an attempt to remove relative path traversal.

There are several common ways that an attacker can point a file access to a file the attacker has under their control. A software system that accepts input in the form of '/absolute/pathname/here' or '\absolute\pathname\here' without appropriate validation can also allow an attacker to traverse the file system to unintended locations or access arbitrary files. An attacker can inject a drive letter or Windows volume letter ('C:dirname') into a software system to potentially redirect access to an unintended location or arbitrary file. A software system that accepts input in the form of a backslash absolute path without appropriate validation can allow an attacker to traverse the file system to unintended locations or access arbitrary files. An attacker can inject a Windows UNC (Universal Naming Convention or Uniform Naming Convention) share ('\\UNC\share\name') into a software system to potentially redirect access to an unintended location or arbitrary file. A software system that allows UNIX symbolic links (symlink) as part of paths whether in internal code or through user input can allow an attacker to spoof the symbolic link and traverse the file system to unintended

locations or access arbitrary files. The symbolic link can permit an attacker to read/write/corrupt a file that they originally did not have permissions to access. Failure for a system to check for hard links can result in vulnerability to different types of attacks. For example, an attacker can escalate their privileges if he/she can replace a file used by a privileged program with a hard link to a sensitive file, for example, `etc/passwd`. When the process opens the file, the attacker can assume the privileges of that process.

A software system that allows Windows shortcuts (.LNK) as part of paths whether in internal code or through user input can allow an attacker to spoof the symbolic link and traverse the file system to unintended locations or access arbitrary files. The shortcut (file with the `.lnk` extension) can permit an attacker to read/write a file that they originally did not have permissions to access.

Failure for a system to check for hard links can result in vulnerability to different types of attacks. For example, an attacker can escalate their privileges if he/she can replace a file used by a privileged program with a hard link to a sensitive file (such as `etc/passwd`). When the process opens the file, the attacker can assume the privileges of that process or possibly prevent a program from accurately processing data in a software system.

7.18.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Assume all input is malicious. Attackers can insert paths into input vectors and traverse the file system.
- Use an appropriate combination of black-lists and white-lists to ensure only valid and expected input is processed by the system.
- A sanitizing mechanism can remove characters such as '.' and ';' which may be required for some exploits. An attacker can try to fool the sanitizing mechanism into "cleaning" data into a dangerous form. Suppose the attacker injects a '.' inside a filename (say, "sensi.tiveFile") and the sanitizing mechanism removes the character resulting in the valid filename, "sensitiveFile". If the input data are now assumed to be safe, then the file may be compromised.
- Files can often be identified by other attributes in addition to the file name, for example, by comparing file ownership or creation time. Information regarding a file that has been created and closed can be stored and then used later to validate the identity of the file when it is reopened. Comparing multiple attributes of the file improves the likelihood that the file is the expected one.
- Follow the principle of least privilege when assigning access rights to files.
- Denying access to a file can prevent an attacker from replacing that file with a link to a sensitive file.
- Ensure good compartmentalization in the system to provide protected areas that can be trusted.
- When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories.
- Securely creating temporary files in a shared directory is error prone and dependent on the version of the runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system.
- The mitigation should be centered on converting relative paths into absolute paths and then verifying that the resulting absolute path makes sense with respect to the configuration and rights or permissions.

This may include checking white-lists and black-lists, authorized super user status, access control lists, or other fully trusted status.

7.19 Missing Required Cryptographic Step [XZS]

7.19.1 Description of application vulnerability

Cryptographic implementations should follow the algorithms that define them exactly, otherwise encryption can be faulty.

7.19.2 Cross reference

CWE:

325. Missing Required Cryptographic Step
327. Use of a Broken or Risky Cryptographic Algorithm

7.19.3 Mechanism of failure

Not following the algorithms that define cryptographic implementations exactly can lead to weak encryption. This could be the result of many factors such as a programmer missing a required cryptographic step or using weak randomization algorithms.

7.19.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Implement cryptographic algorithms precisely.
- Use system functions and libraries rather than writing the function.

7.20 Insufficiently Protected Credentials [XYM]

7.20.1 Description of application vulnerability

This weakness occurs when the application transmits or stores authentication credentials and uses an insecure method that is susceptible to unauthorized interception and/or retrieval.

7.20 .2 Cross reference

CWE:

256. Plaintext Storage of a Password
257. Storing Passwords in a Recoverable Format

7.20.3 Mechanism of failure

Storing a password in plaintext may result in a system compromise. Password management issues occur when a password is stored in plaintext in an application's properties or configuration file. A programmer can attempt to remedy the password management problem by obscuring the password with an encoding function, such as

Base64 encoding, but this effort does not adequately protect the password. Storing a plaintext password in a configuration file allows anyone who can read the file access to the password-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plaintext.

The storage of passwords in a recoverable format makes them subject to password reuse attacks by malicious users. If a system administrator can recover the password directly or use a brute force search on the information available to him, he can use the password on other accounts.

The use of recoverable passwords significantly increases the chance that passwords will be used maliciously. In fact, it should be noted that recoverable encrypted passwords provide no significant benefit over plain-text passwords since they are subject not only to reuse by malicious attackers but also by malicious insiders.

7.20.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Avoid storing passwords in easily accessible locations.
- Never store a password in plaintext.
- Ensure that strong, non-reversible encryption is used to protect stored passwords.
- Consider storing cryptographic hashes of passwords as an alternative to storing in plaintext.

7.21 Missing or Inconsistent Access Control [XZN]

7.21.1 Description of application vulnerability

The software does not perform access control checks in a consistent manner across all potential execution paths.

7.21.2 Cross reference

CWE:

285. Missing or Inconsistent Access Control
352. Cross-Site Request Forgery (CSRF**)**
807. Reliance on Untrusted Inputs in a Security Decision

CERT C guidelines: FIO06-C

7.21.3 Mechanism of failure

For web applications, attackers can issue a request directly to a page (URL) that they may not be authorized to access. If the access control policy is not consistently enforced on every page restricted to authorized users, then an attacker could gain access to and possibly corrupt these resources.

7.21.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- For web applications, make sure that the access control mechanism is enforced correctly at the server side on every page. Users should not be able to access any information simply by requesting direct access to that page, if they do not have authorization. Ensure that all pages containing sensitive information are not cached, and that all such pages restrict access to requests that are accompanied by an active and authenticated session token associated with a user who has the required permissions to access that page.

7.22 Authentication Logic Error [XZO]

7.22.1 Description of application vulnerability

The software does not properly ensure that the user has proven their identity.

7.22.2 Cross reference

CWE:

287. Improper Authentication
288. Authentication Bypass by Alternate Path/Channel
289. Authentication Bypass by Alternate Name
290. Authentication Bypass by Spoofing
294. Authentication Bypass by Capture-replay
301. Reflection Attack in an Authentication Protocol
302. Authentication Bypass by Assumed-Immutable Data
303. Improper Implementation of Authentication Algorithm
305. Authentication Bypass by Primary Weakness

7.22.3 Mechanism of failure

There are many ways that an attacker can potentially bypass the validation of a user. Some of the ways are means of impersonating a legitimate user while others are means of bypassing the authentication mechanisms that are in place. In either case, a user who should not have access to the software system gains access.

Authentication bypass by alternate path or channel occurs when a product requires authentication, but the product has an alternate path or channel that does not require authentication. Note that this is often seen in web applications that assume that access to a particular *CGI* (Common Gateway Interface) program can only be obtained through a "front" screen, but this problem is not just in web applications.

Authentication bypass by alternate name occurs when the software performs authentication based on the name of the resource being accessed, but there are multiple names for the resource, and not all names are checked.

Authentication bypass by capture-replay occurs when it is possible for a malicious user to sniff network traffic and bypass authentication by replaying it to the server in question to the same effect as the original message (or with minor changes). Messages sent with a capture-relay attack allow access to resources that are not otherwise accessible without proper authentication. Capture-replay attacks are common and can be difficult to defeat without cryptography. They are a subset of network injection attacks that rely on listening in on previously sent valid commands, then changing them slightly if necessary and resending the same commands to the server. Since any attacker who can listen to traffic can see sequence numbers, it is necessary to sign messages with some kind

of cryptography to ensure that sequence numbers are not simply doctored along with content.

Reflection attacks capitalize on mutual authentication schemes to trick the target into revealing the secret shared between it and another valid user. In a basic mutual-authentication scheme, a secret is known to both a valid user and the server; this allows them to authenticate. In order that they may verify this shared secret without sending it plainly over the wire, they utilize a Diffie-Hellman-style scheme in which they each pick a value, then request the hash of that value as keyed by the shared secret. In a reflection attack, the attacker claims to be a valid user and requests the hash of a random value from the server. When the server returns this value and requests its own value to be hashed, the attacker opens another connection to the server. This time, the hash requested by the attacker is the value that the server requested in the first connection. When the server returns this hashed value, it is used in the first connection, authenticating the attacker successfully as the impersonated valid user.

Authentication bypass by assumed-immutable data occurs when the authentication scheme or implementation uses key data elements that are assumed to be immutable, but can be controlled or modified by the attacker, for example, if a web application relies on a cookie "`Authenticated=1`".

Authentication logic error occurs when the authentication techniques do not follow the algorithms that define them exactly and so authentication can be jeopardized. For instance, a malformed or improper implementation of an algorithm can weaken the authorization technique.

An authentication bypass by primary weakness occurs when the authentication algorithm is sound, but the implemented mechanism can be bypassed as the result of a separate weakness that is primary to the authentication error.

7.22.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Funnel all access through a single choke point to simplify how users can access a resource. For every access, perform a check to determine if the user has permissions to access the resource. Avoid making decisions based on names of resources (for example, files) if those resources can have alternate names.
- Canonicalize the name to match that of the file system's representation of the name. This can sometimes be achieved with an available API (for example, in Win32 the `GetFullPathName` function).
- Utilize some sequence or time stamping functionality along with a checksum that takes this into account to ensure that messages can be parsed only once.
- Use different keys for the initiator and responder or of a different type of challenge for the initiator and responder.

7.23 Hard-coded Password [XYP]

7.23.1 Description of application vulnerability

Hard coded passwords may compromise system security in a way that cannot be easily remedied. It is never a good idea to hardcode a password. Not only does hard coding a password allow all of the project's developers to

view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.

7.23.2 Cross reference

CWE:

259. Hard-Coded Password
798. Use of Hard-coded Credentials

7.23.3 Mechanism of failure

The use of a hard-coded password has many negative implications – the most significant of these being a failure of authentication measures under certain circumstances. On many systems, a default administration account exists which is set to a simple default password that is hard-coded into the program or device. This hard-coded password is the same for each device or system of this type and often is not changed or disabled by end users. If a malicious user comes across a device of this kind, it is a simple matter of looking up the default password (which is likely freely available and public on the Internet) and logging in with complete access. In systems that authenticate with a back-end service, hard-coded passwords within closed source or drop-in solution systems require that the back-end service use a password that can be easily discovered. Client-side systems with hard-coded passwords present even more of a threat, since the extraction of a password from a binary is exceedingly simple. If hard-coded passwords are used, it is almost certain that unauthorized users will gain access through the account in question.

7.23.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Rather than hard code a default username and password for first time logins, utilize a "first login" mode that requires the user to enter a unique strong password.
- For front-end to back-end connections, there are three solutions that may be used.
    1. Use of generated passwords that are changed automatically and must be entered at given time intervals by a system administrator. These passwords will be held in memory and only be valid for the time intervals.
    2. The passwords used should be limited at the back end to only performing actions for the front end, as opposed to having full access.
    3. The messages sent should be tagged and checksummed with time sensitive values so as to prevent replay style attacks.

8. New Vulnerabilities

8.1 General

This clause provides language-independent descriptions of vulnerabilities under consideration for inclusion in the next edition of this International Technical Report. It is intended that revisions of these descriptions will be

incorporated into Clauses 6 and 7 of the next edition and that they will be treated in the language-specific annexes of that edition.

8.2 Terminology

The following descriptions are written in a language-independent manner except when specific languages are used in examples.

This clause will, in general, use the terminology that is most natural to the description of each individual vulnerability. Hence the terminology may differ from description to description.

8.3 Concurrency – Activation [CGA]

8.3.1 Description of application vulnerability

A vulnerability can occur if an attempt has been made to activate a thread, but a programming error or the lack of some resource prevents the activation from completing. The activating thread may not have sufficient visibility or awareness into the execution of the activated thread to determine if the activation has been successful. The unrecognized activation failure can cause a protocol failure in the activating thread or in other threads that rely upon some action by the unactivated thread. This may cause the other thread(s) to wait forever for some event from the unactivated thread, or may cause an unhandled event or exception in the other threads.

8.3.2 Cross References

CWE:

364. Signal Handler Race Condition

Hoare A., "*Communicating Sequential Processes*", Prentice Hall, 1985

Holzmann G., "*The SPIN Model Checker: Principles and Reference Manual*", Addison Wesley Professional. 2003

UPPAAL, available from www.uppaal.com,

Larsen, Peterson, Wang, "*Model Checking for Real-Time Systems*", Proceedings of the 10th International Conference on Fundamentals of Computation Theory, 1995

Ravenscar Tasking Profile, specified in ISO/IEC 8652:1995 Ada with TC 1:2001 and AM 1:2007

8.3.3 Mechanism of Failure

The context of the problem is that all threads except the main thread are activated by program steps of another thread. The activation of each thread requires that dedicated resources be created for that thread, such as a thread stack, thread attributes, and communication ports. If insufficient resources remain when the activation attempt is made, the activation will fail. Similarly, if there is a program error in the activated thread or if the activated thread detects an error that causes it to terminate before beginning its main work, then it may appear to have failed during activation. When the activation is “static”, resources have been preallocated, so activation failure because of a lack of resources will not occur. However errors may occur for reasons other than resource allocation and the results of an activation failure will be similar.

If the activating thread waits for each activated thread, then the activating thread will likely be notified of activation failures (if the particular construct or capability supports activation failure notification) and can be

programmed to take alternate action. If notification occurs but alternate action is not programmed, then the program will execute erroneously. If the activating thread is loosely coupled with the activated threads, and the activating thread does not receive notification of a failure to activate, then it may wait indefinitely for the unactivated thread to do its work, or may make wrong calculations because of incomplete data.

Activation of a single thread is a special case of activations of collections of threads simultaneously. This paradigm (activation of collections of threads) can be used in languages that parallelise calculations and create anonymous threads to execute each slice of data. In such situations the activating thread is unlikely to individually monitor each activated thread, so a failure of some to activate without explicit notification to the activating thread can result in erroneous calculations.

If the rest of the application is unaware that an activation has failed, an incorrect execution of the application algorithm may occur, such as deadlock of threads waiting for the activated thread, or possibly causing errors or incorrect calculations.

8.3.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- All languages that permit concurrency within the language, or that use support libraries and operating systems (such as POSIX or Windows) that provide concurrency control mechanisms. In essence all traditional languages on fully functional operating systems (such as POSIX-compliant OS or Windows) can access the OS-provided mechanisms.

8.3.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Always check return codes on operating system command, library provided or language thread activation mechanisms.
- Handle errors and exceptions that occur on activation.
- Create explicit synchronization protocols, to ensure that all activations have occurred before beginning the parallel algorithm, if not provided by the language or by the threading subsystem.
- Use programming language provided features that couple the activated thread with the activating thread to detect activation errors so that errors can be reported and recovery made.
- Use static activation in preference to dynamic activation so that static analysis can guarantee correct activation of threads.

8.3.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Consider including automatic synchronization of thread initiation as part of the concurrency model.
- Provide a mechanism permitting query of activation success.

8.4 Concurrency – Directed termination [CGT]

8.4.1 Description of application vulnerability

This discussion is associated with the effects of unsuccessful or late termination of a thread. For a discussion of premature termination, see [CGS] Concurrency – Premature Termination.

When a thread is working cooperatively with other threads and is directed to terminate, there are a number of error situations that may occur that can lead to compromise of the system. The termination directing thread may request that one or more other threads abort or terminate, but the terminated thread(s) may not be in a state such that the termination can occur, may ignore the direction, or may take longer to abort or terminate than the application can tolerate. In any case, on most systems, the thread will not terminate until it is next scheduled for execution.

Unexpectedly delayed termination or the consumption of resources by the termination itself may cause a failure to meet deadlines, which, in turn, may lead to other failures.

8.4.2 Cross references

CWE:

364. Signal Handler Race Condition

Hoare C.A.R., "*Communicating Sequential Processes*", Prentice Hall, 1985

Holzmann G., "*The SPIN Model Checker: Principles and Reference Manual*", Addison Wesley Professional. 2003

Larsen, Peterson, Wang, "*Model Checking for Real-Time Systems*", Proceedings of the 10th International Conference on Fundamentals of Computation Theory, 1995

The Ravenscar Tasking Profile, specified in ISO/IEC 8652:1995 Ada with TC 1:2001 and AM 1:2007

8.4.3 Mechanism of failure

The abort of a thread may not happen if a thread is in an abort-deferred region and does not leave that region (for whatever reason) after the abort directive is given. Similarly, if abort is implemented as an event sent to a thread and it is permitted to ignore such events, then the abort will not be obeyed.

The termination of a thread may not happen if the thread ignores the directive to terminate, or if the finalization of the thread to be terminated does not complete.

If the termination directing thread continues on the false assumption that termination has completed, then any sort of failure may occur.

8.4.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- All languages that permit concurrency within the language, or support libraries and operating systems (such as POSIX-compliant or Windows operating systems) that provide hooks for concurrency control.

8.4.5 Avoiding the vulnerability or mitigating its effect

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use mechanisms of the language or system to determine that aborted threads or threads directed to terminate have successfully terminated. Such mechanisms may include direct communication, runtime-level checks, explicit dependency relationships, or progress counters in shared communication code to verify progress.
- Provide mechanisms to detect and/or recover from failed termination.
- Use static analysis techniques, such as CSP or model-checking to show that thread termination is safely handled.
- Where appropriate, use scheduling models where threads never terminate.

8.4.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Provide a mechanism (either a language mechanism or a service call) to signal either another thread or an entity that can be queried by other threads when a thread terminates.

8.5 Concurrent Data Access [CGX]

8.5.1 Description of application vulnerability

Concurrency presents a significant challenge to program correctly, and has a large number of possible ways for failures to occur, quite a few known attack vectors, and many possible but undiscovered attack vectors. In particular, data visible from more than one thread and not protected by a sequential access lock can be corrupted by out-of-order accesses. This, in turn, can lead to incorrect computation, premature program termination, livelock, or system corruption.

8.5.2 Cross references

CWE:

214. Information Exposure Through Process Environment
362. Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
366. Race Condition Within a Thread
368. Context Switching Race Conditions
413. Improper Resource Locking
764. Multiple Locks of a Critical Resource
765. Multiple Unlocks of a Critical Resource
821. Missing Synchronization
821. Incorrect Synchronization

ISO IEC 8692 *Programming Language Ada*, with TC 1:2001 and AM 1:2007.
Burns A. and Wellings A., Language Vulnerabilities - Let's not forget Concurrency, IRTAW 14, 2009.
C.A.R Hoare, A model for communicating sequential processes, 1980

8.5.3 Mechanism of failure

Shared data can be monitored or updated directly by more than one thread, possibly circumventing any access lock protocol in operation. Some concurrent programs do not use access lock mechanisms but rely upon other mechanisms such as timing or other program state to determine if shared data can be read or updated by a thread. Regardless, direct visibility to shared data permits direct access to such data concurrently. Arbitrary behaviour of any kind can result.

8.5.4 Applicable language characteristics

The vulnerability is intended to be applicable to

- All languages that provide concurrent execution and data sharing, whether as part of the language or by use of underlying operation system facilities, including facilities such as event handlers and interrupt handlers.

8.5.5 Avoiding the vulnerability or mitigating its effect

Software developers can avoid the vulnerability or mitigate its effects in the following ways.

- Place all data in memory regions accessible to only one thread at a time.
- Use languages and those language features that provide a robust sequential protection paradigm to protect against data corruption. For example, Ada's protected objects and Java's Protected class, provide a safe paradigm when accessing objects that are exclusive to a single program.
- Use operating system primitives, such as the POSIX locking primitives for synchronization to develop a protocol equivalent to the Ada “protected” and Java “Protected” paradigm.
- Where order of access is important for correctness, implement blocking and releasing paradigms, or provide a test in the same protected region to check for correct order and generate errors if the test fails. For example, the following structure in Ada could be used to implement an enforced order.

8.5.6 Implications for standardization

In future standardisation activities, the following items should be considered:

- Languages that do not presently consider concurrency should consider creating primitives that let applications specify regions of sequential access to data. Mechanisms such as protected regions, Hoare monitors or synchronous message passing between threads result in significantly fewer resource access mistakes in a program.

Provide the possibility of selecting alternative concurrency models that support static analysis, such as one of the models that are known to have safe properties. For examples, see [9], [10], and [17].

8.6 Concurrency – Premature Termination [CGS]

8.6.1 Description of application vulnerability

When a thread is working cooperatively with other threads and terminates prematurely for whatever reason but unknown to other threads, then the portion of the interaction protocol between the terminated thread and other threads is damaged. This may result in:

- indefinite blocking of the other threads as they wait for the terminated thread if the interaction protocol was synchronous;
- other threads receiving wrong or incomplete results if the interaction was asynchronous; or
- deadlock if all other threads were depending upon the terminated thread for some aspect of their computation before continuing.

8.6.2 Cross references

CWE:

364. Signal Handler Race Condition

Hoare C.A.R., "*Communicating Sequential Processes*", Prentice Hall, 1985
Holzmann G., "*The SPIN Model Checker: Principles and Reference Manual*", Addison Wesley Professional. 2003
Larsen, Peterson, Wang, "*Model Checking for Real-Time Systems*", Proceedings of the 10th International Conference on Fundamentals of Computation Theory, 1995
The Ravenscar Tasking Profile, specified in ISO/IEC 8652:1995 Ada with TC 1:2001 and AM 1:2007

8.6.3 Mechanism of failure

If a thread terminates prematurely, threads that depend upon services from the terminated thread (in the sense of waiting exclusively for a specific action before continuing) may wait forever since held locks may be left in a locked state resulting in waiting threads never being released or messages or events expected from the terminated thread will never be received.

If a thread depends on the terminating thread and receives notification of termination, but the dependent thread ignores the termination notification, then a protocol failure will occur in the dependent thread. For asynchronous termination events, an unexpected event may cause immediate transfer of control from the execution of the dependent thread to another (possible unknown) location, resulting in corrupted objects or resources; or may cause termination in the master thread[7].

These conditions can result in

- premature shutdown of the system;
- corruption or arbitrary execution of code;
- livelock;
- deadlock;

[7] This may cause the failure to propagate to other threads.

depending upon how other threads handle the termination errors.

If the thread termination is the result of an abort and the abort is immediate, there is nothing that can be done within the aborted thread to prepare data for return to master tasks, except possibly the management thread (or operating system) notifying other threads that the event occurred. If the aborted thread was holding resources or performing active updates when aborted, then any direct access by other threads to such locks, resources or memory may result in corruption of those threads or of the complete system, up to and including arbitrary code execution.

8.6.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages with the following characteristics:

- Languages that permit concurrency within the language, or support libraries and operating systems (such as POSIX-compliant or Windows operating systems) that provide hooks for concurrency control.

8.6.5 Avoiding the vulnerability or mitigating its effect

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use concurrency mechanisms that are known to be robust.
- At appropriate times use mechanisms of the language or system to determine that necessary threads are still operating. Such mechanisms may be direct communication, runtime-level checks, explicit dependency relationships, or progress counters in shared communication code to verify progress.
- Handle events and exceptions from termination.
- Provide manager threads to monitor progress and to collect and recover from improper terminations or abortions of threads.
- Use static analysis techniques, such as model checking, to show that thread termination is safely handled.

8.6.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Provide a mechanism to preclude the abort of a thread from another thread during critical pieces of code. Some languages (for example, Ada or Real-Time Java) provide a notion of an abort-deferred region.
- Provide a mechanism to signal another thread (or an entity that can be queried by other threads) when a thread terminates.
- Provide a mechanism that, within critical pieces of code, defers the delivery of asynchronous exceptions or asynchronous transfers of control.

8.7 Protocol Lock Errors [CGM]

8.7.1 Description of application vulnerability

Concurrent programs use protocols to control

- The way that threads interact with each other,
- How to schedule the relative rates of progress,
- How threads participate in the generation and consumption of data,
- The allocation of threads to the various roles,
- The preservation of data integrity, and
- The detection and correction of incorrect operations.

When protocols are not correct, or when a vulnerability lets an exploit destroy a protocol, then the concurrent portions fail to work co-operatively and the system behaves incorrectly.

This vulnerability is related to [CGX] Shared Data Access and Corruption, which discusses situations where the protocol to control access to resources is explicitly visible to the participating partners and makes use of visible shared resources. In comparison, this vulnerability discusses scenarios where such resources are protected by protocols, and considers ways that the protocol itself may be misused.

8.7.2 Cross references

CWE:

413. Improper Resource Locking
414. Missing Lock Check
609. Double Checked Locking
667. Improper Locking
821. Incorrect Synchronization
833. Deadlock

C.A.R. Hoare, A model for communicating sequential processes, 1980

Larsen, K.G., Petterssen, P, Wang, Y, UPPAAL in a nutshell, 1997

8.7.3 Mechanism of failure

Threads use locks and protocols to schedule their work, control access to resources, exchange data, and to effect communication with each other. Protocol errors occur when the expected rules for co-operation are not followed, or when the order of lock acquisitions and release causes the threads to quit working together. These errors can be as a result of:

- deliberate termination of one or more threads participating in the protocol,
- disruption of messages or interactions in the protocol,
- errors or exceptions raised in threads participating in the protocol, or
- errors in the programming of one or more threads participating in the protocol.

In such situations, there are a number of possible consequences:

- *deadlock*, where every thread eventually quits computing as it waits for results from another thread, no further progress in the system is made,
- *livelock*, where one or more threads commandeer all of the computing resource and effectively lock out the other portions, no further progress in the system is made,
- data may be corrupted or lack currency (timeliness), or

- one or more threads detect an error associated with the protocol and terminate prematurely, leaving the protocol in an unrecoverable state.

The potential damage from attacks on protocols depends upon the nature of the system using the protocol and the protocol itself. Self-contained systems using private protocols can be disrupted, but it is highly unlikely that predetermined executions (including arbitrary code execution) can be obtained. On the other extreme, threads communicating openly between systems using well-documented protocols can be disrupted in any arbitrary fashion with effects such as the destruction of system resources (such as a database), the generation of wrong but plausible data, or arbitrary code execution. In fact, many documented client-server based attacks consist of some abuse of a protocol such as SQL transactions.

8.7.4 Applicable language characteristics

The vulnerability is intended to be applicable to languages with the following characteristics:

- Languages that support concurrency directly.
- Languages that permit calls to operating system primitives to obtain concurrent behaviours.
- Languages that permit IO or other interaction with external devices or services.
- Languages that support interrupt handling directly or indirectly (via the operating system).

8.7.5 Avoiding the vulnerability or mitigating its effect

Software developers can avoid the vulnerability or mitigate its effects in the following ways:

- Consider the use of synchronous protocols, such as defined by CSP, Petri Nets or by the Ada rendezvous protocol since these can be statically shown to be free from protocol errors such as deadlock and livelock.
- Consider the use of simple asynchronous protocols that exclusively use concurrent threads and protected regions, such as defined by the Ravenscar Tasking Profile, which can also be shown statically to have correct behaviour using model checking technologies, as shown by [46].
- When static verification is not possible, consider the use of detection and recovery techniques using simple mechanisms and protocols that can be verified independently from the main concurrency environment. Watchdog timers coupled with checkpoints constitute one such approach.
- Use high-level synchronization paradigms, for example monitors, rendezvous, or critical regions.
- Design the architecture of the application to ensure that some threads or tasks never block, and can be available for detection of concurrency error conditions and for recovery initiation.
- Use model checkers to model the concurrent behaviour of the complete application and check for states where progress fails. Place all locks and releases in the same subprograms, and ensure that the order of calls and releases of multiple locks are correct.

8.7.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Raise the level of abstraction for concurrency services.
- Provide services or mechanisms to detect and recover from protocol lock failures.

- Design concurrency services that help to avoid typical failures such as deadlock.

8.8 Inadequately Secure Communication of Shared Resources [CGY]

8.8.1 Description of application vulnerability

A resource that is directly visible from more than one process (at the same approximate time) and is not protected by access locks can be hijacked or used to corrupt, control or change the behaviour of other processes in the system. Many vulnerabilities that are associated with concurrent access to files, shared memory or shared network resources fall under this vulnerability, including resources accessed via stateless protocols such as HTTP and remote file protocols.

8.8.2 Cross references

CWE:

15. External Control of System or Configuration Setting

642. External Control of Critical State Data

Burns A. and Wellings A., Language Vulnerabilities - Let's not forget Concurrency, IRTAW 14, 2009.

8.8.3 Mechanism of failure

Any time that a shared resource is open to general inspection, the resource can be monitored by a foreign process to determine usage patterns, timing patterns, and access patterns to determine ways that a planned attack can succeed[8]. Such monitoring could be, but is not limited to:

- Reading resource values to obtain information of value to the applications.
- Monitoring access time and access thread to determine when a resource can be accessed undetected by other threads (for example, Time-of-Check-Time-Of-Use attacks rely upon a determinable amount of time between the check on a resource and the use of the resource when the resource could be modified to bypass the check).
- Monitoring a resource and modification patterns to help determine the protocols in use.
- Monitoring access times and patterns to determine quiet times in the access to a resource that could be used to find successful attack vectors.

This monitoring can then be used to construct a successful attack, usually in a later attack.

Any time that a resource is open to general update, the attacker can plan an attack by performing experiments to:

- Discover how changes affect patterns of usage, timing, and access.
- Discover how application threads detect and respond to forged values.

[8] Such monitoring is almost always possible by a process executing with system privilege, but even small slips in access controls and permissions let such resources be seen from other (non system level) processes. Even the existence of the resource, its size, or its access dates/times and history (such as "last accessed time") can give valuable information to an observer.

Any time that a shared resource is open to shared update by a thread, the resource can be changed in ways to further an attack once it is initiated. For example, in a well-known attack, a process monitors a certain change to a known file and then immediately replaces a virus free file with an infected file to bypass virus checking software.

With careful planning, similar scenarios can result in the foreign process determining a weakness of the attacked process leading to an exploit consisting of anything up to and including arbitrary code execution.

8.8.4 Avoiding the vulnerability or mitigating its effect

Software developers can avoid the vulnerability or mitigate its effects in the following ways.

- Place all shared resources in memory regions accessible to only one process at a time.
- Protect resources that must be visible with encryption or with checksums to detect unauthorized modifications.
- Protect access to shared resources using permissions, access control, or obfuscation.
- Have and enforce clear rules with respect to permissions to change shared resources.
- Detect attempts to alter shared resources and take immediate action.

Annex A
(*informative*)
Vulnerability Taxonomy and List

A.1 General

This Technical Report is a catalog that will continue to evolve. For that reason, a scheme that is distinct from sub-clause numbering has been adopted to identify the vulnerability descriptions. Each description has been assigned an arbitrarily generated, unique three-letter code. These codes should be used in preference to sub-clause numbers when referencing descriptions because they will not change as additional descriptions are added to future editions of this Technical Report. However, it is recognized that readers may need assistance in locating descriptions of interest.

This annex provides a taxonomical hierarchy of vulnerabilities, which users may find to be helpful in locating descriptions of interest. A.2 is a taxonomy of the programming language vulnerabilities described in Clause 6 and A.3 is a taxonomy of the application vulnerabilities described in Clause 7. A.4 lists the vulnerabilities in the alphabetical order of their three-letter codes and provides a cross-reference to the relevant sub-clause.

A.2 Outline of Programming Language Vulnerabilities

A.2.1. Types
- A.2.1.1. Representation
  - A.2.1.1.1. [IHN] Type System
  - A.2.1.1.2. [STR] Bit Representations
- A.2.1.2. Floating-point
  - A.2.1.2.1. [PLF] Floating-point Arithmetic
- A.2.1.3. Enumerated Types
  - A.2.1.3.1. [CCB] Enumerator Issues
- A.2.1.4. Integers
  - A.2.1.4.1. [FLC] Numeric Conversion Errors
- A.2.1.5. Characters and strings
  - A.2.1.5.1 [CJM] String Termination
- A.2.1.6. Arrays
  - A.2.1.6.1. [HCB] Buffer Boundary Violation (Buffer Overflow)
  - A.2.1.6.2. [XYZ] Unchecked Array Indexing
  - A.2.1.6.3. [XYW] Unchecked Array Copying
- A.2.1.7. Pointers
  - A.2.1.7.1. [HFC] Pointer Casting and Pointer Type Changes
  - A.2.1.7.2. [RVG] Pointer Arithmetic
  - A.2.1.7.3. [XYH] Null Pointer Dereference
  - A.2.1.7.4. [XYK] Dangling Reference to Heap

A.2.2. Type Conversions/Limits
- A.2.2.1. [FIF] Arithmetic Wrap-around Error
- A.2.2.1 [PIK] Using Shift Operations for Multiplication and Division
- A.2.2.2. [XZI] Sign Extension Error

A.2.3. Declarations and Definitions
- A.2.3.1. [NAI] Choice of Clear Names
- A.2.3.2. [WXQ] Dead store

A.3 Outline of Application Vulnerabilities

A.4 Vulnerability List

Code	Vulnerability Name	Sub-clause	Page
[AMV]	Type-breaking Reinterpretation of Data	6.40	85
[BJL]	Namespace Issues	6.23	57
[BQF]	Unspecified Behaviour	6.54	105
[BRS]	Obscure Language Features	6.53	104
[BVQ]	Unspecified Functionality	7.3	112
[CBF]	Unrestricted File Upload	7.10	120

[CCB]	Enumerator Issues	6.6	32
[CGA]	Concurrency - Activation	8.3	139
[CGM]	Protocol Lock Errors	8.7	145
[CGS]	Concurrency - Premature Termination	8.6	144
[CGT]	Concurrency - Directed termination	8.4	141
[CGX]	Concurrent Data Access	8.5	142
[CGY]	Inadequately Secure Communication of Shared Resources	8.8	148
[CJM]	String Termination	6.8	36
[CLL]	Switch Statements and Static Analysis	6.29	68
[CSJ]	Passing Parameters and Return Values	6.34	74
[DCM]	Dangling References to Stack Frames	6.35	77
[DJS]	Inter-language Calling	6.46	95
[EOJ]	Demarcation of Control Flow	6.30	69
[EWD]	Structured Programming	6.33	73
[EWF]	Undefined Behaviour	6.55	107
[EWR]	Path Traversal	7.18	131
[FAB]	Implementation-defined Behaviour	6.56	108
[FIF]	Arithmetic Wrap-around Error	6.16	47
[FLC]	Numeric Conversion Errors	6.7	34
[GDL]	Recursion	6.37	80
[HCB]	Buffer Boundary Violation (Buffer Overflow)	6.9	37
[HFC]	Pointer Casting and Pointer Type Changes	6.12	42
[HJW]	Unanticipated Exceptions from Library Routines	6.49	99
[HTS]	Resource Names	7.11	121
[IHN]	Type System	6.3	26
[JCW]	Operator Precedence/Order of Evaluation	6.25	61
[KLK]	Distinguished Values in Data Types	7.4	113
[KOA]	Likely Incorrect Expression	6.27	64
[LAV]	Initialization of Variables	6.24	59
[LRM]	Extra Intrinsics	6.44	92
[MEM]	Deprecated Language Features	6.57	110
[MXB]	Suppression of Language-defined Run-time Checking	6.51	102
[NAI]	Choice of Clear Names	6.19	51
[NMP]	Pre-processor Directives	6.50	100
[NSQ]	Library Signature	6.48	98
[NYY]	Dynamically-linked Code and Self-modifying Code	6.47	97
[OTR]	Subprogram Signature Mismatch	6.36	79
[OYB]	Ignored Error Status and Unhandled Exceptions	6.38	82
[PIK]	Using Shift Operations for Multiplication and Division	6.17	49
[PLF]	Floating-point Arithmetic	6.5	30
[REU]	Termination Strategy	6.39	84
[RIP]	Inheritance	6.43	91
[RST]	Injection	7.12	123
[RVG]	Pointer Arithmetic	6.13	43
[SAM]	Side-effects and Order of Evaluation	6.26	62
[SKL]	Provision of Inherently Unsafe Operations	6.52	103
[STR]	Bit Representations	6.4	28
[SYM]	Templates and Generics	6.42	89
[TEX]	Loop Control Variables	6.31	70
[TRJ]	Argument Passing to Library Functions	6.45	93
[WXQ]	Dead Store	6.20	53
[XYH]	Null Pointer Dereference	6.14	44
[XYK]	Dangling Reference to Heap	6.15	45
[XYL]	Memory Leak	6.41	87

[XYM]	Insufficiently Protected Credentials	7.20	134
[XYN]	Adherence to Least Privilege	7.5	114
[XYO]	Privilege Sandbox Issues	7.6	115
[XYP]	Hard-coded Password	7.23	137
[XYQ]	Dead and Deactivated Code	6.28	66
[XYS]	Executing or Loading Untrusted Code	7.7	117
[XYT]	Cross-site Scripting	7.13	126
[XYW]	Unchecked Array Copying	6.11	41
[XYZ]	Unchecked Array Indexing	6.10	39
[XZH]	Off-by-one Error	6.32	72
[XZI]	Sign Extension Error	6.18	50
[XZK]	Sensitive Information Uncleared Before Use	7.17	131
[XZL]	Discrepancy Information Leak	7.16	130
[XZN]	Missing or Inconsistent Access Control	7.21	135
[XZO]	Authentication Logic Error	7.22	136
[XZP]	Resource Exhaustion	7.9	119
[XZQ]	Unquoted Search Path or Element	7.14	128
[XZR]	Improperly Verified Signature	7.15	129
[XZS]	Missing Required Cryptographic Step	7.19	134
[XZX]	Memory Locking	7.8	118
[YOW]	Identifier Name Reuse	6.22	55
[YZS]	Unused Variable	6.21	54

Annex B
(*informative*)
Language Specific Vulnerability Template

Each language-specific annex should have the following heading information and initial sections:

Annex <language>
(*Informative*)
Vulnerability descriptions for language <language>

<language>.1 Identification of standards

[This sub-clause should list the relevant language standards and other documents that describe the language treated in the annex. It need not be simply a list of standards. It should do whatever is required to describe the language that is the baseline.]

<language>.2 General terminology and concepts

[This sub-clause should provide an overview of general terminology and concepts that are utilized throughout the annex.]

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

<language>.<x> <Vulnerability Name> [<3 letter tag>]

<language>.<x>.0 Status, history, and bibliography

[Revision history. This clause will eventually be removed.]

<language>.<x>.1 Applicability to language

[This section describes what the language does or does not do in order to deal with the vulnerability.]

<language>.<x>.2 Guidance to language users

[This section describes what the programmer or user should do regarding the vulnerability.]

In those cases where a vulnerability is simply not applicable to the language, the following format should be used instead:

<language>.<x> <Vulnerability Name> [<3 letter tag>]

This vulnerability is not applicable to <language>.

Following the final vulnerability description, there should be a single sub-clause as follows:

<language>.<x> Implications for standardization

[This section provides the opportunity to discuss changes anticipated for future versions of the language specification.]

Annex C
(*informative*)
Vulnerability descriptions for the language Ada

C.1 Identification of standards and associated documentation

ISO/IEC 8652:1995 Information Technology – Programming Languages—Ada.
ISO/IEC 8652:1995/COR.1:2001, Technical Corrigendum to Information Technology – Programming Languages—Ada.
ISO/IEC 8652:1995/AMD.1:2007, Amendment to Information Technology – Programming Languages—Ada.
ISO/IEC TR 15942:2000, Guidance for the Use of Ada in High Integrity Systems.
ISO/IEC TR 24718:2005, Guide for the use of the Ada Ravenscar Profile in high integrity systems.
Lecture Notes on Computer Science 5020, "Ada 2005 Rationale: The Language, the Standard Libraries," John Barnes, Springer, 2008.
Ada 95 Quality and Style Guide, SPC-91061-CMC, version 02.01.01. Herndon, Virginia: Software Productivity Consortium, 1992.
Ada Language Reference Manual, The consolidated Ada Reference Manual, consisting of the international standard (ISO/IEC 8652:1995): *Information Technology -- Programming Languages -- Ada*, as updated by changes from *Technical Corrigendum 1* (ISO/IEC 8652:1995:TC1:2000), and Amendment 1 (ISO/IEC 8526:AMD1:2007).
IEEE 754-2008, IEEE Standard for Binary Floating Point Arithmetic, IEEE, 2008.
IEEE 854-1987, IEEE Standard for Radix-Independent Floating-Point Arithmetic, IEEE, 1987

C.2 General terminology and concepts

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

Access object: An object of an access type.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Discrete type: An integer type or an enumeration type.

Discriminant: A parameter for a composite type. It can control, for example, the bounds of a component of the type if the component is an array. A discriminant for a task type can be used to pass data to a task of the type upon creation.

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

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

Enumeration Type: An enumeration type is a discrete type defined by an enumeration of its values, which may be named by identifiers or character literals. In Ada, the types Character and Boolean are enumeration types. The defining identifiers and defining character literals of an enumeration type must be distinct. The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.

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

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

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

Explicit Conversion: The Ada term explicit conversion is equivalent to the term cast in Section 6.3.3.

Fixed-point types: Real-valued types with a specified error bound (called the 'delta' of the type) that provide arithmetic operations carried out with fixed precision (rather than the relative precision of floating-point types).

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

Hiding: A declaration can be *hidden*, either from direct visibility, or from all visibility, within certain parts of its scope. Where *hidden from all visibility*, it is not visible at all (neither using a direct_name nor a selector_name). Where *hidden from direct visibility*, only direct visibility is lost; visibility using a selector_name is still possible.

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

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

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

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

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

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

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

a) Implicit conversions: there are few situations in Ada that allow for implicit conversions. An example is the assignment of a value of a type to a polymorphic variable of an encompassing

class. In all cases where implicit conversions are permitted, neither static nor dynamic type safety or application type semantics (see below) are endangered by the conversion.

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

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

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

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

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

Operational and Representation Attributes: The values of certain implementation-dependent characteristics can be obtained by querying the applicable attributes. Some attributes can be specified by the user; for example:

- X'Alignment: allows the alignment of objects on a storage unit boundary at an integral multiple of a specified value.
- X'Size: denotes the size in bits of the representation of the object.
- X'Component_Size: denotes the size in bits of components of the array type X.

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

Partition: A partition is a part of a program. Each partition consists of a set of library units. Each partition may run in a separate address space, possibly on a separate computer. A program may contain just one partition. A distributed program typically contains multiple partitions, which can execute concurrently.

Pointer: Synonym for “access object.”

Pragma: A directive to the compiler.

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

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

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

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

Pragma Discard_Names: Specifies that storage used at run-time for the names of certain entities may be reduced.

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

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

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

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

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

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

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

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

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

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

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

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

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

Storage Pool: A named location in an Ada program where all of the objects of a single access type will be allocated. A storage pool can be sized exactly to the requirements of the application by allocating only what is

needed for all objects of a single type without using the centrally managed heap. Exceptions raised due to memory failures in a storage pool will not adversely affect storage allocation from other storage pools or from the heap. Storage pools for types whose values are of equal length do not suffer from fragmentation.

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

> **pragma** Restrictions(No_Allocators): prevents the use of allocators.
>
> **pragma** Restrictions(No_Local_Allocators): prevents the use of allocators after the main program has commenced.
>
> **pragma** Restrictions(No_Implicit_Heap_Allocations): prevents the use of allocators that would use the heap, but permits allocations from storage pools.

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

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

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

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

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

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

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

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

C.3 Type System [IHN]

C.3.1 Applicability to language

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

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

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

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

Unchecked conversions circumvent the type system and therefore can cause unspecified behaviour (see C.40 [AMV]).

C.3.2 Guidance to language users

- The predefined 'Valid attribute for a given subtype may be applied to any value to ascertain if the value is a valid value of the subtype. This is especially useful when interfacing with type-less systems or after Unchecked_Conversion.
- A conceivable measure to prevent incorrect unit conversions is to restrict explicit conversions to the bodies of user-provided conversion functions that are then used as the only means to effect the transition between unit systems. These bodies are to be critically reviewed for proper conversion factors.
- Exceptions raised by type and subtype conversions shall be handled.

C.4 Bit Representation [STR]

C.4.1 Applicability to language

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.

The vulnerabilities caused by the inherent conceptual complexity of bit level programming are as described in Section 6.4.

C.4.2 Guidance to language users

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

- The use of record and array types with the appropriate representation specifications added so that the objects are accessed by their logical structure rather than their physical representation. These representation specifications may address: order, position, and size of data components and fields.
- The use of pragma Atomic and **pragma** Atomic_Components to ensure that all updates to objects and components happen atomically.
- The use of pragma Volatile and **pragma** Volatile_Components to notify the compiler that objects and components must be read immediately before use as other devices or systems may be updating them between accesses of the program.

- The default object layout chosen by the compiler may be queried by the programmer to determine the expected behaviour of the final representation.

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

C.5 Floating-point Arithmetic [PLF]

C.5.1 Applicability to language

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

The vulnerability in Ada is as described in Section 6.5.2.

C.5.2 Guidance to language users

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

C.6 Enumerator Issues [CCB]

C.6.1 Applicability to language

Enumeration representation specification may be used to specify non-default representations of an enumeration type, for example when interfacing with external systems. All of the values in the enumeration type must be defined in the enumeration representation specification. The numeric values of the representation must preserve the original order. For example:

```
type IO_Types is (Null_Op, Open, Close, Read, Write, Sync);

for IO_Types use (Null_Op => 0, Open => 1, Close => 2,
```

```
Read => 4, Write => 8, Sync => 16 );
```

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

C.6.2 Guidance to language users

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

C.7 Numeric Conversion Errors [FLC]

C.7.1 Applicability to language

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

In the case of explicit conversions, range bound checks are applied, so no truncation can occur, and an exception will be generated if the operand of the conversion exceeds the bounds of the target type or subtype.

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

Ada permits the definition of subtypes of existing types that can impose a restricted range of values, and implicit conversions can occur for values of different subtypes belonging to the same type, but such conversions still involve range checks that prevent any loss of data or violation of the bounds of the target subtype.

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

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

C.7.2 Guidance to language users

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

- Use range checks on conversions involving scalar types and subtypes to prevent generation of invalid data.
- Use static analysis tools during program development to verify that conversions cannot violate the range of their target.

C.8 String Termination [CJM]

With the exception of unsafe programming (see C.2), 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.

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

With the exception of unsafe programming (see C.2), this vulnerability is not applicable to Ada as this vulnerability can only happen as a consequence of unchecked array indexing or unchecked array copying (see C.10 [XYZ] and C.11 [XYW]).

C.10 Unchecked Array Indexing [XYZ]

C.10.1 Applicability to language

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

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

C.10.2 Guidance to language users

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

C.11 Unchecked Array Copying [XYW]

With the exception of unsafe programming (see C.2), 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.

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

C.12.1 Applicability to language

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

The vulnerabilities described in Section 6.12 exist in Ada only if unchecked type conversions or unsafe taking of addresses are applied (see Section C.2). Other permitted type conversions can never misrepresent the type of the designated entity.

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

C.12.2 Guidance to language users

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

C.13 Pointer Arithmetic [RVG]

With the exception of unsafe programming (see C.2), this vulnerability is not applicable to Ada as Ada does not allow pointer arithmetic.

C.14 Null Pointer Dereference [XYH]

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

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

C.15 Dangling Reference to Heap [XYK]

C.15.1 Applicability to language

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.

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

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

C.15.2 Guidance to language users

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

C.16 Arithmetic Wrap-around Error [FIF]

With the exception of unsafe programming (see C.2), this vulnerability is not applicable to Ada as wrap-around arithmetic in Ada is limited to modular types. Arithmetic operations on such types use modulo arithmetic, and thus no such operation can create an invalid value of the type.

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

C.17 Using Shift Operations for Multiplication and Division [PIK]

With the exception of unsafe programming (see C.2), 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.

C.18 Sign Extension Error [XZI]

With the exception of unsafe programming (see C.2), this vulnerability is not applicable to Ada as Ada does not, explicitly or implicitly, allow unsigned extension operations to apply to signed entities or vice-versa.

C.19 Choice of Clear Names [NAI]

C.19.1 Applicability to language

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

This vulnerability does not address overloading, which is covered in Section C.22.YOW.

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

- Mixed casing. Ada treats upper and lower case letters in names as identical. Thus no confusion can arise through an attempt to use Item and ITEM as distinct identifiers with different meanings.
- Underscores and periods. Ada permits single underscores in identifiers and they are significant. Thus BigDog and Big_Dog are different identifiers. But multiple underscores (which might be confused with a single underscore) are forbidden, thus Big__Dog is forbidden. Leading and trailing underscores are also forbidden. Periods are not permitted in identifiers at all.
- Singular/plural forms. Ada does permit the use of identifiers which differ solely in this manner such as Item and Items. However, the user might use the identifier Item for a single object of a type T and the identifier Items for an object denoting an array of items that is of a type array (…) of T. The use of Item where Items was intended or vice versa will be detected by the compiler because of the type violation and the program rejected so no vulnerability would arise.

- International character sets. Ada compilers strictly conform to the appropriate international standard for character sets.
- Identifier length. All characters in an identifier in Ada are significant. Thus Long_IdentifierA and Long_IdentifierB are always different. An identifier cannot be split over the end of a line. The only restriction on the length of an identifier is that enforced by the line length and this is guaranteed by the language standard to be no less than 200.

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

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

C.19.2 Guidance to language users

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

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

C.20 Dead store [WXQ]

C.20.1 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.

C.20.2 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.

C.21 Unused Variable [YZS]

C.21.1 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.

C.21.2 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.

C.22 Identifier Name Reuse [YOW]

C.22.1 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.

C.22.2 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.

C.23 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.

C.24 Initialization of Variables [LAV]

C.24.1 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 OYB regarding exception handling.) Thus, the only remaining vulnerability is the potential use of a faulty but subtype-

conformant value of an uninitialized variable, since it is technically indistinguishable from a value legitimately computed by the application.

For 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.

C.24.2 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.

C.25 Operator Precedence/Order of Evaluation [JCW]

C.25.1 Applicability to language

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

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

C.25.2 Guidance to language users

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

C.26 Side-effects and Order of Evaluation [SAM]

C.26.1 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.

C.26.2 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.

C.27 Likely Incorrect Expression [KOA]

C.27.1 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 (i.e., "**and then**" and "**or else**" vs. "**and**" and "**or**"), as in

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

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

C.27.2 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.

C.28 Dead and Deactivated Code [XYQ]

C.28.1 Applicability to language

Ada allows the usual sources of dead code (described in 6.28) that are common to most conventional programming languages.

C.28.2 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 (e.g., 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.

C.29 Switch Statements and Static Analysis [CLL]

C.29.1 Applicability to language

With the exception of unsafe programming (see C.2) and the use of default cases, this vulnerability is not applicable to Ada as Ada ensures that a case statement provides exactly one alternative for each value of the expression's subtype. This restriction is enforced at compile time. The **others** clause may be used as the last choice of a case statement to capture any remaining values of the case expression type that are not covered by the preceding case choices. If the value of the expression is outside of the range of this subtype (e.g., due to an uninitialized variable), then the resulting behaviour is well-defined (Constraint_Error is raised). Control does not flow from one alternative to the next. Upon reaching the end of an alternative, control is transferred to the end of the **case** statement.

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

C.29.2 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.[9]

C.30 Demarcation of Control Flow [EOJ]

With the exception of unsafe programming (see C.2), 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.

C.31 Loop Control Variables [TEX]

With the exception of unsafe programming (see C.2), 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.

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

C.32 Off-by-one Error [XZH]

C.32.1 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.

C.32.2 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.

C.33 Structured Programming [EWD]

C.33.1 Applicability to language

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

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

C.33.2 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.

C.34 Passing Parameters and Return Values [CSJ]

C.34.1 Applicability to language

Ada employs the mechanisms (e.g., 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.

C.34.2 Guidance to language users

- Follow avoidance advice in Section 6.34.

C.35 Dangling References to Stack Frames [DCM]

C.35.1 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.

C.35.2 Guidance to language users

- Only use 'Address attribute on static objects (e.g., a register address).
- Do not use 'Address to provide indirect untyped access to an object.
- Do not use conversion between Address and access types.
- Use access types in all circumstances when indirect access is needed.
- Do not suppress accessibility checks.
- Avoid use of the attribute Unchecked_Access.
- Use ‘Access attribute in preference to ‘Address.

C.36 Subprogram Signature Mismatch [OTR]

C.36.1 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.

C.36.2 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.

C.37 Recursion [GDL]

C.37.1 Applicability to language

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

C.37.2 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.

C.38 Ignored Error Status and Unhandled Exceptions [OYB]

C.38.1 Applicability to language

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

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

C.38.2 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.

C.39 Termination Strategy [REU]

C.39.1 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.

C.39.2 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.

C.40 Type-breaking Reinterpretation of Data [AMV]

C.40.1 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.

C.40.2 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.

C.41 Memory Leak [XYL]

C.41.1 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.

C.41.2 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.

C.42 Templates and Generics [SYM]

With the exception of unsafe programming (see C.2), 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.

C.43 Inheritance [RIP]

C.43.1 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.

C.43.2 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.

C.44 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.

C.45 Argument Passing to Library Functions [TRJ]

C.45.1 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.

C.45.2 Guidance to language users

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

C.46 Inter-language Calling [DJS]

C.46.1 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.

C.46.2 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 artefacts, 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.

C.47 Dynamically-linked Code and Self-modifying Code [NYY]

With the exception of unsafe programming (see C.2), 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.

C.48 Library Signature [NSQ]

C.48.1 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.

C.48.2 Guidance to language users

- The mitigation mechanisms of Section 6.48.5 are applicable.

C.49 Unanticipated Exceptions from Library Routines [HJW]

C.49.1 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.

C.49.2 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.

C.50 Pre-Processor Directives [NMP]

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

C.51 Suppression of Language-defined Run-time Checking [MXB]

C.51.1 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.

C.51.2 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.

C.52 Provision of Inherently Unsafe Operations [SKL]

C.52.1 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.

C.53 Obscure Language Features [BRS]

C.53.1 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.

C.53.2 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.

C.54 Unspecified Behaviour [BQF]

C.54.1 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.

C.54.2 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.

C.55 Undefined Behaviour [EWF]

C.55.1 Applicability to language

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

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

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

The full list is given in the index of the Ada Standard under *erroneous execution*.

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

C.55.2 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.

C.56 Implementation-Defined Behaviour [FAB]

C.56.1 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.

C.56.2 Guidance to language users

Many implementation-defined limits have associated constants declared in language-defined packages, generally **package** System. In particular, the maximum range of integers is given by System.Min_Int .. System.Max_Int, and other limits are indicated by constants such as System.Max_Binary_Modulus, System.Memory_Size, System.Max_Mantissa, etc. Other implementation-defined limits are implicit in normal 'First and 'Last attributes of language-defined (sub) types, such as System.Priority'First and System.Priority'Last. Furthermore, the implementation-defined representation aspects of types and subtypes can be queried by language-defined attributes. Thus, code can be parameterized to adjust to implementation-defined properties without modifying the code.

- Programmers should be aware of the contents of Annex M of the Ada Standard and avoid implementation-defined behaviour whenever possible.
- Programmers should make use of the constants and subtype attributes provided in package System and elsewhere to avoid exceeding implementation-defined limits.
- Programmers should minimize use of any predefined numeric types, as the ranges and precisions of these are all implementation defined. Instead, they should declare their own numeric types to match their particular application needs.

- When there are implementation-defined formats for strings, such as Exception_ Information, any necessary processing should be localized in packages with implementation-specific variants.

C.57 Deprecated Language Features [MEM]

C.57.1 Applicability to language

If obsolescent language features are used, then the mechanism of failure for the vulnerability is as described in Section 6.57.3.

C.57.2 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.

C.58 Implications for standardization

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

- Some languages (e.g., Java) require that all local variables either be initialized at the point of declaration or on all paths to a reference. Such a rule could be considered for Ada (see C.24 [LAV]).
- **Pragma** Restrictions could be extended to allow the use of these features to be statically checked (see C.33 [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 C.54 [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 C.55 [EWF]).
- Language standards should specify relatively tight boundaries on implementation-defined behaviour whenever possible, and the standard should highlight what levels represent a portable minimum capability on which programmers may rely. For languages like Ada that allow user declaration of numeric types, the number of predefined numeric types should be minimized (for example, strongly discourage or disallow declarations of Byte_Integer, Very_Long_Integer, etc., in **package** Standard) (see C.56 [FAB]).
- Ada could define a **pragma** Restrictions identifier No_Hiding that forbids the use of a declaration that result in a local homograph (see C.22 [YOW]).
- Add the ability to declare in the specification of a function that it is pure, i.e., it has no side effects (see C.26 [SAM]).
- **Pragma** Restrictions could be extended to restrict the use of 'Address attribute to library level static objects (see C.35 [DCM]).
- Future standardization of Ada should consider implementing a language-provided reference counting storage management mechanism for dynamic objects (see C.41 [XYL]).
- Provide mechanisms to prevent further extensions of a type hierarchy (see C.43 [RIP]).
- Future standardization of Ada should consider support for arbitrary pre- and postconditions (see C.45 [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 C.48 [NSQ]).

Annex D
(*informative*)
Vulnerability descriptions for the language C

D.1 Identification of standards and associated documents

ISO/IEC 9899:2011 — *Programming Languages—C*
ISO/IEC TR 24731-1:2007 — *Extensions to the C library — Part 1: Bounds-checking interfaces*
ISO/IEC TR 24731-2:2010 — *Extensions to the C library — Part 2: Dynamic Allocation Functions*
ISO/IEC 9899:1999/Cor. 1:2001 — *Programming languages —C*
ISO/IEC 9899:1999/Cor. 2:2004 — *Programming languages —C*
ISO/IEC 9899:1999/Cor. 3:2007 — *Programming languages —C*
GNU Project. GCC Bugs "Non-bugs" http://gcc.gnu.org/bugs.html#nonbugs_c (2009).

D.2 General terminology and concepts

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[10] 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 {
```

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

```
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.

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.

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`.

D.3 Type System [IHN]

D.3.1 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;
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;
```

most 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.

D.3.2 Guidance to language users

- 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.

D.4Bit Representations [STR]

D.4.1 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,

```
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.

D.4.2 Guidance to language users

- 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
        …
```

D.5Floating-point Arithmetic [PLF]

D.5.1 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:

```
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.

D.5.2 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.

D.6Enumerator Issues [CCB]

D.6.1 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`, etc. 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++){
     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`.

D.6.2 Guidance to language users

- 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.
- 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;
```

D.7Numeric Conversion Errors [FLC]

D.7.1 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.

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.

D.7.2 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.

D.8String Termination [CJM]

D.8.1 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.

D.8.2 Guidance to language users

- Use safer and more secure functions for string handling from the ISO TR24731-1, Extensions to the C library – *Part 1: Bounds-checking interfaces*[11] or the ISO TR24731-2 — *Part II: Dynamic allocation functions*. Both of these Technical Reports define alternative string handling library functions to the existing 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.

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

D.9.1 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];
}
```

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];
      }
```

[11] Currently this is an optionally normative annex in the WG 14 working draft.

```
}
```

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.

D.9.2 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 alternative functions as specified in ISO/IEC TR 24731-1:2007 or TR 24731-2:2010. These Technical Reports provides alternative functions for the C Library (as defined in ISO/IEC 9899:1999) that promotes safer, more secure programming. 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, the functions in ISO/IEC TR 24731-1:2007 are re-entrant: they never return pointers to static objects owned by the function. ISO/IEC TR 24731-1:2007 also contains functions that address insecurities with the C input-output facilities.

D.10 Unchecked Array Indexing [XYZ]

D.10.1 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).

D.10.2 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 safer and more secure functions for string handling from the ISO TR24731-1, Extensions to the C library—- Part 1: Bounds-checking interfaces. These are alternative string handling library functions to the existing 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.

D.11 Unchecked Array Copying [XYW]

D.11.1 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 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.

D.11.2 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.

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

D.12.1 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.

D.12.2 Guidance to language users

- 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.

D.13 Pointer Arithmetic [RVG]

D.13.1 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:

```
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.

D.13.2 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.

D.14 Null Pointer Dereference [XYH]

D.14.1 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.

D.14.2 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.

D.15 Dangling Reference to Heap [XYK]

D.15.1 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`.

D.15.2 Guidance to language users

- 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.

D.16 Arithmetic Wrap-around Error [FIF]

D.16.1 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;
```

```
}
```

D.16.2 Guidance to language users

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

```
a + b    a – b    a * b    a++   a--a += b
a -= ba *= ba << ba >> 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.

D.17 Using Shift Operations for Multiplication and Division [PIK]

D.17.1 Applicability to language

The issues for C are well defined in the main body of this document in [PIK]. Also see, D.16.

D.17.2 Guidance to language users

The guidance for C users is well defined in the main body of this document in [PIK]. Also see, D.16.

D.18 Sign Extension Error [XZI]

Does not apply to C, since instead of conversion routines, C uses direct casts and implicit conversions. This allows the compiler to pick the correct signedness.

D.19 Choice of Clear Names [NAI]

D.19.1 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 allow 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 behaviour.

D.19.2 Guidance to language users

- 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.

D.20 Dead Store [WXQ]

D.20.1 Applicability to Language

Because C is an imperative language, programs in C can contain dead stores. This can result from an error in the initial design or implementation of a program, or from an incomplete or erroneous modification of an existing program.

A store into a volatile-qualified variable generally should not be considered a dead store because accessing such a variable may cause additional side effects, such as input/output (memory-mapped I/O) or observability by a debugger or another thread of execution.

D.20.2 Guidance to Language Users

- Use compilers and analysis tools to identify dead stores in the program.
- Declare variables as volatile when they are intentional targets of a store whose value does not appear to be used.

D.21 Unused Variable [YZS]

D.21.1 Applicability to language

Variables may be declared, but never used when writing code or the need for a variable may be eliminated in the code, but the declaration may remain. Most compilers will report this as a warning and the warning can be easily resolved by removing the unused variable.

D.21.2 Guidance to language users

- Resolve all compiler warnings for unused variables. This is trivial in C as one simply needs to remove the declaration of the variable. Having an unused variable in code indicates that either warnings were turned off during compilation or were ignored by the developer.

D.22 Identifier Name Reuse [YOW]

D.22.1 Applicability to language

C allows scoping so that a variable that is not declared locally may be resolved to some outer block and that resolution may cause the variable to operate on an entity other than the one intended.

Because the variable name `var1` was reused in the following example, the printed value of `var1` may be unexpected.

```
int var1;          /* declaration in outer scope */
var1 = 10;
{
      int var2;
      int var1;        /* declaration in nested (inner) scope */
      var2 = 5;
      var1 = 1;        /* var1 in inner scope is 1 */
}
print ("var1=%d\n", var1);  /* will print "var1=10" as var1 refers */
                            /* to var1 in the outer scope */
```

Removing the declaration of `var2` will result in a diagnostic message being generated making the programmer aware of an undeclared variable. However, removing the declaration of `var1` in the inner block will not result in a diagnostic as `var1` will be resolved to the declaration in the outer block and a programmer maintaining the code could very easily miss this subtlety. The removing of inner block `var1` will result in the printing of "`var1=1`" instead of "`var1=10`".

D.22.2 Guidance to language users

- Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and can be used in the same context. A language-specific project coding convention can be used to ensure that such errors are detectable with static analysis.
- Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and has a type that permits it to occur in at least one context where the first entity can occur.
- Ensure that all identifiers differ within the number of characters considered to be significant by the implementations that are likely to be used, and document all assumptions.

D.23 Namespace Issues [BJL]

Does not apply to C.

D.24 Initialization of Variables [LAV]

D.24.1 Applicability to language

Local, automatic variables can assume unexpected values if they are used before they are initialized. The C Standard specifies, "If an object that has automatic storage duration is not initialized explicitly, its value is

indeterminate". In the common case, on architectures that make use of a program stack, this value defaults to whichever values are currently stored in stack memory. While uninitialized memory often contains zeros, this is not guaranteed. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

Assuming that an uninitialized variable is 0 can lead to unpredictable program behaviour when the variable is initialized to a value other than 0.

Many implementations will issue a diagnostic message indicating that a variable was not initialized.

D.24.2 Guidance to language users

- Heed compiler warning messages about uninitialized variables. These warnings should be resolved as recommended to achieve a clean compile at high warning levels.
- Do not use memory allocated by functions such as `malloc()` before the memory is initialized as the memory contents are indeterminate.

D.25 Operator Precedence/Order of Evaluation [JCW]

D.25.1 Applicability to language

The order of evaluation of the operands in C is clearly defined, as is the order of evaluation.

Mixed logical operators are allowed without parentheses.

D.25.2 Guidance to language users

- Use parentheses any time mixed logical operators are used.

D.26 Side-effects and Order of Evaluation [SAM]

D.26.1 Applicability to language

C allows expressions to have side effects. If two or more side effects modify the same expression as in:

```
int v[10];
int i;
/* … */
i = v[i++];
```

the behaviour is undefined and this can lead to unexpected results. Either the "`i++`" is performed first or the assignment "`i=v[i]`" is performed first. Because the order of evaluation can have drastic effects on the functionality of the code, this can greatly impact portability.

There are several situations in C where the order of evaluation of subexpressions or the order in which side effects take place is unspecified including:

- The order in which the arguments to a function are evaluated (C99, Section 6.5.2.2,"Function calls").
- The order of evaluation of the operands in an assignment statement (C99, Section 6.5.16,"Assignment

operators").

- The order in which any side effects occur among the initialization list expressions is unspecified. In particular, the evaluation order need not be the same as the order of subobject initialization (C99, Section 6.7.8, “Initialization").

Because these are unspecified behaviours, testing may give the false impression that the code is working and portable, when it could just be that the values provided cause evaluations to be performed in a particular order that causes side effects to occur as expected.

D.26.2 Guidance to language users

- Expressions should be written so that the same effects will occur under any order of evaluation that the C standard permits since side effects can be dependent on an implementation specific order of evaluation.

D.27 Likely Incorrect Expression [KOA]

D.27.1 Applicability to language

C has several instances of operators which are similar in structure, but vastly different in meaning. This is so common that the C example of confusing the Boolean operator “==” with the assignment “=” is frequently cited as an example among programming languages. Using an expression that is technically correct, but which may just be a null statement can lead to unexpected results.

C is also provides a lot of freedom in constructing statements. This freedom, if misused, can result in unexpected results and potential vulnerabilities.

The flexibility of C can obscure the intent of a programmer. Consider:

```
int x,y;
   /* … */
if (x = y){
     /* … */
}
```

A fair amount of analysis may need to be done to determine whether the programmer intended to do an assignment as part of the `if` statement (perfectly valid in C) or whether the programmer made the common mistake of using an “=” instead of a “==”. In order to prevent this confusion, it is suggested that any assignments in contexts that are easily misunderstood be moved outside of the Boolean expression. This would change the example code to:

```
int x,y;
   /* … */
x = y;
if (x == 0) {
 /* … */
}
```

This would clearly state what the programmer meant and that the assignment of y to x was intended.

Programmers can easily get in the habit of inserting the "`;`" statement terminator at the end of statements. However, inadvertently doing this can drastically alter the meaning of code, even though the code is valid as in the following example:

```
int a,b;
/* … */
if (a == b);  // the semi-colon will make this a null statement
{
 /* … */
}
```

Because of the misplaced semi-colon, the code block following the `if` will always be executed. In this case, it is extremely likely that the programmer did not intend to put the semi-colon there.

D.27.2 Guidance to language users

- Simplify statements with interspersed comments to aid in accurately programming functionality and help future maintainers understand the intent and nuances of the code. The flexibility of C permits a programmer to create extremely complex expressions.
- Assignments embedded within other statements can be potentially problematic. Each of the following would be clearer and have less potential for problems if the embedded assignments were conducted outside of the expressions:

```
int a,b,c,d;
/* … */
if ((a == b) || (c = (d-1)))       /* the assignment to c may not
                                      occur if a is equal to b */
```

or:

```
int a,b,c;
/* … */
foo (a=b, c);
```

Each is a valid C statement, but each may have unintended results.

- Null statements should have a source line of their own. This, combined with enforcement by static analysis, would make clearer the intention that the statement was meant to be a null statement.

D.28 Dead and Deactivated Code [XYQ]

D.28.1 Applicability to language

As with any programming language that contains branching statements, C programs can potentially contain dead code. It is of concern primarily since dead code may reveal a logic flaw or an unintentional mistake on the part of the programmer. Sometimes statements can be inserted in C programs as defensive programming such as adding a default case to a switch statement even though the expectation is that the default can never be reached – until through some twist of logic or through modifications to the code the notifying error message reveals the surprising event. These types of defensive statements may be able to be shown to be computationally impossible

and thus are dead code. Those are not the focus. The focus is on those statements which are not defensive and which are unreachable. It is impossible to identify all such cases and therefore only those which are blatant and that indicate deeper issues of flawed logic may be able to be identified and removed.

C uses some operators that can be confused with other operators. For instance, the common mistake of using an assignment operator in a Boolean test as in:

```
int a,b;
/* … */
if (a = b)
…
```

can cause portions of code to become dead code since unless `b` can contain the value 0, the `else` portion of the `if` statement cannot be reached.

D.28.2 Guidance to language users

- Eliminate dead code to the extent possible from C programs.
- Use compilers and analysis tools to assist in identifying unreachable code.
- Use "`//`" comment syntax instead of "`/*…*/`" comment syntax to avoid the inadvertent commenting out sections of code.
- Delete deactivated code from programs due to the possibility of accidentally activating it.

D.29 Switch Statements and Static Analysis [CLL]

D.29.1 Applicability to language

Because of the way in which the switch-case statement in C is structured, it can be relatively easy to unintentionally omit the `break` statement between cases causing unintended execution of statements for some cases.

C contains a `switch` statement of the form:

```
char abc;
/* … */
switch (abc) {
   case 1:
        sval = "a";
   break;
   case 2:
   sval = "b";
   break;
   case 3:
   sval = "c";
   break;
   default:
   printf ("Invalid selection\n");
```

If there isn't a default case and the switched expression doesn't match any of the cases, then control simply shifts to the next statement after the switch statement block. Unintentionally omitting a `break` statement between two cases will cause subsequent cases to be executed until a `break` or the end of the switch block is reached. This could cause unexpected results.

D.29.2 Guidance to language users

- Only a direct fall through should be allowed from one case to another. That is, every nonempty `case` statement should be terminated with a `break` statement as illustrated in the following example:

```
int i;
/* … */
switch (i) {
 case 1:
 case 2:
 i++;     /* fall through from case 1 to 2 is permitted */
 break;
 case 3:
 j++;
   case 4:       /* fall through from case 3 to 4 is not permitted */
                 /* as it is not a direct fall through due to the */
                 /* j++ statement */
}
```

- All `switch` statements should have a default value if only to indicate that there could exist a case that was unanticipated and thought impossible by the developers. The only exception is for switches on an enumerated type where all possible values can be exhausted. Even in the case of enumerated types, it is suggested that a default be inserted in anticipation of possible code changes to the enumerated type.

D.30 Demarcation of Control Flow [EOJ]

D.30.1 Applicability to language

C is a block-structured language, while languages such as Ada and Pascal are comb-structured languages. Therefore, it may not be readily apparent which statements are part of a loop construct or an `if` statement.

Consider the following section of code:

```
int foo(int a, const int *b) {
   int i=0;
   /* … */
        a = 0;
   for (i=0; i<10; i++);
   {
      a = a + b[i];
   }
}
```

At first it may appear that a will be a sum of the numbers b[0] to b[9]. However, even though the code is structured so that the "a = a + b[i]" code is structured to appear within the for loop, the ";" at the end of the for statement causes the loop to be on a null statement (the ";") and the "a = a + b[i];" statement to only be executed once. In this case, this mistake may be readily apparent during development or testing. More subtle cases may not be as readily apparent leading to unexpected results.

If statements in C are also susceptible to control flow problems since there isn't a requirement in C for there to be an else statement for every if statement. An else statement in C always belong to the most recent if statement without an else. However, the situation could occur where it is not readily apparent to which if statement an else due to the way the code is indented or aligned.

D.30.2 Guidance to language users

- Enclose the bodies of if, else, while, for, etc. in braces. This will reduce confusion and potential problems when modifying the software. For example:

```
int a,b,i;

/* … */

if (i = 10){
 a = 5;    /* this is correct */
 b = 10;
}
 else
    a = 10;      /* this is incorrect -- the assignments to b */
                 /* were added later and were expected to */
    b = 5;    /* be part of the if and else and indented */
              /* as such, but did not become part of the else */
```

- Use a final else statement or a comment stating why the final else isn't necessary in all if and else if statements.

D.31 Loop Control Variables [TEX]

D.31.1 Applicability to language

C allows the modification of loop control variables within a loop. Though this is usually not considered good programming practice as it can cause unexpected problems, the flexibility of C expects the programmer to use this capability responsibly.

Since the modification of a loop control variable within a loop is infrequently encountered, reviewers of C code may not expect it and hence miss noticing the modification. Modifying the loop control variable can cause unexpected results if not carefully done. In C, the following is valid:

```
int a,i;
for (i=1; i<10; i++){
```

```
        …
        if (a > 7)
          i = 10;
      …
    }
```

which would cause the `for` loop to exit once `a` is greater than 7 regardless of the number of loops that have occurred.

D.31.2 Guidance to language users

- Do not modify a loop control variable within a loop. Even though the capability exists in C, it is still considered to be a poor programming practice.

D.32 Off-by-one Error [XZH]

D.32.1 Applicability to language

Arrays are a common place for off by one errors to manifest. In C, arrays are indexed starting at 0, causing the common mistake of looping from 0 to the size of the array as in:

```
int foo() {
  int a[10];
  int i;
  for (i=0, i<=10, i++)
        …
  return (0);
}
```

Strings in C are also another common source of errors in C due to the need to allocate space for and account for the string sentinel value. A common mistake is to expect to store an n length string in an n length array instead of length n+1 to account for the sentinel '\0'. Interfacing with other languages that do not use sentinel values in strings can also lead to an off by one error.

C does not flag accesses outside of array bounds, so an off by one error may not be as detectable in C as in some other languages. Several good and freely available tools for C can be used to help detect accesses beyond the bounds of arrays that are caused by an off by one error. However, such tools will not help in the case where only a portion of the array is used and the access is still within the bounds of the array.

Looping one more or one less is usually detectable by good testing. Due to the structure of the C language, this may be the main way to avoid this vulnerability. Unfortunately some cases may still slip through the development and test phase and manifest themselves during operational use.

D.32.2 Guidance to language users

- Use careful programming, testing of border conditions and static analysis tools to detect off by one errors in C.

D.33 Structured Programming [EWD]

D.33.1 Applicability to language

It is as easy to write structured programs in C as it is not to. C contains the `goto` statement, which can create unstructured code. Also, C has `continue`, `break`, and `return` that can create a complicated control flow, when used in an undisciplined manner. Spaghetti code can be more difficult for C static analyzers to analyze and is sometimes used on purpose to intentionally obfuscate the functionality of software. Code that has been modified multiple times by an assortment of programmers to add or remove functionality or to fix problems can be prone to become unstructured.

Because unstructured code in C can cause problems for analyzers (both automated and human) of code, problems with the code may not be detected as readily or at all as would be the case if the software was written in a structured manner.

D.33.2 Guidance to language users

- Write clear and concise structured code to make code as understandable as possible.
- Restrict the use of `goto`, `continue`, `break` and `return` to encourage more structured programming.
- Encourage the use of a single exit point from a function. At times, this guidance can have the opposite effect, such as in the case of an `if` check of parameters at the start of a function that requires the remainder of the function to be encased in the `if` statement in order to reach the single exit point. If, for example, the use of multiple exit points can arguably make a piece of code clearer, then they should be used. However, the code should be able to withstand a critique that a restructuring of the code would have made the need for multiple exit points unnecessary.

D.34 Passing Parameters and Return Values [CSJ]

D.34.1 Applicability to language

C uses *call by value* parameter passing. The parameter is evaluated and its value is assigned to the formal parameter of the function that is being called. A formal parameter behaves like a local variable and can be modified in the function without affecting the actual argument. An object can be modified in a function by passing the address to the object to the function, for example

```
void swap(int *x, int *y) {
   int t = *x;
   *x = *y;
   *y = t;
}
```

Where `x` and `y` are integer pointer formal parameters, and `*x` and `*y` in the `swap()` function body dereference the pointers to access the integers.
C macros use a *call by name* parameter passing; a call to the macro replaces the macro by the body of the macro. This is call *macro expansion*. Macro expansion is applied to the program source text and amounts to the substitution of the formal parameters with the actual parameter expressions. Formal parameters are often

parenthesized to avoid syntax issues after the expansion. Call by name parameter passing reevaluates the actual parameter expression each time the formal parameter is read.

D.34.2 Guidance to language users

- Use caution for reevaluation of function calls in parameters with macros.
- Use caution when passing the address of an object. The object passed could be an alias[12].

D.35 Dangling References to Stack Frames [DCM]

D.35.1 Applicability to language

C allows the address of a variable to be stored in a variable. Should this variable's address be, for example, the address of a local variable that was part of a stack frame, then using the address after the local variable has been deallocated can yield unexpected behaviour as the memory will have been made available for further allocation and may indeed been allocated for some other use. Any use of perishable memory after it has been deallocated can lead to unexpected results.

D.35.2 Guidance to language users

- Do not assign the address of an object to any entity which persists after the object has ceased to exist. This is done in order to avoid the possibility of a dangling reference. Once the object ceases to exist, then so will the stored address of the object preventing accidental dangling references.
- Long lived pointers that contain block-local addresses should be assigned the null pointer value before executing a return from the block.

D.36 Subprogram Signature Mismatch [OTR]

D.36.1 Applicability to language

Functions in C may be called with more or less than the number of parameters the receiving function expects. However, most C compilers will generate a warning or an error about this situation. If the number of arguments does not equal the number of parameters, the behaviour is undefined. This can lead to unexpected results when the count or types of the parameters differs from the calling to the receiving function. If too few arguments are sent to a function, then the function could still pop the expected number of arguments from the stack leading to unexpected results.

C allows a variable number of arguments in function calls. A good example of an implementation of this is the `printf()` function. This is specified in the function call by terminating the list of parameters with an ellipsis (`, ...`). After the comma, no information about the number or types of the parameters is supplied. This can be a useful feature for situations such as `printf()`, but the use of this feature outside of special situations can be the basis for vulnerabilities.

[12] An alias is a variable or formal parameter that refers to the same location as another variable or formal parameter.

Functions may or may not be defined with a function definition. The function definition may or may not contain a parameter type list. If a function that accepts a variable number of arguments is defined without a parameter

type list that ends with the ellipsis notation, the behaviour is undefined.

If the calling and receiving functions differ in the type of parameters, C will, if possible, do an implicit conversion such as the call to `sqrt()` that expects a double:

```
double sqrt(double)
```

the call:

```
root2 = sqrt(2);
```

coerces the integer 2 into the double value 2.0.

D.36.2 Guidance to language users

- Use a function prototype to declare a function with its expected parameters to allow the compiler to check for a matching count and types of the parameters. The prototype contains just the name of the function and its parameters without the body of code that would normally follow.
- Do not use the variable argument feature except in rare instances. The variable argument feature such as is used in `printf()` is difficult to use in a type safe manner.

D.37 Recursion [GDL]

D.37.1 Applicability to language

C permits recursive calls both directly and indirectly through any chain of other functions. However, recursive functions must be implemented carefully in C,C does not have protective mechanisms that could avert serious problems such as an overly large consumption of resources or an overrun of buffers. Since C is frequently cited for its high performance efficiency, the use of recursion in C can be counter to this as recursion can be inefficient both in execution time and memory usage. Some of the modern compilers perform tail-call optimization to make recursion efficient and resource-friendly.

As with many languages, the high consumption of resources for recursive calls can apply to C. It is difficult to predict the complete range of values that a recursive function can execute that will lead to a manageable consumption of resources. Part of this difficulty is that the range of values can change depending on the current load of the host. Manipulation of the input values to a recursive function can result in an intentional exhaustion of system resources leading to a denial of service.

D.37.2 Guidance to language users

- Only use recursion in rare instances. Although recursion can shorten programs considerably, there is a high performance penalty which is contrary to the usual high efficiency of C.
- Only use recursion if it can be proven that adequate resources exist to support the maximum level of recursion possible.

D.38 Ignored Error Status and Unhandled Exceptions [OYB]

D.38.1 Applicability to language

The C standard does not include exception handling, therefore only error status will be covered.

C provides the include file <errno.h> that defines the macros EDOM, EILSEQ and ERANGE, which expand to integer constant expressions with type int, distinct positive values and which are suitable for use in #if preprocessing directives. C also provides the integer errno that can be set to a nonzero value by any library function (if the use of errno is not documented in the description of the function in the C Standard, errno could be used whether or not there is an error). Though these values are defined, inconsistencies in responding to error conditions can lead to vulnerabilities.

D.38.2 Guidance to language users

- Check the returned error status upon return from a function. The C standard library functions provide an error status as the return value and sometimes in an additional global error value.
- Set errno to zero before a library function call in situations where a program intends to check errno before a subsequent library function call.
- Use errno_t to make it readily apparent that a function is returning an error code. Often a function that returns an errno error code is declared as returning a value of type int. Although syntactically correct, it is not apparent that the return code is an errno error code. TR 24731-1 introduced the new type errno_t in <errno.h> that is defined to be type int.

D.39 Termination Strategy [REU]

D.39.1 Applicability to language

Choosing when and where to exit is a design issue, but choosing how to perform the exit may result in the host being left in an unexpected state. C provides several ways of terminating a program including exit(), _Exit(), and abort(). A return from the initial call to the main function is equivalent to calling the exit() function with the value returned by the main function as its argument (this is if the return type of the main function is a type compatible with int, otherwise the termination status returned to the host environment is unspecified) or simply reaching the "}" that terminates the main function returns a value of 0.

All of the termination strategies in C have undefined, unspecified, and/or implementation defined behaviour associated with them. For example, if more than one call to the exit() function is executed by a program, the behaviour is undefined. The amount of clean-up that occurs upon termination such as the removal of temporary files or the flushing of buffers varies and may be implementation defined.

A call to exit() or _Exit() will terminate a program normally. Abnormal program termination will occur when abort() is used to exit a program (unless the signal SIGABRT is caught and the signal handler does not return). Unlike a call to exit(), when either _Exit() or abort() are used to terminate a program, it is implementation defined as to whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed. This can leave a system in an unexpected state.

C provides the function `atexit()` that allows functions to be registered so that at normal program termination, the registered functions will be executed to perform desired functions. C99 requires the capability to register *at least* 32 functions. Implementations expecting more than 32 registered functions may yield unexpected results.

D.39.2 Guidance to language users

- Use a return from the `main()` program as it is the cleanest way to exit a C program.
- Use `exit()` to quickly exit from a deeply nested function.
- Use `abort()` in situations where an abrupt halt is needed. If `abort()` is necessary, the design should protect critical data from being exposed after an abrupt halt of the program.
- Become familiar with the undefined, unspecified and/or implementation aspects of each of the termination strategies.

D.40 Type-breaking Reinterpretation of Data [AMV]

D.40.1 Applicability to language

The primary way in C that a reinterpretation of data is accomplished is through a `union` which may be used to interpret the same piece of memory in multiple ways. If the use of the union members is not managed carefully, then unexpected and erroneous results may occur.

C allows the use of pointers to memory so that an integer pointer could be used to manipulate character data. This could lead to a mistake in the logic that is used to interpret the data leading to unexpected and erroneous results.

D.40.2 Guidance to language users

- Avoid the use of unions as it is relatively easy for there to exist an unexpected program flow that leads to a misinterpretation of the union data.

D.41 Memory Leak [XYL]

D.41.1 Applicability to language

C can allow memory leaks as many programs use dynamically allocated memory. C relies on manual memory management rather than a built in garbage collector primarily since automated memory management can be unpredictable, impact performance and is limited in its ability to detect unused memory such as memory that is still referenced by a pointer, but is never used.

Memory is dynamically allocated in C using the library calls `malloc()`, `calloc()`, and `realloc()`. When the program no longer needs the dynamically allocated memory, it can be released using the library call `free()`. Should there be a flaw in the logic of the program, memory continues to be allocated but is not freed when it is no longer needed. A common situation is where memory is allocated while in a function, the memory is not freed before the exit from the function and the lifetime of the pointer to the memory has ended upon exit from the function.

D.41.2 Guidance to language users

- Use debugging tools such as leak detectors to help identify unreachable memory.
- Allocate and free memory in the same module and at the same level of abstraction to make it easier to determine when and if an allocated block of memory has been freed.
- Use `realloc()` only to resize dynamically allocated arrays.
- Use garbage collectors that are available to replace the usual C library calls for dynamic memory allocation which allocate memory to allow memory to be recycled when it is no longer reachable. The use of garbage collectors may not be acceptable for some applications as the delay introduced when the allocator reclaims memory may be noticeable or even objectionable leading to performance degradation.

D.42 Templates and Generics [SYM]

Does not apply to C.

D.43 Inheritance [RIP]

Does not apply to C.

D.44 Extra Intrinsics [LRM]

Does not apply to C.

D.45 Argument Passing to Library Functions [TRJ]

D.45.1 Applicability to language

Parameter passing in C is either pass by reference or pass by value. There isn't a guarantee that the values being passed will be verified by either the calling or receiving functions. So values outside of the assumed range may be received by a function resulting in a potential vulnerability.

A parameter may be received by a function that was assumed to be within a particular range and then an operation or series of operations is performed using the value of the parameter resulting in unanticipated results and even a potential vulnerability.

D.45. Guidance to language users

- Do not make assumptions about the values of parameters.
- Do not assume that the calling or receiving function will be range checking a parameter. It is always safest to not make any assumptions about parameters used in C libraries. Because performance is sometimes cited as a reason to use C, parameter checking in both the calling and receiving functions is considered a waste of time. Since the calling routine may have better knowledge of the values a parameter can hold, it may be considered the better place for checks to be made as there are times when a parameter doesn't need to be checked since other factors may limit its possible values. However, since the receiving routine understands how the parameter will be used and it is good practice to check all inputs, it makes sense for the receiving routine to check the value of parameters. Therefore, in C it is

difficult to create a blanket statement as to where the parameter checks should be made and as a result, parameter checks are recommended in both the calling and receiving routines unless knowledge about the calling or receiving routines dictates that this isn't needed.

D.46 Inter-language Calling [DJS]

The C Standard defines the calling conventions, data layout, error handing and return conventions needed to use C from another language. Ada and Fortran have developed a guideline to call C using the Standard.

D.47 Dynamically-linked Code and Self-modifying Code [NYY]

D.47.1 Applicability to language

Most loaders allow dynamically linked libraries also known as shared libraries. Code is designed and tested using a suite of shared libraries which are loaded at execution time. The process of linking and loading is outside the scope of the C standard.

C can allow self-modifying code. In C there isn't a distinction between data space and code space, executable commands can be altered as desired during the execution of the program. Although self-modifying code may be easy to do in C, it can be difficult to understand, test and fix leading to potential vulnerabilities in the code.

Self-modifying code can be done intentionally in C to obfuscate the effect of a program or in some special situations to increase performance. Because of the ease with which executable code can be modified in C, accidental (or maliciously intentional) modification of C code can occur if pointers are misdirected to modify code space instead of data space or code is executed in data space. Accidental modification usually leads to a program crash. Intentional modification can also lead to a program crash, but used in conjunction with other vulnerabilities can lead to more serious problems that affect the entire host.

D.47.2 Guidance to language users

- Use signatures to verify that the shared libraries used are identical to the libraries with which the code was tested.
- Do not use self-modifying code except in rare instances. In those rare instances, self-modifying code in C can and should be constrained to a particular section of the code and well commented.

D.48 Library Signature [NSQ]

D.48.1 Applicability to language

Integrating C and another language into a single executable relies on knowledge of how to interface the function calls, argument lists and data structures so that symbols match in the object code during linking. Byte alignments can be a source of data corruption.

For instance, when calling Fortran from C, several issues arise. Neither C nor Fortran check for mismatch argument types or even the number of arguments. C passes arguments by value and Fortran passes arguments by reference, so addresses must be passed to Fortran rather than values in the argument list. Multidimensional arrays in C are stored in row major order, whereas Fortran stores them in column major order. Strings in C are

terminated by a null character, whereas Fortran uses the declared length of a string. These are just some of the issues that arise when calling Fortran programs from C. Each language has its differences with C, so different issues arise with each interface.

Writing a library wrapper is the traditional way of interfacing with code from another language. However, this can be quite tedious and error prone.

D.48.2 Guidance to language users

- Use a tool, if possible, to automatically create the interface wrappers.
- Minimize the use of those issues known to be error prone when interfacing from C, such as passing character strings, passing multi-dimensional arrays to a column major language, interfacing with other parameter formats such as call by reference or name and receiving return codes.

D.49 Unanticipated Exceptions from Library Routines [HJW]

D.49.1 Applicability to language

Calling software routines produced outside of the control of the main application developer puts all of the code at the mercy of the called routines. An unanticipated exception generated from a library routine could have devastating consequences.

D.49.2 Guidance to language users

- Check the values of parameters to ensure appropriate values are passed to libraries in order to reduce or eliminate the chance of an unanticipated exception

D.50 Pre-processor Directives [NMP]

D.50.1 Applicability to language

The C pre-processor allows the use of macros that are text-replaced before compilation.

Function-like macros look similar to functions but have different semantics. Because the arguments are text-replaced, expressions passed to a function-like macro may be evaluated multiple times. This can result in unintended and undefined behaviour if the arguments have side effects or are pre-processor directives as described by C99 §6.10 [1]. Additionally, the arguments and body of function-like macros should be fully parenthesized to avoid unintended and undefined behaviour [2].

The following code example demonstrates undefined behaviour when a function-like macro is called with arguments that have side-effects (in this case, the increment operator) [2]:

```
#define CUBE(X) ((X) * (X) * (X))
/* ... */
int i = 2;
int a = 81 / CUBE(++i);
```

The above example could expand to:

```
int a = 81 / ((++i) * (++i) * (++i));
```

this is undefined behaviour so this macro expansion is difficult to predict.

Another mechanism of failure can occur when the arguments within the body of a function-like macro are not fully parenthesized. The following example shows the `CUBE` macro without parenthesized arguments [2]:

```
#define CUBE(X) (X * X * X)
/* ... */
int a = CUBE(2 + 1);
```

This example expands to:

```
int a = (2 + 1 * 2 + 1 * 2 + 1)
```

which evaluates to 7 instead of the intended 27.

D.50.2 Guidance to language users

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

- Replace macro-like functions with inline functions where possible. Although making a function inline only suggests to the compiler that the calls to the function be as fast as possible, the extent to which this is done is implementation-defined. Inline functions do offer consistent semantics and allow for better analysis by static analysis tools.
- Ensure that if a function-like macro must be used, that its arguments and body are parenthesized.
- Do not embed pre-processor directives or side-effects such as an assignment, increment/decrement, volatile access, or function call in a function-like macro.

D.51 Suppression of Language-defined Run-time Checking [MXB]

Does not apply to C.

D.52 Provision of Inherently Unsafe Operations [SKL]

Does not apply to C.

D.53 Obscure Language Features [BRS]

D.53.1 Applicability to language

C is a relatively small language with a limited syntax set lacking many of the complex features of some other languages. Many of the complex features in C are not implemented as part of the language syntax, but rather implemented as library routines. As such, most of the available features in C are used relatively frequently.

Common use across a variety of languages may make some features less obscure. Because of the unstructured code that is frequently the result of using `goto`'s, the `goto` statement is frequently restricted, or even outright banned, in some C development environments. Even though the `goto` is encountered infrequently and the use

of it considered obscure, because it is fairly obvious as to its purpose and since its use is common to many other languages, the functionality of it is easily understood by even the most junior of programmers.

The use of a combination of features adds yet another dimension. Particular combinations of features in C may be used rarely together or fraught with issues if not used correctly in combination. This can cause unexpected results and potential vulnerabilities.

D.53.2 Guidance to language users

- Organizations should specify coding standards that restrict or ban the use of features or combinations of features that have been observed to lead to vulnerabilities in the operational environment for which the software is intended.

D.54 Unspecified Behaviour [BQF]

D.54.1 Applicability to language

The C standard has documented, in Annex J.1, 54 instances of unspecified behaviour. Examples of unspecified behaviour are:

- The order in which the operands of an assignment operator are evaluated
- The order in which any side effects occur among the initialization list expressions in an initializer
- The layout of storage for function parameters

Reliance on a particular behaviour that is unspecified leads to portability problems because the expected behaviour may be different for any given instance. Many cases of unspecified behaviour have to do with the order of evaluation of subexpressions and side effects. For example, in the function call

```
f1(f2(x), f3(x));
```

the functions `f2` and `f3` may be called in any order possibly yielding different results depending on the order in which the functions are called.

D.54.2 Guidance to language users

- Do not rely on unspecified behaviour because the behaviour can change at each instance. Thus, any code that makes assumptions about the behaviour of something that is unspecified should be replaced to make it less reliant on a particular installation and more portable.

D.55 Undefined Behaviour [EWF]

D.55.1 Applicability to language

The C standard does not impose any requirements on undefined behaviour. Typical undefined behaviours include doing nothing, producing unexpected results, and terminating the program.

The C standard has documented, in Annex J.2, 191 instances of undefined behaviour that exist in C. One example of undefined behaviour occurs when the value of the second operand of the `/` or `%` operator is zero. This is

generally not detectable through static analysis of the code, but could easily be prevented by a check for a zero divisor before the operation is performed. Leaving this behaviour as undefined lessens the burden on the implementation of the division and modulo operators.

Other examples of undefined behaviour are:

- Referring to an object outside of its lifetime
- The conversion to or from an integer type that produces a value outside of the range that can be represented
- The use of two identifiers that differ only in non-significant characters

Relying on undefined behaviour makes a program unstable and non-portable. While some cases of undefined behaviour may be consistent across multiple implementations, it is still dangerous to rely on them. Relying on undefined behaviour can result in errors that are difficult to locate and only present themselves under special circumstances. For example, accessing memory deallocated by `free()` or `realloc()` results in undefined behaviour, but it may work most of the time.

D.55.2 Guidance to language users

- Eliminate to the extent possible all cases of undefined behaviour from a program

D.56 Implementation-defined Behaviour [FAB]

D.56.1 Applicability to language

The C standard has documented, in Annex J.3, 112 instances of implementation-defined behaviour. Examples of implementation-defined behaviour are:

- The number of bits in a byte
- The direction of rounding when a floating-point number is converted to a narrower floating-point number
- The rules for composing valid file names

Relying on implementation-defined behaviour can make a program less portable across implementations. However, this is less true than for unspecified and undefined behaviour.

The following code shows an example of reliance upon implementation-defined behaviour:

```
unsigned int x = 50;
x += (x << 2) + 1;  // x = 5x + 1
```

Since the bitwise representation of integers is implementation-defined, the computation on `x` will be incorrect for implementations where integers are not represented in two's complement form.

D.56.2 Guidance to language users

- Eliminate to the extent possible any reliance on implementation-defined behaviour from programs in order to increase portability. Even programs that are specifically intended for a particular

implementation may in the future be ported to another environment or sections reused for future implementations.

D.57 Deprecated Language Features [MEM]

D.57.1 Applicability to language

C has deprecated one function, the function `gets()`. The `gets()` function copies a string from standard input into a fixed-size array. There is no safe way to use `gets()` because it performs an unbounded copy of user input. Thus, every use of gets constitutes a buffer overflow vulnerability.

C has deprecated several language features primarily by tightening the requirements for the feature:

- Implicit `int` declarations are no longer allowed.
- Functions cannot be implicitly declared. They must be defined before use or have a prototype.
- The use of the function `ungetc()` at the beginning of a binary file is deprecated.
- The deprecation of aliased array parameters has been removed.
- A `return` without expression is not permitted in a function that returns a value (and vice versa).

Violating any of these features will generate a diagnostic message.

D.57.2 Guidance to language users

- Although backward compatibility is sometimes offered as an option for compilers so one can avoid changes to code to be compliant with current language specifications, updating the legacy software to the current standard is a better option.

D.58 Implications for standardization

Future standardization efforts should consider:

- Moving in the direction over time to being a more strongly typed language. Much of the use of weak typing is simply convenience to the developer in not having to fully consider the types and uses of variables. Stronger typing forces good programming discipline and clarity about variables while at the same time removing many unexpected run time errors due to implicit conversions. This is not to say that C should be strictly a strongly typed language – some advantages of C are due to the flexibility that weaker typing provides. It is suggested that when enforcement of strong typing does not detract from the good flexibility that C offers (for example, adding an integer to a character to step through a sequence of characters) and is only a convenience for programmers (for example, adding an integer to a floating-point number), then the standard should specify the stronger typed solution.
- A common warning in Annex I should be added for floating-point expressions being used in a Boolean test for equality.
- Modifying or deprecating many of the C standard library functions that make assumptions about the occurrence of a string termination character.
- Define a string construct that does not rely on the null termination character.
- Defining an array type that does automatic bounds checking.

- Deprecating less safe functions such as `strcpy()` and `strcat()` where a more secure alternative is available.
- Defining safer and more secure replacement functions such as `memncpy()` and `memncmp()` to complement the `memcpy()` and `memcmp()` functions (see in Implications for standardization.XYW).
- Defining an array type that does automatic bounds checking.
- Defining functions that contain an extra parameter in `memcpy()` and `memmove()` for the maximum number of bytes to copy. In the past, some have suggested that the size of the destination buffer be used as an additional parameter. Some critics state that this solution is easy to circumvent by simply repeating the parameter that was used for the number of bytes to copy as the parameter for the size of the destination buffer. This analysis and criticism is correct. What is needed is a failsafe check as to the maximum number of bytes to copy. There are several reasons for creating new functions with an additional parameter. This would make it easier for static analysis to eliminate those cases where the memory copy could not be a problem (such as when the maximum number of bytes is demonstrably less than the capacity of the receiving buffer). Manual analysis or more involved static analysis could then be used for the remaining situations where the size of the destination buffer may not be sufficient for the maximum number of bytes to copy. This extra parameter may also help in determining which copies could take place among objects that overlap. Such copying is undefined according to the C standard. It is suggested that safer versions of functions that include a restriction `max_n` on the number of bytes `n` to copy (for example, `void *memncpy(void * restrict s1,const void * restrict s2,size_t n), const size_t max_n`) be added to the standard in addition to retaining the current corresponding functions (for example, `memcpy(void * restrict s1,const void * restrict s2,size_t n))`). The additional parameter would be consistent with the copying function pairs that have already been created such as `strcpy()`/`strncpy()` and `strcat()`/`strncat()`. This would allow a safer version of memory copying functions for those applications that want to use them in to facilitate both safer and more secure code and more efficient and accurate static code reviews.[14]
- Restrictions on pointer arithmetic that could eliminate common pitfalls. Pointer arithmetic is error prone and the flexibility that it offers is useful, but some of the flexibility is simply a shortcut that if restricted could lessen the chance of a pointer arithmetic based error.
- Modifying the library `free(void *ptr)` so that it sets `ptr` to `NULL` to prevent reuse of `ptr`.
- Defining a standard way of declaring an attribute to indicate that a variable is intentionally unused.
- A common warning in Annex I should be added for variables with the same name in nested scopes.
- Creating a few standardized precedence orders. Standardizing on a few precedence orders will help to eliminate the confusing intricacies that exist between languages. This would not affect current languages as altering precedence orders in existing languages is too onerous. However, this would set a basis for the future as new languages are created and adopted. Stating that a language uses “ISO precedence order A” would be useful rather than having to spell out the entire precedence order that differs in a conceptually minor way from some other languages, but in a major way when programmers attempt to switch between languages.
- Deprecating the `goto` statement. The use of the `goto` construct is often spotlighted as the antithesis of

[14] This has been addressed by WG 14 in an optionally normative annex in the current working paper

good structured programming. Though its deprecation will not instantly make all C code structured, deprecating the `goto` and leaving in place the restricted `goto` variations (for example, `break` and `continue`) and possibly adding other restricted `goto`'s could assist in encouraging safer and more secure C programming in general.

- Defining a "fallthru" construct that will explicitly bind multiple switch cases together and eliminate the need for the `break` statement. The default would be for a case to break instead of falling through to the next case. Granted this is a major shift in concept, but if it could be accomplished, less unintentional errors would occur.
- Defining an identifier type for loop control that cannot be modified by anything other than the loop control construct would be a relatively minor addition to C that could make C code safer and encourage better structured programming.
- Defining a standardized interface package for interfacing C with many of the top programming languages and a reciprocal package should be developed of the other top languages to interface with C.
- Joining with other languages in developing a standardized set of mechanisms for detecting and treating error conditions so that all languages to the extent possible could use them. Note that this does not mean that all languages should use the same mechanisms as there should be a variety (label parameters, auxiliary status variables), but each of the mechanisms should be standardized.
- Since fault handling and exiting of a program is common to all languages, it is suggested that common terminology such as the meaning of fail safe, fail hard, fail soft, etc. along with a core API set such as `exit`, `abort`, etc. be standardized and coordinated with other languages.
- Deprecating unions. The primary reason for the use of unions to save memory has been diminished considerably as memory has become cheaper and more available. Unions are not statically type safe and are historically known to be a common source of errors, leading to many C programming guidelines specifically prohibiting the use of unions.
- Creating a recognizable naming standard for routines such that one version of a library does parameter checking to the extent possible and another version does no parameter checking. The first version would be considered safer and more secure and the second could be used in certain situations where performance is critical and the checking is assumed to be done in the calling routine. A naming standard could be made such that the library that does parameter checking could be named as usual, say "library_xyz" and an equivalent version that does not do checking could have a "_p" appended, such as "library_xyz_p". Without a naming standard such as this, a considerable number of wasted cycles will be conducted doing a double check of parameters or even worse, no checking will be done in both the calling and receiving routines as each is assuming the other is doing the checking.
- Creating an Annex that lists deprecated features.

Annex E
(*informative*)
Vulnerability descriptions for the language Python

E.1 Identification of standards and associated documents

Enums for Python (Python recipe). (n.d.). Retrieved from ActiveState: http://code.activestate.com/recipes/67107/

Isaac, A. G. (2010, 06 23). *Python Introduction*. Retrieved 05 12, 2011, from https://subversion.american.edu/aisaac/notes/python4class.xhtml#introduction-to-the-interpreter

Lutz, M. (2009). *Learning Python.* Sebastopol, CA: O'Reilly Media, Inc.

Lutz, M. (2011). *Programming Python.* Sebastopol, CA: O'Reilly Media, Inc.

Martelli, A. (2006). *Python in a Nutshell.* Sebastopol, CA: O'Reilly Media, Inc.

Norwak, H. (n.d.). *10 Python Pitfalls*. Retrieved 05 13, 2011, from 10 Python Pitfalls: http://zephyrfalcon.org/labs/python_pitfalls.html

Pilgrim, M. (2004). *Dive Into Python.*

Python Gotchas. (n.d.). Retrieved from http://www.ferg.org/projects/python_gotchas.html

source, G. (n.d.). *Big List of Portabilty in Python*. Retrieved 6 12, 2011, from stackoverflow: http://stackoverflow.com/questions/1883118/big-list-of-portability-in-python

The Python Language Reference. (n.d.). Retrieved from python.org: http://docs.python.org/reference/index.html#reference-index

E.2 General Terminology and Concepts

E.2.1 General Terminology

assignment statement: Used to create (or rebind) a variable to an object. The simple syntax is `a=b`, the augmented syntax applies an operator at assignment time (e.g., `a += 1`) and therefore cannot create a variable since it operates using the current value referenced by a variable. Other syntaxes support multiple targets (e.g., `x = y = z = 1`).

body: The portion of a compound statement that follows the header. It may contain other compound (nested) statements.

boolean: A truth value where `True` equivalences to any non-zero value and `False` equivalences to zero. Commonly expressed numerically as 1 (true), or 0 (false) but referenced as `True` and `False`.

built-in: A function provided by the Python language intrinsically without the need to import it (e.g., `str, slice, type`).
class: A program defined type which is used to instantiate objects and provide attributes that are common to all the objects that it instantiates.

comment: Comments are preceded by a hash symbol "#".

complex number: A number made up of two parts each expressed as floating-point numbers: a real and an imaginary part. The imaginary part is expressed with a trailing upper or lower case "`J` or `j`".

compound statement: A structure that contains and controls one or more statements.

CPython: The standard implementation of Python coded in ANSI portable C.

dictionary: A built-in mapping consisting of zero or more key/value "pairs". Values are stored and retrieved using keys which can be of mixed types (with some caveats beyond the scope of this annex).

docstring: One or more lines in a unit of code that serve to document the code. Docstrings are retrievable at run-time.

exception: An object that encapsulates the attributes of an exception (an error or abnormal event). Raising an exception is a process that creates the exception object and propagates it through a process that is optionally defined in a program. Lacking an exception 'handler", Python terminates the program with an error message.

floating-point number: A real number expressed with a decimal point, an exponent expressed as an upper or lower case "e or E" or both (e.g., `1.0, 27e0, .456`).

function: A grouping of statements, either built-in or defined in a program using the `def` statement, which can be called as a unit.

garbage collection: The process by which the memory used by unreferenced object and their namespaces is reclaimed. Python provides a `gc` module to allow a program to direct when and how garbage collection is done.

global: A variable that is scoped to a module and can be referenced from anywhere within the module including within functions and classes defined in that module.

guerrilla patching: Also known as Monkey Patching, the practice of changing the attributes and/or methods of a module's class at run-time from outside of the module.

immutability: The characteristic of being unchangeable. Strings, tuples, and numbers are immutable objects in Python.

import: A mechanism that is used to make the contents of a module accessible to the importing program.

inheritance: The ability to define a class that is a subclass of other classes (i.e., superclass). Inheritance uses a method resolution order (MRO) to resolve references to the correct inheritance level (i.e., it resolves attributes (methods and variables)).

instance: A single occurrence of a class that is created by calling the class as if it was a function (e.g., `a = Animal()`).

integer: An integer can be of any length but is more efficiently processed if it can be internally represented by a 32 or 64 bit integer. Integer literals can be expressed in binary, decimal, octal, or hexadecimal formats.

keyword: An identifier that is reserved for special meaning to the Python interpreter (e.g., `if`, `else`, `for`, `class`).

lambda expression: A convenient way to express a single return function statement within another statement instead of defining a separate function and referencing it.

list: An ordered sequence of zero or more items which can be modified (i.e., is mutable) and indexed.

literals: A string or number (e.g., `'abc', 123, 5.4`). Note that a string literal can use either double quote (“) or single apostrophe pairs (‘) to delimit a string.

membership: If an item occurs within a sequence it is said to be a member. Python has built-ins to test for membership (e.g., `if a in b`). Classes can provide methods to override built-in membership tests.

module: A file containing source language (i.e., statements) in Python (or another) language. A module has its own namespace and scope and may contain definitions for functions and classes. A module is only executed when first imported and upon reloading.

mutability: The characteristic of being changeable. Lists and dictionaries are two examples of Python objects that are mutable.

name: A variable that references a Python object such as a number, string, list, dictionary, tuple, set, builtin, module, function, or class.

namespace: A place where names reside with their references to the objects that they represent. Examples of objects that have their own namespaces include: blocks, modules, classes, and functions. Namespaces provide a way to enforce scope and thus prevent name collisions since each unique name exists in only one namespace.

none: A null object.

number: An integer, floating point, decimal, or complex number.

operator: Non-alphabetic characters, characters, and character strings that have special meanings within expressions (e.g., `+, -, not, is`).

overriding: Coding an attribute in a subclass to replace a superclass attribute.

package: A collection of one or more other modules in the form of a directory.

pickling: The process of serializing objects using the `pickle` module.

polymorphism: The meaning of an operation – generally a function/method call – depends on the objects being operated upon, not the *type* of object. One of Python’s key principles is that object interfaces support operations

regardless of the type of object being passed. For example, string methods support addition and multiplication just as methods on integers and other numeric objects do.

recursion: The ability of a function to call itself. Python supports recursion to a level of 1,000 unless that limit is modified using the `setrecursionlimit` function.

scope: The visibility of a name is its scope. All names within Python exist within a specific namespace which is tied to a single block, function, class, or module in which the name was last assigned a value.

script: A unit of code generally synonymous with a *program* but usually connotes code run at the highest level as in "*scripts run modules*".

self: By convention, the name given to a class' instance variable.

sequence: An ordered container of items that can be indexed or sliced using positive numbers. Python provides three built-in sequences: strings, tuples, and lists. New sequences can also be defined in libraries, extension modules, or within classes.

set: An unordered sequence of zero or more items which do not need to be of the same type. Sets can be frozen (immutable) or unfrozen (mutable).

short-circuiting operators: Operators `and` and `or` can short-circuit the evaluation of their operand if the left side evaluates to true (in the case of the `or`) or false (in the case of `and`). For example, in the expression `a or b`, there is no need to evaluate `b` if `a` is `True`, likewise in the expression `a and b`, there is no need to evaluate `b` if `a` is `False`.

statement: An expression that generally occupies one line. Multiple statements can occupy the same line if separated by a semicolon (`;`) but this is very unconventional in Python where each line typically contains one statement.

string: A built-in sequence object consisting of one or more characters. Unlike many other languages, Python strings cannot be modified (i.e., they are "immutable") and they do not have a termination character.

tuple: A sequence of zero or more items (e.g., `(1,2,3)` or `("A", "B", "C")`). Tuples are immutable and may contain different object types (e.g., `(1, "a", 5.678)`).

variable: Python variables (i.e., names) are not like variables in most other languages - they are never declared they are dynamically referenced to objects, they have no type, and they may be bound to objects of different types at different times. Variables are bound explicitly (e.g., `a = 1` binds `a` to the integer `1`) and unbound implicitly (e.g., `a=1; a=2`). In the last example, `a` is bound to the object (value) `1` then implicitly unbound to that object when bound to `2` - a process known as rebinding. Variables can also be unbound explicitly using the `del` statement (e.g., `del a, b, c`).

E.2.2 Key Concepts

The key concepts discussed in this section are not entirely unique to Python but they are implemented in Python in ways that are not intuitive to new and experienced programmers alike.

Dynamic Typing – A frequent source of confusion is Python's dynamic typing and its effect on variable assignments (*name* is synonymous with *variable* in this annex). In Python there are no static declarations of variables - they are created, rebound, and deleted dynamically. Further, variables are not the objects that they point to - they are just references to objects which can be, and frequently are, bound to other objects at any time:

```
a = 1 # a is bound to an integer object whose value is 1
a = 'abc' # a is now bound to a string object
```

Variables have no type – they reference objects which have types thus the statement `a = 1` creates a new variable called `a` that references a new object whose value is `1` and type is integer. That variable can be deleted with a `del` statement or bound to another object any time as shown above. Refer to E.3 Type System [IHN] for more on this subject. For the purpose of brevity this annex often treats the term variable (or name) as being the object which is technically incorrect but simpler. E.g., in the statement `a = 1`, the numeric object `a` is assigned the value `1`. In reality the name `a` is assigned to a newly created *object* of type integer which is assigned the value `1`.

Section E.43 Extra Intrinsics [LRM] covers dynamic typing in more detail.

Mutable and Immutable Objects - Note that in the statement: `a = a + 1`, Python creates a *new* object whose value is calculated by adding `1` to the value of the current object referenced by `a`. If, prior to the execution of this statement `a`'s object had contained a value of `1`, then a new integer object with a value of `2` would be created. The integer object whose value was `1` is now marked for deletion using garbage collection (provided no other variables reference it). Note that the value of `a` is not updated in place, i.e., the object references by `a` does not simply have `1` added to it as would be typical in other languages. The reason this does not happen in Python is because integer objects, as well as string, number and tuples, are immutable – they cannot be changed in place. Only lists and dictionaries can be changed in place – they are mutable. In practice this restriction of not being able to change a mutable object in place is mostly transparent but a notable exception is when immutable objects are passed as a parameter to a function or class. See Section E.23 Initialization of Variables [LAV] for a description of this.

The underling actions that are performed to enable the *apparent* in-place change do not update the immutable object – they create a new object and "point" the variable to new object. This can be proven as below (the `id` function returns an object's address):

```
a = 'abc'
print(id(a))#=> 30753768
a = 'abc' + 'def'
print(id(a))#=> 52499320
print(a)#=> abcdef
```

The updating of objects referenced in the parameters passed to a function or class is governed by whether the object is mutable, in which case it is updated in place, or immutable in which case a local copy of the object is created and updated which has no effect on the passed object. This is described in more detail in Section E.33 Passing Parameters and Return Values [CSJ].

E.3 Type System [IHN]

E.3.1 Applicability to language

Python abstracts all data as objects and every object has a type (in addition to an identity and a value). Extensions to Python, written in other languages, can define new types.

Python is also a strongly typed language – you cannot perform operations on an object that are not valid for that type. Python's dynamic typing is a key feature designed to promote polymorphism to provide flexibility. Another aspect of dynamic typing is a variable does not maintain any type information – that information is held by the object that the variable references at a specific time. A Python program is free to assign (bind), and reassign (rebind), any variable to any type of object at any time.

Variables are created when they are first assigned a value (see E.19 for more on this subject). Variables are generic in that they do not have a type, they simply reference objects which hold the object's type information. Variables in an expression are replaced with the object they reference when that expression is evaluated therefore a variable must be explicitly assigned before being referenced otherwise a run-time exception is raised:

```
a = 1
if a == 1 : print(b) # error – b is not defined
```

When line 1 above is interpreted an object of type `integer` is created to hold the value `1` and the variable `a` is created and linked to that object. The second line illustrates how an error is raised if a variable (`b` in this case) is referenced before being assigned to an object.

```
a = 1
b = a
a = 'x'
print(a,b)#=> x 1
```

Variables can share references as above – `b` is assigned to the same object as `a`. This is known as a shared reference. If `a` is later reassigned to another object (as in line 3 above), `b` will still be assigned to the initial object that `a` was assigned to when `b` shared the reference, in this case `b` would equal to 1.

The subject of shared references requires particular care since its effect varies according to the rules for in-place object changes. In-places object changes are allowed only for mutable (i.e., alterable) objects. Numeric objects and strings are immutable (unalterable). Lists and dictionaries are mutable which affects how shared references operate as below:

```
a = [1,2,3]
b = a
a[0] = 7
print(a) # [7, 2, 3]
print(b) # [7, 2, 3]
```

In the example above, a and b have a shared reference to the same list object so a change to that list object affects both references. If the shared reference effects are not well understood the change to b can cause unexpected results.

Automatic conversion occurs only for numeric types of objects. Python converts (coerces) from the simplest type up to the most complex type whenever different numeric types are mixed in an expression. For example:

```
a = 1
b = 2.0
c = a + b; print(c) #=> 3.0
```

In the example above, the integer a is converted up to floating point (i.e., 1.0) before the operation is performed. The object referred to by a is not affected – only the intermediate values used to resolve the expression are converted. If the programmer does not realize this conversion takes place he may expect that c is an integer and use it accordingly which could lead to unexpected results.

Automatic conversion also occurs when an integer becomes too large to fit within the constraints of the large integer specified in the language (typically C) used to create the Python interpreter. When an integer becomes too large to fit into that range it is converted to an unlimited precision integer of arbitrary length.

Explicit conversion methods can also be used to explicitly convert between types though this is seldom required since Python will automatically convert as required. Examples include:

```
a = int(1.6666) # a converted to 1
b = float(1) # b converted to 1.0
c = int('10') # c integer 10 created from a string
d = str(10) # d string '10' created from an integer
e = ord('x') # e integer assigned integer value 120
f = chr(121) # f assigned the string 'y'
```

Dynamic typing is a key feature of Python which promotes polymorphism for flexibility. Strict typing can, however, be imposed:

```
a = 'abc' # a refers to a string object
if isinstance(a, str): print('a type is string')
```

Using code to explicitly check the type of an object is strongly discouraged in Python since it defeats the benefit that dynamic typing provides - flexibility which allows functions to potentially operate correctly with objects of more than one type.

E.3.2 Guidance to language users

- Pay special attention to issues of magnitude and precision when using mixed type expressions;
- Be aware of the consequences of shared references;
- Be aware of the conversion from simple to complex; and
- Do not check for specific types of objects unless there is good justification, for example, when calling an extension that requires a specific type.

E.4 Bit Representations [STR]

E.4.1 Applicability to language

Python provides hexadecimal, octal and binary built-in functions. `oct` converts to octal, `hex` to hexadecimal and `bin` to binary:

```
print(oct(256)) # 0o400
print(hex(256)) # 0x100
print(bin(256)) # 0b100000000
```

The notations shown as comments above are also valid ways to specify octal, hex and binary values respectively:

```
print(0o400)# => 256
a=0x100+1; print(a)# => 257
```

The built-in `int` function can be used to convert strings to numbers and optionally specify any number base:

```
int('256') # the integer 256 in the default base 10
int('400', 8) # => 256
int('100', 16) # => 256
int('24', 5) # => 14
```

Python stores integers that are beyond the implementation's largest integer size as an internal arbitrary length so that programmers are only limited by performance concerns when very large integers are used (and by memory when extremely large numbers are used). For example:

```
a=2**100 # => 1267650600228229401496703205376
```

Python treats positive integers as being infinitely padded on the left with zeroes and negative numbers (in two's complement notation) with 1's on the left when used in bitwise operations:

```
a<<b # a shifted left b bits
a>>b # a shifted right b bits
```

There is no overflow check for shifting left or right so a program expecting an exception to halt it will instead unexpectedly continue leading to unexpected results.

E.4.2 Guidance to language users

- Keep in mind that using a very large integer will have a negative effect on performance; and
- Don't use bit operations to simulate multiplication and division.

E.5 Floating-point Arithmetic [PLF]

E.5.1 Applicability to language

Python supports floating-point arithmetic. Literals are expressed with a decimal point and or an optional `e` or `E`:

```
1., 1.0, .1, 1.e0
```

There is no way to determine the precision of the implementation from within a Python program. For example, in the CPython implementation, it's implemented as a C double which is approximately 53 bits of precision.

E.5.2 Guidance to language users

- Use floating-point arithmetic only when absolutely needed;
- Do not use floating-point arithmetic when integers or booleans would suffice;
- Be aware that precision is lost for some real numbers (i.e., floating-point is an approximation with limited precision for some numbers);
- Be aware that results will frequently vary slightly by implementation (see E.52 for more on this subject); and
- Testing floating-point numbers for equality (especially for loops) can lead to unexpected results. Instead, if floating-point numbers are needed for loop control use >= or <= comparisons.

E.6 Enumerator Issues [CCB]

E.6.1 Applicability to language

Python has an `enumerate` built-in type but it is not at all related to the implementation of enumeration as defined in other languages where constants are assigned to symbols. Given that enumeration is a useful programming device and that there is no enumeration construct in Python, many programmers choose to implement their own "enum" objects or types using a wide variety of methods including the creation of "enum" classes, lists, and even dictionaries. One simple method is to simply assign a list of names to integers:

```
Red, Green, Blue = range (3)
print(Red, Green, Blue) # => 0 1 2
```

Code can then reference these "enum" values as they would in other languages which have native support for enumeration:

```
a = 1
if a == Green: print("a=Green")# => a=Green
```

There are disadvantages to the approach above though since any of the "enum" variables could be assigned new values at any time thereby undoing their intended role as "pseudo" constants. There are many forum discussions and articles that illustrate other, safer ways to simulate enumeration which are beyond the scope of this annex.

E.6.2 Guidance to language users

Use of enumeration requires careful attention to readability, performance, and safety. There are many complex, but useful ways to simulate enums in Python [(Enums for Python (Python recipe))]and many simple ways including the use of sets:

```
colors = {'red', 'green', 'blue'}
if "red" in colors: print('valid color')
```

Be aware that the technique shown above, as with almost all other ways to simulate enums, is not safe since the variable can be bound to another object at any time.

E.7 Numeric Conversion Errors [FLC]

E.7.1 Applicability to language

Python converts numbers to a common type before performing any arithmetic operations. The common type is coerced using the following rules as defined in the standard (http://docs.python.org/release/1.4/ref/ref5.html):

> If either argument is a complex number, the other is converted to the complex type;
> otherwise, if either argument is a floating point number, the other is converted to floating point;
> otherwise, if either argument is a long integer, the other is converted to long integer;
> otherwise, both must be plain integers and no conversion is necessary.

Integers in the Python language are of a length bounded only by the amount of memory in the machine. Integers are stored in an internal format that has faster performance when the number is smaller than the largest integer supported by the implementation language and platform.

Implicit or explicit conversion floating point to integer, implicitly (or explicitly using the `int` function), will typically cause a loss of precision:

```
a = 3.0; print(int(a))# => 3 (no loss of precision)
a = 3.1415; print(int(a))# => 3 (precision lost)
```

Precision can also be lost when converting from very large integer to floating point. Losses in precision, whether from integer to floating point or vice versa, do not generate errors but can lead to unexpected results especially when floating point numbers are used for loop control.

E.7.2 Guidance to language users

- Though there is generally no need to be concerned with an integer getting too large (rollover) or small, be aware that iterating or performing arithmetic with very large positive or small (negative) integers will hurt performance; and
- Be aware of the potential consequences of precision loss when converting from floating point to integer.

E.8 String Termination [CJM]

In Python strings are immutable objects whose length can be queried with built-in functions therefore Python does not permit accesses past the end, or beginning, of a string.

```
a = '12345'
b = a[5] #=> IndexError: string index out of range
```

E.9 Buffer Boundary Violation [HCB]

This vulnerability is not applicable to Python because Python's run-time checks the boundaries of arrays and raises an exception when an attempt is made to access beyond a boundary.

E.10 Unchecked Array Indexing [XYZ]

This vulnerability is not applicable to Python because Python's run-time checks the boundaries of arrays and raises an exception when an attempt is made to access beyond a boundary.

E.11 Unchecked Array Copying [XYW]

This vulnerability is not applicable to Python because Python's run-time checks the boundaries of arrays and raises an exception when an attempt is made to access beyond a boundary.

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

This vulnerability is not applicable to Python because Python does not use pointers.

E.13 Pointer Arithmetic [RVG]

This vulnerability is not applicable to Python because Python does not use pointers.

E.14 Null Pointer Dereference [XYH]

This vulnerability is not applicable to Python because Python does not use pointers.

E.15 Dangling Reference to Heap [XYK]

This vulnerability is not applicable to Python because Python does not use pointers. Specifically, Python only uses namespaces to access objects therefore when an object is deallocated, any reference to it causes an exception to be raised.

E.16 Arithmetic Wrap-around Error [FIF]

E.16.1 Applicability to language

Operations on integers in Python cannot cause wrap-around errors because integers have no maximum size other than what the memory resources of the system can accommodate.

Normally the `OverflowError` exception is raised for floating point wrap-around errors but, for implementations of Python written in C, exception handling for floating point operations cannot be assumed to catch this type of error because they are not standardized in the underlying C language. Because of this, most floating point operations cannot be depended on to raise this exception.

E.16.2 Guidance to language users

- Be cognizant that most arithmetic and bit manipulation operations on non-integers have the potential for undetected wrap-around errors.
- Avoid using floating point or decimal variables for loop control but if you must use these types then bound the loop structures so as to not exceed the maximum or minimum possible values for the loop control variables.

- Test the implementation that you are using to see if exceptions are raised for floating point operations and if they are then use exception handling to catch and handle wrap-around errors.

E.17 Using Shift Operations for Multiplication and Division [PIK]

E.17.1 Applicability to language

This vulnerability is not applicable to Python because it does not check for overflow. In addition there is no practical way to overflow an integer since integers have unlimited precision.

```
>>> print(-1<<100)#=> -1267650600228229401496703205376
>>> print(1<<100) #=>  1267650600228229401496703205376
```

E.18 Sign Extension Error [XZI]

This vulnerability is not applicable to Python because Python converts between types without ever extending the sign.

E.19 Choice of Clear Names [NAI]

E.19.1 Applicability to language

Python provides very liberal naming rules:

- Names may be of any length and consist of letters, numerals, and underscores only. All characters in a name are significant. Note that unlike some other languages where only the first *n* number of characters in a name are significant, ***all*** characters in a Python name are significant. This eliminates a common source of name ambiguity when names are identical up to the significant length and vary afterwards which effectively makes all such names a reference to one common variable.
- All names must start with an underscore or a letter; and
- Names are case sensitive, for example, `Alpha`, `ALPHA`, and `alpha` are each unique names. While this is a feature of the language that provides for more flexibility in naming, it is also can be a source of programmer errors when similar names are used which differ only in case, for example, `aLpha` versus `alpha`.

The following naming conventions are not part of the standard but are in common use:

- Class names start with an upper case letter, all other variables, functions, and modules are in all lower case;
- Names starting with a single underscore (_) are not imported by the `from` *`module`* `import *` statement – this not part of the standard but most implementations enforce it; and
- Names starting and ending with two underscores (__) are system-defined names.
- Names starting with, but not ending with, two underscores are local to their class definition
- Python provides a variety of ways to package names into namespaces so that name clashes can be avoided:
- Names are scoped to functions, classes, and modules meaning there is normally no collision with names utilized in outer scopes and vice versa; and

- Names in modules (a file containing one or more Python statements) are local to the module and are referenced using qualification (e.g., a function `x` in module `y` is referenced as `y.x`). Though local to the module, a module's names can be, and routinely are, copied into another namespace with a `from module import` statement.

Python's naming rules are flexible by design but are also susceptible to a variety of unintentional coding errors:

- Names are never declared but they must be assigned values before they are referenced. This means that some errors will never be exposed until runtime when the use of an unassigned variable will raise an exception (see E.23).
- Names can be unique but may look similar to other names, for example, `alpha` and `aLpha`, `__x` and `_x`, `_beta__` and `__beta_` which could lead to the use of the wrong variable. Python will not detect this problem at compile-time.

Python utilizes dynamic typing with types determined at runtime. There are no type or variable declarations for an object ,which can lead to subtle and potentially catastrophic errors:

```
x = 1
# lots of code…
if some rare but important case:
    X = 10
```

In the code above the programmer intended to set (lower case) `x` to 10 and instead created a new *upper case* `X` to `10` so the *lower case* `x` remains unchanged. Python will not detect a problem because there is no problem – it sees the upper case `X` assignment as a legitimate way to bring a *new* object into existence. It could be argued that Python could statically detect that `X` is never referenced and therefore indicate the assignment is dubious but there are also cases where a dynamically defined function defined downstream could legitimately reference `X` as a `global`.

E.19.2 Guidance to language users

- For more guidance eon Python's naming conventions, refer to Python Style Guides contained in PEP 8 at http://www.python.org/dev/peps/pep-0008/ .
- Avoid names that differ only by case unless necessary to the logic of the usage;
- Adhere to Python's naming conventions;
- Do not use overly long names;
- Use names that are not similar (especially in the use of upper and lower case) to other names;
- Use meaningful names; and
- Use names that are clear and visually unambiguous because the compiler cannot assist in detecting names that appear similar but are different.

E.20 Dead Store [WXQ]

E.20.1 Applicability to language

It is possible to assign a value to a variable and never reference that variable which causes a “dead store”. This in itself is not harmful, other than the memory that it wastes, but if there is a substantial amount of dead stores then performance could suffer or, in an extreme case, the program could halt due to lack of memory.

Python provides the ability to dynamically create variables when they are first assigned a value. In fact, assignment is the *only* way to bring a variable into existence. All values in a Python program are accessed through a reference which refers to a memory location which is always an object (e.g., number, string, list etc.). A variable is said to be bound to an object when it is assigned to that object. A variable can be rebound to another object which can be of any type. For example:

```
a = 'alpha' # assignment to a string
a = 3.142 # rebinding to a float
a = b = (1, 2, 3) # rebinding to a tuple
print(a) # => (1, 2, 3)
del a
print(b)# => (1, 2, 3)
      print(a)# => NameError: name 'a' is not defined
```

The first three statements show dynamic binding in action. The variable `a` is bound to a string, then to a float, then to another variable which in turn is assigned a tuple of value `(1, 2, 3)`. The `del` statement then unbinds the variable `a` from the tuple object which effectively deletes the `a` variable (if there were no other references to the tuple object it too would have been deleted because an object with zero references is *marked* for garbage collection (but is not necessarily actually deleted immediately)). But in this case we see that `b` is still referencing the tuple object so the tuple is not deleted. The final statement above shows that an exception is raised when an unbound variable is referenced.

The way in which Python dynamically binds and rebinds variables is a source of some confusion to new programmers and even experienced programmers who are used to static binding where a variable is permanently bound to a single memory location.

The Python language, by design, allows for dynamic binding and rebinding. Because Python performs a syntactic analysis and not a semantic analysis (with one exception which is covered in E.22.1 Namespace Issues [BJL] Applicability to language) and because of the dynamic way in which variables are brought into a program at run-time, Python cannot warn that a variable is referenced but never assigned a value. The following code illustrates this:

```
if a > b:
    import x
else:
    import y
```

Depending on the current value of `a` and `b`, either module `x` or `y` is imported into the program. If `x` assigns a value to a variable `z` and module `y` references `z` then, dependent on which import statement is executed first

(an import always executes all code in the module when it is first imported), an unassigned variable reference exception will or will not be raised.

E.20.2 Guidance to language users

- Avoid rebinding except where it adds value;
- Ensure that when examining code that you take into account that a variable can be bound (or rebound) to another object (of same or different type) at any time; and
- Variables local to a function are deleted automatically when the encompassing function is exited but, though not a common practice, you can also explicitly delete variables using the `del` statement when they are no longer needed.

E.21 Unused Variable [YZS]

The applicability to language and guidance to language users sections of the E.19 write-up are applicable here.

E.22 Identifier Name Reuse [YOW]

E.22.1 Applicability to language

Python has the concept of namespaces which are simply the places where names exist in memory. Namespaces are associated with functions, classes, and modules. When a name is created (i.e., when it is first assigned a value), it is associated (i.e., bound) to the namespace associated with the location where the assignment statement is made (e.g., in a function definition). The association of a variable to a specific namespace is elemental to how scoping is defined in Python.

Scoping allows for the definition of more than one variable with the same name to reference different objects. For example:

```
a = 1
def x():
    a = 2
    print(a)#=> 2
print(a) #=> 1
```

The `a` variable within the function `x` above is local to the function only – it is created when `x` is called and disappears when control is returned to the calling program. If the function needed to update the outer variable named `a` then it would need to specify that `a` was a global before referencing it as in:

```
a = 1
def x():
    global a
    a = 2
    print(a)#=> 2
print(a) #=> 2
```

In the case above, the function is updating the variable `a` that is defined in the calling module. There is a subtle but important distinction on the locality versus global nature of variables: *assignment* is always local unless `global` is specified for the variable as in the example above where `a` is *assigned* a value of `2`. If the function had instead simply *referenced* `a` without assigning it a value, then it would reference the topmost variable `a` which, by definition, is always a global:

```
a = 1
def x():
    print(a)
x() #=> 1
```

The rule illustrated above is that attributes of modules (i.e., variable, function, and class names) are global to the module meaning any function or class can reference them.

Scoping rules cover other cases where an identically named variable name references different objects:

- A nested function's variables are in the scope of the nested function only; and
- Variables defined in a module are in *global* scope which means they are scoped to the module only and are therefore not visible within functions defined in that module (or any other function) unless explicitly identified as `global` at the start of the function.

Python has ways to bypass implicit scope rules:

- The `global` statement which allows an inner reference to an outer scoped variable(s); and
- The `nonlocal` statement which allows an enclosing function definition to reference a nested function's variable(s).

The concept of scoping makes it safer to code functions because the programmer is free to select any name in a function without worrying about accidentally selecting a name assigned to an outer scope which in turn could cause unwanted results. In Python, one must be explicit when intending to circumvent the intrinsic scoping of variable names. The downside is that identical variable names, which are totally unrelated, can appear in the same module which could lead to confusion and misuse unless scoping rules are well understood.

Names can also be qualified to prevent confusion as to which variable is being referenced:

```
a = 1
class xyz():
    a = 2
    print(a)#=> 2
print(xyz.a, a) #=> 2 1
```

The final `print` function call above references the `a` variable within the `xyz` class and the global `a`.

E.22.2 Guidance to language users

- Do not use identical names unless necessary to reference the correct object;
- Avoid the use of the `global` and `nonlocal` specifications because they are generally a bad programming practice for reasons beyond the scope of this annex and because their bypassing of

standard scoping rules make the code harder to understand; and

- Use qualification when necessary to ensure that the correct variable is referenced.

E.23 Namespace Issues [BJL]

E.23.1 Applicability to language

Python has a hierarchy of namespaces which provides isolation to protect from name collisions, ways to explicitly reference down into a nested namespace, and a way to reference up to an encompassing namespace. Generally speaking, namespaces are very well isolated. For example, a program's variables are maintained in a separate namespace from any of the functions or classes it defines or uses. The variables of modules, classes, or functions are also maintained in their own protected namespaces.

Accessing a namespace's attribute (i.e., a variable, function, or class name), is generally done in an explicit manner to make it clear to the reader (and Python) which attribute is being accessed:

```
n = Animal.num # fetches a class' variable called num
x = mymodule.y # fetches a module's variable called y
```

The examples above exhibit qualification – there is no doubt where a variable is being fetched from. Qualification can also occur from an encompassed namespace up to the encompassing namespace using the global statement:

```
def x():
    global y
    y = 1
```

The example above uses an explicit `global` statement which makes it clear that the variable `y` is not local to the function `x;` it assigns the value of `1` to the variable `y` in the encompassing module[16].

Python also has some subtle namespace issues that can cause unexpected results especially when using imports of modules. For example, assuming module `a.py` contains:

```
a = 1
```

And module `b.py` contains:

```
b = 1
```

Executing the following code is not a problem since there is no variable name collision in the two modules (the `from` *`modulename`* `import *` statement brings all of the attributes of the named module into the local namespace):

```
from a import *
print(a) #=> 1
```

[16] *V*alues are assigned to objects which in turn are referenced by variables but it's simpler to say the value is assigned to the variable. Also, the encompassing code could be at a prompt level instead of a module. For brevity this annex uses this simpler, though not as exact, wording.

```
from b import *
print(b) #=> 1
```

Later on the author of the `b` module adds a variable named `a` and assigns it a value of `2`. `B.py` now contains:

```
b = 1
a = 2 # new assignment
```

The programmer of module `b.py` may have no knowledge of the `a` module and may not consider that a program would import both `a` and `b`. The importing program, with no changes, is run again:

```
from a import *
print(a) #=> 1
from b import *
print(a) #=> 2
```

The results are now different because the importing program is susceptible to unintended consequences due to changes in variable assignments made in two unrelated modules as well as the sequence in which they were imported. Also note that the `from` *`modulename`* `import *` statement brings all of the modules attributes into the importing code which can silently overlay like-named variables, functions, and classes.

A common misunderstanding of the Python language is that Python detects local names (a local name is a name that lives within a class or function's namespace) *statically* by looking for one or more assignments to a name within the class/function. If one or more assignments are found then the name is noted as being local to that class/function. This can be confusing because if only *references* to a name are found then the name is referencing a global object so the only way to know if a reference is local or global, barring an explicit global statement, is to examine the entire function definition looking for an assignment. This runs counter to Python's goal of Explicit is Better Than Implicit (EIBTI):

```
a = 1
def f():
 print(a)
 a = 2
f() #=> UnboundLocalError: local variable 'a' referenced before
        assignment
# now with the assignment commented out
a = 1
def f():
 print(a)#=> 1
 #a = 2
# Assuming a new session:
a = 1
def f():
    global a
    a = 2
f()
print(a)#=> 2
```

Note that the rules for determining the locality of a name applies to the assignment operator = as above, but also to all other kinds of assignments which includes module names in an `import` statement, function and class names, and the arguments declared for them. See E.21 for more detail on this.

Name resolution follows a simple Local, Enclosing, Global, Built-ins (LEGB) sequence:

- First the local namespace is searched;
- Then the enclosing namespace (i.e. a `def` or `lambda` (A `lambda` is a single expression function definition));
- Then the global namespace; and
- Lastly the built-in's namespace.

E.23.2 Guidance to language users

- When practicable, consider using the `import` statement without the `from` clause. This forces the importing program to use qualification to access the imported module's attributes;
- When using the `import` statement, rather than use the `from *` form (which imports all of the module's attributes into the importing program's namespace), instead use the `from` clause to explicitly name the attributes that you want to import (e.g., `from alpha import a, b, c`) so that variables, functions and classes are not inadvertently overlaid; and
- Avoid implicit references to global values from within functions to make code clearer. In order to update globals within a function or class, place the `global` statement at the beginning of the function definition and list the variables so it is clearer to the reader which variables are local and which are global (e.g., `global a, b, c`).

E.24 Initialization of Variables [LAV]

E.24.1 Applicability of language

Python does not check to see if a statement references an uninitialized variable until runtime. This is by design in order to support dynamic typing which in turn means there is no ability to declare a variable. Python therefore has no way to know if a variable is referenced before or after an assignment. For example:

```
If y > 0:
    print(x)
```

The above statement is legal at compile time even if `x` is not defined (i.e., assigned a value). An exception is raised at runtime only if the statement is executed and `y>0`. This scenario does not lend itself to static analysis because, as in the case above, it may be perfectly logical to not ever print `x` unless `y>0`.

There is no ability to use a variable with an uninitialized value because *assigned* variables always reference objects which always have a value and *unassigned* variables do not exist. Therefore Python raises an exception when an unassigned (i.e., non-existent) variable is referenced.

Initialization of class arguments can cause unexpected results when an argument is set to a default object which is mutable:

```
def x(y=[]):
    y.append(1)
    print(y)
x([2])#=> [2, 1], as expected (default was not needed)
x() # [1]
x() # [1, 1] continues to expand with each subsequent call
```

The behaviour above is not a bug - it is a defined behaviour for mutable objects but it's a very bad idea in almost all cases to assign default values to mutable objects.

E.24.2 Guidance to language users

- Ensure that it is not logically possible to reach a reference to a variable before it is assigned. The example above illustrates just such a case where the programmer wants to print the value of `x` but has not assigned a value to `x` – this proves that there is missing, or bypassed, code needed to provide `x` with a meaningful value at runtime.

E.25 Operator Precedence/Order of Evaluation [JCW]

E.25.1 Applicability to language

Python provides many operators and levels of precedence so it is not unexpected that operator precedence and order of operation are not well understood and hence misused. For example:

```
1 + 2 * 3 #=> 7, evaluates as 1 + (2 * 3)
(1 + 2) * 3 #=> 9, parenthesis are allowed to coerce precedence
```

Expressions that use and or or are evaluated left to right which can cause a short circuit:

```
a or b or c
```

In the expression above `c` is never evaluated if either `a` or `b` evaluate to `True` because the entire expression evaluates to `True` immediately when any sub expression evaluates to `True`. The short circuit effect is non-consequential above but in the case below the effect is subtle and potentially destructive:

```
def x(i):
    if i:
        return True
    else:
        1/0  # Hard stop
a = 1
b = 0
while True:
    if x(a) or x(b):
        print('a or b is True')
```

The code above will go into an endless loop because `x(b)` is never evaluated. If it was the program would terminate due to an attempted division by zero.

E.25.2 Guidance to language users

- Use parenthesis liberally to force intended precedence and increase readability;
- Be aware that short-circuited expressions can cause subtle errors because not all sub-expressions may be evaluated; and
- Break large/complex statements into smaller ones using temporary variables for interim results.

E.26 Side-effects and Order of Evaluation [SAM]

E.26.1 Applicability to language

Python supports sequence unpacking (parallel assignment) in which each element of the right hand side (expressed as a tuple) is evaluated and then assigned to each element of the left-hand side (LHS) in left to right sequence. For example, the following is a safe way to exchange values in Python:

```
a = 1
b = 2
a, b = b, a # swap values between a and b
print (a,b)#=> 2, 1
```

Assignment of the targets (LHS) proceeds left to right so overlaps on the left side are not safe:

```
a = [0,0]
i = 0
i, a[i] = 1, 2 #=> Index is set to 1; list is updated at [1]
print(a) #=> 0,2
```

Python Boolean operators are often used to assign values as in:

```
a = b or c or d or None
```

`a` is assigned the first value of the first object that has a non-zero (i.e., `True`) value or, in the example above, the value `None` if `b`, `c`, and `d` are all `False`. This is a common and well understood practice. However, trouble can be introduced when functions or other constructs with side effects are used on the right side of a Boolean operator:

```
if a() or b()
```

If function `a` returns a `True` result then function `b` will not be called which may cause unexpected results.

E.26.2 Guidance to language users

- Be aware of Python's short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression; if necessary perform each expression first and then evaluate the results:

```
x = a()
y = b()
if x or y …
```

- Be aware that, even though overlaps between the left hand side and the right hand side are safe, it is possible to have unintended results when the variables on the left side overlap with one another so always ensure that the assignments and left to right sequence of assignments to the variables on the left hand side never overlap. If necessary, and/or if it makes the code easier to understand, consider breaking the statement into two or more statements;

```
# overlapping
a = [0,0]
i = 0
i, a[i] = 1, 2 #=> Index is set to 1; list is updated at [1]
print(a) #=> 0,2
# Non-overlapping
a = [0,0]
i, a[0] = 1, 2
print(a) #=> 2,0
```

E.27 Likely Incorrect Expression [KOA]

E.27.1 Applicability to language

Python goes to some lengths to help prevent likely incorrect expressions:

- Testing for equivalence cannot be confused with assignment:

```
a = b = 1
if (a=b): print(a,b) #==> syntax error
if (a==b): print(a,b) #==> 1 1
```

- Boolean operators use English words `not`, `and`, `or`; bitwise operators use symbols `~`, `&`, `|` respectively. However Python does have some subtleties that can cause unexpected results:
  - Skipping the parentheses after a function does not invoke a call to the function and will fail silently because it's a legitimate reference to the function object:

```
class a:
  def demo():
     print("in demo")
a.demo()#=> in demo
a.demo  #=> <function demo at 0x000000000342A9C8>
x = a.demo
x() #=> in demo
```

    The two lines that reference the function without trailing parentheses above demonstrate how that syntax is a reference to the function *object* and not a call to the function.

- Built-in functions that perform in-place operations on mutable objects (i.e., lists, dictionaries, and some class instances) do not return the changed object – they return `None`:

```
a = []
a.append("x")
```

```
print(a) #=> ['x']
a = a.append("y")
print(a) #=> None
```

E.27.2 Guidance to language users

- Be sure to add parentheses after a function call in order to invoke the function; and
- Keep in mind that any function that changes a mutable object in place returns a `None` object – not the changed object since there is no need to return an object because the object has been changed by the function.

E.28 Dead and Deactivated Code [XYQ]

E.28.1 Applicability to language

There are many ways to have dead or deactivated code occur in a program and Python is no different in that regard. Further, Python does not provide static analysis to detect such code nor does the very dynamic design of Python's language lend itself to such analysis.

The module and related `import` statement provides convenient ways to group attributes (e.g., functions, names, and classes) into a file which can then be copied, in whole, or in part (using the `from` statement), into another Python module. All of the attributes of a module are copied when either of the following forms of the `import` statement is used. This is roughly equivalent to simply copying in all of code directly into the importing program which can result in code that is never invoked (e.g., functions which are never called and hence "dead"):

```
import modulename
from modulename import *
```

The `import` statement in Python loads a module into memory, compiles it into byte code, and then executes it. Subsequent executions of an import for that same module are ignored by Python and have no effect on the program whatsoever. The `reload` statement is required to force a module, and its attributes, to be loaded, compiled, and executed.

E.28.2 Guidance to language users

- Import just the attributes that are required by using the `from` statement to avoid adding dead code; and
- Be aware that subsequent imports have no effect; use the `reload` statement instead if a fresh copy of the module is desired.

E.29 Switch Statements and Static Analysis [CLL]

E.29.1 Applicability to language

By design Python does not have a switch statement nor does it have the concept of labels or branching to a demarcated "place". Python enforces structure by not providing these constructs but it also provides several statements to select actions to perform based on the value of a variable or expression. The first of these are the

`if/elif/else` statements which operate as they do in other languages so this warrants no further coverage here.

Python provides a `break` statement which allows a loop to be broken with an immediate branch to the first statement after the loop body:

```
a = 1
while True:
    if a > 3:
         break
    else:
        print(a)
        a += 1
```

The loop above prints 1, 2 and 3, each on separate lines, then terminates upon execution of the `break` statement.

E.29.2 Guidance to language users

Use `if/elseif/else` statements to provide the equivalent of switch statements.

E.30 Demarcation of Control Flow [EOJ]

E.30.1 Applicability to language

Python makes demarcation of control flow very clear because it uses indentation (using spaces or tabs – but not both) and dedentation as the *only* demarcation construct:

```
a, b = 1, 1
if a:
    print("a is True")
else:
    print("False")
    if b:
        print("b is true")
print("back to main level")
```

The code above prints "`a is True`" followed by "`back to main level`". Note how control is passed from the first `if` statement's `True` path to the main level based entirely on indentation while in most other languages the final line would execute only when the second `if` evaluated to `True`.

E.30.2 Guidance to language users

Use only spaces or tabs, not both, to indent to demark control flow.

E.31 Loop Control Variables [TEX]

E.31.1 Applicability to language

Python provides two loop control statements: `while` and `for`. They each support very flexible control constructs beyond a simple loop control variable. Assignments in the loop control statement (i.e., `while` or `for`) which can be a frequent source of problems, are not allowed in Python – Python's loop control statements use expressions which *cannot* contain assignment statements.

The `while` statement leaves the loop control entirely up to the programmer as in the example below:

```
a = 1
while a:
    print('in loop')
    a = False # force loop to end after one iteration
else:
    print('exiting loop')
```

The `for` statement is unusual in that it does not provide a loop control variable therefore it is not possible to vary the sequence or number of loops that are performed other than by the use of the `break` statement (covered in E.29) which can be used to immediately branch to the statement after the loop block.

When using the `for` statement to iterate though an iterable object such as a list, there is no way to influence the loop "count" because it's not exposed. The variable `a` in the example below takes on the value of the first, then the second, then the third member of the list:

```
x = ['a', 'b', 'c']
for a in x:
    print(a)
#=>a
#=>b
#=>c
```

It is possible, though not recommended, to change a mutable object as it is being traversed which in turn changes the number of loops performed. In the case below the loop is performed only two times instead of the three times had the list been left intact:

```
x = ['a', 'b', 'c']
for a in x:
    print(a)
    del x[0]
print(x)
#=> a
#=> c
#=> ['c']
```

E.31.2 Guidance to language users

- Be careful to only modify loop control variables in ways that are easily understood and in ways that cannot lead to a premature exit or an endless loop.
- When using the `for` statement to iterate through a mutable object, do not add or delete members because it could have unexpected results.

E.32 Off-by-one Error [XZH]

E.32.1 Applicability to language

The Python language itself is vulnerable to off by one errors as is any language when used carelessly or by a person not familiar with Python's index from zero versus from one. Python does not prevent off by one errors but its runtime bounds checking for strings and lists does lessen the chances that doing so will cause harm. It is also not possible to index past the end or beginning of a string or list by being off by one because Python does not use a sentinel character and it always checks indexes before attempting to index into strings and lists and raises an exception when their bounds are exceeded.

E.32.2 Guidance to language users

- Be aware of Python's indexing from zero and code accordingly.

E.33 Structured Programming [EWD]

E.33.1 Applicability to language

Python is designed to make it simpler to write structured program by requiring indentation and dedentation to show scope of control in blocks of code:

```
a = 1
b = 1
if a == b:
    print("a == b")#=> a == b
    if a > b:
        print("a > b")
else:
    print("a != b")
```

In many languages the last `print` statement would be executed because they associate the `else` with the immediately prior `if` while Python uses indentation to link the `else` with its associated `if` statement (i.e. the one *above* it).

Python also encourages structured programming by *not* introducing any language constructs which could lead to unstructured code (e.g., GO TO statements).

Python does have two statements that could be viewed as unstructured. The first is the `break` statement. It's used in a loop to exit the loop and continue with the first statement that follows the last statement within the

loop block. This is a type of branch but it is such a useful construct that few would consider it “unstructured” or a bad coding practice.

The second is the `try/except` block which is used to trap and process exceptions. When an exception is thrown a branch is made to the `except` block:

```
def divider(a,b):
    return a/b
try:
    print(divider(1,0))
except ZeroDivisionError:
    print('division by zero attempted')
```

E.33.2 Guidance to language users

- Python offers few constructs that could lead to unstructured code. However, judicious use of `break` statements is encouraged to avoid confusion.

E.34 Passing Parameters and Return Values [CSJ]

E.34.1 Applicability to language

Python’s only subprogram type is the function. Even though the `import` statement does execute the imported module’s top level code (the first time it is imported), the `import` statement cannot effectively be used as a way to repeatedly execute a series of statements

Python passes arguments by assignment which is similar to passing by pointer or reference. Python assigns the passed arguments to the function’s local variables but unlike some other languages, simply having the address of the caller’s argument does not automatically allow the called function to change any of the objects referenced by those arguments – only *mutable* objects referenced by passed arguments can be changed. Python has no concept of aliasing where a function’s variables are mapped to the caller’s variables such that any changes made to the function’s variables are mapped over to the memory location of the caller’s arguments.

```
a = 1
def f(x):
    x += 1
    print(x)#=> 2
f(a)
print(a)#=> 1
```

In the example above, an immutable integer is passed as an argument and the function’s local variable is updated and then discarded when the function goes out of scope therefore the object the caller’s argument references is not affected. In the example below, the argument is mutable and is therefore updated in place:

```
a = [1]
def f(x):
    x[0] = 2
f(a)
```

```
print(a)#=> [2]
```

Note that the list object `a` is not changed – it's the same object but its content at index `0` has changed.

The `return` statement can be used to return a value for a function:

```
def doubler(x):
   return x * 2
x = 1
x = doubler(x)
print(x)#=>
```

The example above also demonstrates a way to emulate a call by reference by assigning the returned object to the passed argument. This is not a true call by reference and Python does not replace the value of the object `x`, rather it creates a new object `x` and assigns it the value returned from the `doubler` function as proven by the code below which displays the address of the initial and the new object `x`:

```
def doubler(x):
   return x * 2
x = 1
print(id(x)) #=> 506081728
x = doubler(x)
print(id(x)) #=> 506081760
```

The object replacement process demonstrated above follows Python's normal processing of *any* statement which changes the value of an immutable object and is not a special exception for function returns.

Note that Python functions return a value of `none` when no `return` statement is executed or when a `return` with no arguments is executed.

E.34.2 Guidance to language users

- Create copies of mutable objects before calling a function if changes are not wanted to mutable arguments; and
- If a function wants to ensure that it does not change mutable arguments it can make copies of those arguments and operate on them instead.

E.35 Dangling References to Stack Frames [DCM]

This vulnerability is not applicable to Python because, while Python does provide a way to inspect the address of an object, for example, the `id` function, it does not provide a way to use that address to access an object.

E.36 Subprogram Signature Mismatch [OTR]

E.36.1 Applicability to language

Python supports positional, *"keyword=value"*, or both kinds of arguments. It also supports variable numbers of arguments and, other than the case of variable arguments, will check at runtime for the correct number of arguments making it impossible to corrupt the call stack in Python when using standard modules.

Python has extensive extension and embedding APIs that includes functions and classes to use when extending or embedding Python. These provide for subprogram signature checking at runtime for modules coded in non-Python languages. Discussion of this API is beyond the scope of this annex but the reader should be aware that improper coding of any non-Python modules or their interface could cause a call stack problem

E.36.2 Guidance to language users

- Ensure that you are using a trusted source when using non-standard imported modules.

E.37 Recursion [GDL]

E.37.1 Applicability to language

Recursion is supported in Python and is, by default, limited to a depth of 1,000 which can be overridden using the `setrecursionlimit` function. If the limit is set high enough, a runaway recursion could exhaust all memory resources leading to a denial of service.

E.37.2 Guidance to language users

- Avoid recursion when practical unless the bounds are well-defined and protected; and
- Consider utilizing the default limit of recursion to a level of 1,000 (default limit) or less by modifying the limit using the `setrecursionlimit` function.

E.38 Ignored Error Status and Unhandled Exceptions [OYB]

E.38.1 Applicability to language

Python provides statements to handle exceptions which considerably simplify the detection and handling of exceptions. Rather than being a vulnerability, Python's exception handling statements provide a way to foil denial of service attacks:

```
def mainpgm(x, y):
    return x/y
for x in range(3):
    try:
        y = mainpgm(1,x)
    except:
        print('Problem in mainpgm')
        # clean up code…
```

```
else:
    print (y)
```

The example code above prints:

```
Problem in mainpgm
1.0
0.5
```

The idea above is to ensure that the main program, which could be a web server, is allowed to continue to run after an exception by virtue of the `try/except` statement pair.

E.38.2 Guidance to language users

- Use Python's exception handling with care in order to not catch errors that are intended for other exception handlers; and
- Use exception handling, but directed to specific tolerable exceptions, to ensure that crucial processes can continue to run even after certain exceptions are raised.

E.39 Termination Strategy [REU]

E.39.1 Applicability to language

Python has a rich set of exception handling statements which can be utilized to implement a termination strategy that assures the best possible outcome ranging from a hard stop to a clean-up and fail soft strategy. Refer to E.38 for an example of an implementation that cleans up and continues.

E.39.2 Guidance to language users

- Use Python's exception handling statements to implement an appropriate termination strategy.

E.40 Type-breaking Reinterpretation of Data [AMV]

This vulnerability is not applicable to Python because assignments are made to objects and the object always holds the type – not the variable, therefore all referenced objects has the same type and there is no way to have more than one type for any given object.

E.41 Memory Leak [XYL]

E.41.1 Applicability to language

Python supports automatic garbage collection so in theory it should not have memory leaks. However, there are at least three general cases in which memory can be retained after it is no longer needed. The first is when implementation-dependent memory allocation/de-allocation algorithms (or even bugs) cause a leak – this is beyond the scope of this annex. The second general case is when objects remain referenced after they are no longer needed. This is a logic error which requires the programmer to modify the code to delete references to objects when they are no longer required.

There is a third very subtle memory leak case wherein objects mutually reference one another without any outside references remaining – a kind of deadly embrace where one object references a second object (or group of objects) so the second object(s) can't be collected but the second object(s) also reference the first one(s) so it/they too can't be collected. This group is known as cyclic garbage. Python provides a garbage collection module called `gc` which has functions which enable the programmer to enable and disable cyclic garbage collection as well as inspect the state of objects tracked by the cyclic garbage collector so that these, often very subtle leaks, can be traced and eliminated.

E.41.2 Guidance to language users

- Release all objects when they are no longer required.

E.42 Templates and Generics [SYM]

This vulnerability is not applicable to Python because Python does not implement these mechanisms.

E.43 Inheritance [RIP]

E.43.1 Applicability to language

Python supports inheritance through a hierarchical search of namespaces starting at the subclass and proceeding upward through the superclasses. Multiple inheritance is also supported. Any inherited methods are subject to the same vulnerabilities that occur whenever using code that is not well understood.

E.43.2 Guidance to language users

- Inherit only from trusted classes; and
- Use Python's built-in documentation (such as docstrings) to obtain information about a class' method before inheriting from it.

E.44 Extra Intrinsics [LRM]

E.44.1 Applicability to language

Python provides a set of built-in intrinsics which are implicitly imported into all Python scripts. Any of the built-in variables and functions can therefore easily be overridden:

```
x = 'abc'
print(len(x))#=> 3
def len(x):
    return 10
print(len(x))#=> 10
```

If the example above the built-in `len` function is overridden with logic that always returns `10`. Note that the `def` statement is executed dynamically so the new overriding `len` function has not yet been defined when the first call to `len` is made therefore the built-in version of `len` is called in line 2 and it returns the expected result (`3` in

this case). After the new `len` function is defined it overrides all references to the builtin-in `len` function in the script. This can later be "undone" by explicitly importing the built-in `len` function with the following code:

```
from builtins import len
print(len(x))#=> 3
```

It's very important to be aware of name resolution rules when overriding built-ins (or anything else for that matter). In the example below, the overriding `len` function is defined within another function and therefore is not found using the LEGB rule for name resolution (see E.23):

```
x = 'abc'
print(len(x))#=> 3
def f(x):
    def len(x):
        return 10
print(len(x))#=> 3
```

E.44.2 Guidance to language users

- Do not override built-in "intrinsics" unless absolutely necessary

E.45 Argument Passing to Library Functions [TRJ]

E.45.1 Applicability to language

Refer to E.35 Subprogram Signature Mismatch [OTR].

E.45.2 Guidance to language users

Refer to E.36 Subprogram Signature Mismatch [OTR].

E.46 Inter-language Calling [DJS]

E.46.1 Applicability to language

Python has a documented API for extending Python using libraries coded in C or C++. The library(s) are then imported into a Python module and used in the same manner as a module written in Python. Python's standard for interfacing to the "C" language is documented in http://docs.python.org/py3k/c-api/.

Conversely, code written in C or C++ can embed Python. The standard for embedding Python is documented in: http://docs.python.org/py3k/extending/embedding.html.

The Jython system is a Java-based implementation that interfaces with Java and IronPython provides interfaces to Microsoft .NET languages.

E.46.2 Guidance to language users

- Use the language interface APIs documented on the Python web site for interfacing to C/C++, the Jython

web site for Java, the IronPython web site for .NET languages, and for all other languages consider creating intermediary C or C++ modules to call functions in the other languages since many languages have documented API's to C and C++.

E.47 Dynamically-linked Code and Self-modifying Code [NYY]

E.47.1 Applicability to language

Python supports dynamic linking by design. The `import` statement fetches a file (known as a module in Python), compiles it and executes the resultant byte code at run time. This is the normal way in which external logic is made accessible to a Python program therefore Python is inherently exposed to any vulnerabilities that cause a different file to be imported:

- Alteration of a file directory path variable to cause the file search locate a different file first; and
- Overlaying of a file with an alternate.

Python also provides an `eval` and an `exec` statement each of which can be used to create self-modifying code:

```
x = "print('Hello " + "World')"
eval(x)#=> Hello World
```

Guerrilla patching, also known as monkey patching, is a way to dynamically modify a module or class at run-time to extend, or subvert their processing logic and/or attributes. It can be a dangerous practice because once "patched" any other modules or classes that use the modified class or module may unwittingly be using code that does not do what they expect which could cause unexpected results.

E.47.2 Guidance to language users

- Avoid using `exec` or `eval` and *never* use these with untrusted code;
- Be careful when using Guerrilla patching to ensure that all users of the patched classes and/or modules continue to function as expected; conversely, be aware of any code that patches classes and/or modules that your code is using to avoid unexpected results; and
- Ensure that the file path and files being imported are from trusted sources.

E.48 Library Signature [NSQ]

E.48.1 Applicability to language

Python has an extensive API for extending or embedding Python using modules written in C, Java, and Fortran. Extensions themselves have the potential for vulnerabilities exposed by the language used to code the extension which is beyond the scope of this annex.

Python does not have a library signature checking mechanism but its API provides functions and classes to help ensure that the signature of the extension matches the expected call arguments and types. See E.36.

E.48.2 Guidance to language users

- Use only trusted modules as extensions; and
- If coding an extension utilize Python's extension API to ensure a correct signature match.

E.49 Unanticipated Exceptions from Library Routines [HJW]

E.49.1 Applicability to language

Python is often extended by importing modules coded in Python and other languages. For modules coded in Python the risks include:

- Interception of an exception that was intended for a module's imported exception handling code (and vice versa); and
- Unintended results due to namespace collisions (covered in E.22 and elsewhere in this annex).

For modules coded in other languages the risks include:

- Unexpected termination of the program; and
- Unexpected side effects on the operating environment.

E.49.2 Guidance to language users

- Wrap calls to library routines and use exception handling logic to intercept and handle exceptions when practicable.

E.50 Pre-processor Directives [NMP]

This vulnerability is not applicable to Python because Python has no pre-processor directives.

E.51 Suppression of Language-defined Run-time Checking [MXB]

This vulnerability is not applicable to Python because Python does not have a mechanism for suppressing run-time error checking. The only suppression available is the suppression of run-time warnings using the command line –W option which suppresses the printing of warnings but does not affect the execution of the program.

E.52 Provision of Inherently Unsafe Operations [SKL]

E.52.1 Applicability to language

Python has very few operations that are inherently unsafe. For example, there is no way to suppress error checking or bounds checking. However there are two operations provided in Python that are inherently unsafe in any language:

- Interfaces to modules coded in other languages since they could easily violate the security of the calling of embedded Python code; and
- Use of the `exec` and `eval` dynamic execution functions (see E.47).

E.52.2 Guidance to language users

- Use only trusted modules; and
- Avoid the use of the `exec` and `eval` functions.

E.53 Obscure Language Features [BRS]

E.53.1 Applicability of language

Python has some obscure language features as described below:

Functions are defined when executed:

```
a = 1
while a < 3:
    if a == 1:
        def f():
            print("a must equal 1")
    else:
        def f():
            print("a must not equal 1")
    f()
    a += 1
```

The function `f` is defined and redefined to result in the output below:

```
a must equal 1
a must not equal 1
```

A function's variables are determined to be local or global using static analysis: if a function only references a variable and never assigns a value to it then it is assumed to be global otherwise it is assumed to be local and is added to the function's namespace. This is covered in some detail in E.23.

A function's default arguments are assigned when a function is *defined*, not when it is *executed*:

```
def f(a=1, b=[]):
    print(a, b)
    a += 1
    b.append("x")
f()
f()
f()
```

The output from above is typically expected to be:

```
1 []
1 []
1 []
```

But instead it prints:

```
1 []
1 ['x']
1 ['x', 'x']
```

This is because neither `a` nor `b` are reassigned when `f` is *called* with *no* arguments because they were assigned values when the function was *defined*. The local variable `a` references an immutable object (an integer) so a new object is created when the `a += 1` statement is created and the default value for the `a` argument remains unchanged. The mutable list object `b` is updated in place and thus "grows" with each new call.

The += Operator does not work as might be expected for mutable objects:

```
x = 1
x += 1
print(x) #=> 2 (Works as expected)
```

But when we perform this with a mutable object:

```
x = [1, 2, 3]
y = x
print(id(x), id(y))#=> 38879880 38879880
x += [4]
print(id(x), id(y))#=> 38879880 38879880
x = x + [5]
print(id(x), id(y))#=> 48683400 38879880
print(x,y)#=> [1, 2, 3, 4, 5] [1, 2, 3, 4]
```

The += operator changes `x` in place while the `x = x + [5]` creates a new list object which, as the example above shows, is not the same list object that `y` still references. This is Python's normal handling for all assignments (immutable or mutable) – create a new object and assign to it the value created by evaluating the expression on the right hand side (RHS):

```
x = 1
print(id(x)) #=> 506081728
x = x + 1
print(id(x)) #=> 506081760
```

Equality (or equivalence) refers to two or more objects having the same value. It is tested using the == operator which can thought of as the 'is equal to test'. On the other hand, two or more *names* in Python are considered identical only if they reference the same object (in which case they would, of course, be equivalent too). For example:

```
a = [0,1]
b = a
c = [0,1]
a is b, b is c, a == c #=> (True, False, True)
```

`a` and `b` are both names that reference the same objects while `c` references a different object which has the same *value* as both `a` and `b`.

Python provides built-in classes for persisting objects to external storage for retrieval later. The complete object, *including its methods*, is serialized to a file (or DBMS) and re-instantiated at a later time by any program which has access to that file/DBMS. This has the potential for introducing rogue logic in the form of object methods within a substituted file or DBMS.

Python supports passing parameters by keyword as in:

```
a = myfunc(x = 1, y = "abc")
```

This can make the code more readable and allows one to skip parameters. It can also reduce errors caused by confusing the order of parameters.

E.53.2 Guidance to language users

Ensure that a function is defined before attempting to call it; Be aware that a function is defined dynamically so its composition and operation may vary due to variations in the flow of control within the defining program;

- Be aware of when a variable is local versus global;
- Do not use mutable objects as default values for arguments in a function definition unless you absolutely need to and you understand the effect;
- Be aware that when using the += operator on mutable objects the operation is done in place;
- Be cognizant that assignments to objects, mutable and immutable, always create a new object;
- Understand the difference between equivalence and equality and code accordingly; and
- Ensure that the file path used to locate a persisted file or DBMS is correct and *never* ingest objects from an untrusted source.

E.54 Unspecified Behaviour [BQF]

E.54.1 Applicability of language

Understanding how Python manages identities becomes less clear when a script is run using integers (or short strings):

```
a=1
b=a
c=1
a is b, b is c, a == c #=> (True, True, True)
```

In the example above `c` references the same object as `a` and `b` even though `c` was never assigned to either `a` or `b`. This is a nuance of how Python is optimized to cache short strings and small integers. Other than in a test for identity as above, this nuance has no effect on the logic of the program (e.g., changing the value of `c` to 2 will not affect `a` or `b`). Refer also to E.2.2 Key Concepts.

When persisting objects using pickling, if an exception is raised then an unspecified number of bytes may have already been written to the file.

E.54.2 Guidance to language users

- Do not rely on the content of error messages – use exception objects instead;
- When persisting object using pickling use exception handling to cleanup partially written files; and
- Do not depend on the way Python may or may not optimize object references for small integer and string objects because it may vary for environments or even for releases in the same environment.

E.55 Undefined Behaviour [EWF]

E.55.1 Applicability to language

Python has undefined behaviour in the following instances:

- Caching of immutable objects can result in (or not result in) a single object being referenced by two or more variables. Comparing the variables for equivalence (i.e., `if a == b`) will always yield a `True` but checking for equality (using the `is` built-in) may, or may not, dependent on the implementation:
```
a = 1
b = 2-1
print(a == b, a is b) #=> (True, ?)
```
- The sequence of keys in a dictionary is undefined because the hashing function used to index the keys is unspecified therefore different implementations are likely to yield different sequences.
- The `Future` class encapsulates the asynchronous execution of a callable. The behaviour is undefined if the `add_done_callback(fn)` method (which attaches the callable `fn` to the future) raises a `BaseException` subclass.
- Modifying the dictionary returned by the `vars` built-in has undefined effects when used to retrieve the dictionary (i.e., the namespace) for an object.
- Form feed characters used for indentation have an undefined effect on the character count used to determine the scope of a block.
- The `catch_warnings` function in the context manager can be used to temporarily suppress warning messages but it can only be guaranteed in a single-threaded application otherwise, a when 2 or more threads are active, the behaviour is undefined.
- When sorting a list using the `sort()` method, attempting to inspect or mutate the content of the list will result in undefined behaviour.
- The order of sort of a list of sets, using `list.sort()`, is undefined as is the use of the function used on a list of sets that depend on total ordering such as `min()`, `max()`, and `sorted()`.
- Undefined behaviour will occur if a thread exits before the main procedure from which it was called itself exits.

E.55.2 Guidance to language users

- Understand the difference between testing for equivalence (e.g. ==) and equality (e.g., `is`) and never depend on object identity tests to pass or fail when the variables reference immutable objects;

- Do not depend on the sequence of keys in a dictionary to be consistent across implementations.
- When launching parallel tasks don't raise a `BaseException` subclass in a callable in the `Future` class;
- Never modify the dictionary object returned by a `vars` call;
- Never use form feed characters for indentation;
- Consider using the `id` function to test for object equality;
- Do not try to use the `catch_warnings` function to suppress warning messages when using more than one thread; and
- Never inspect or change the content of a list when sorting a list using the `sort()` method.

E.56 Implementation–defined Behaviour [FAB]

E.56.1 Applicability to language

Python has implementation-defined behaviour in the following instances:

- Mixing tabs and spaces to indent is defined differently for UNIX and non-UNIX platforms;
- Byte order (little endian or big endian) varies by platform;
- Exit return codes are handled differently by different operating systems;
- The characteristics, such as the maximum number of decimal digits that can be represented, vary by platform;
- The filename encoding used to translate Unicode names into the platform's filenames varies by platform; and
- Python supports integers whose size is limited only by the memory available. Extensive arithmetic using integers larger than the largest integer supported in the language used to implement Python will degrade performance so it may be useful to know the integer size of the implementation.

E.56.2 Guidance to language users

- Always use either spaces or tabs (but not both) for indentations;
- Consider using the -tt command line option to raise an IndentationError;
- Consider using a text editor to find and make consistent, the use of tabs and spaces for indentation;
- Either avoid logic that depends on byte order or use the `sys.byteorder` variable and write the logic to account for byte order dependent on its value ('`little`' or '`big`').
- Use zero (the default exit code for Python) for successful execution and consider adding logic to vary the exit code according to the platform as obtained from `sys.platform` (e.g., '`win32`', '`darwin`' etc.).
- Interrogate the `sys.float.info` system variable to obtain platform specific attributes and code according to those constraints.
- Call the `sys.getfilesystemcoding()` function to return the name of the encoding system used.
- When high performance is dependent on knowing the range of integer numbers that can be used without degrading performance use the `sys.int_info struct` sequence to obtain the number of bits per digit (`bits_per_digit`) and the number of bytes used to represent a digit (`sizeof_digit`).

E.57 Deprecated Language Features [MEM]

E.57.1 Applicability to language

The following features were deprecated in the latest (as of this writing) version of E 3.1. These are documented at http://docs.python.org/release/3.1.3/whatsnew/3.1.html:

- The string.maketrans() function is deprecated and is replaced by new static methods, bytes.maketrans() and bytearray.maketrans(). This change solves the confusion around which types were supported by the string module. Now, str, bytes, and bytearray each have their own maketrans and translate methods with intermediate translation tables of the appropriate type.
- The syntax of the with statement now allows multiple context managers in a single statement:

```
with open('mylog.txt') as infile, open('a.out', 'w') as outfile:
    for line in infile:
         if '<critical>' in line:
             outfile.write(line)
```

- With the new syntax, the `contextlib.nested()` function is no longer needed and is now deprecated.
- Deprecated `PyNumber_Int()`. Use `PyNumber_Long()` instead.
- Added a new `PyOS_string_to_double()` function to replace the deprecated functions `PyOS_ascii_strtod()` and `PyOS_ascii_atof()`.
- Added `PyCapsule` as a replacement for the `PyCObject` API. The principal difference is that the new type has a well defined interface for passing typing safety information and a less complicated signature for calling a destructor. The old type had a problematic API and is now deprecated.

E.57.2 Guidance to language users

- When practicable, migrate Python programs to the current standard.

Annex F
(*informative*)
Vulnerability descriptions for the language Ruby

F.1 Identification of standards and associated documents

IPA Ruby Standardization WG Draft – August 25, 2010

F.2 General Terminology and Concepts

block: A procedure which is passed to a method invocation.

class: An object which defines the behaviour of a set of other objects called its instances.

class variable: A variable whose value is shared by all the instances of a class.

constant: A variable which is defined in a class or a module and is accessible both inside and outside the class or module. The value of a constant is ordinarily expected to remain unchanged during the execution of a program, but IPA Ruby Standardization Draft does not force it.

exception: An object which represents an exceptional event.

global variable: A variable which is accessible everywhere in a program.

implementation-defined: Possibly differing between implementations, but defined for every implementation.

instance method: A method which can be invoked on all the instances of a class.

instance variable: A variable that exists in a set of variable bindings which every object has.

local variable: A variable which is accessible only in a certain scope introduced by a program construct such as a method definition, a block, a class definition, a module definition, a singleton class definition, or the top level of a program.

method: A procedure which, when invoked on an object, performs a set of computations on the object.

method visibility: An attribute of a method which determines the conditions under which a method invocation is allowed.

module: An object which provides features to be included into a class or another module.

object: A computational entity which has states and behaviour. The behaviour of an object is a set of methods which can be invoked on the object.

singleton class: An object which can modify the behaviour of its associated object.

singleton method: An instance method of a singleton class.

unspecified behaviour: Possibly differing between implementations, and not necessarily defined for any particular implementation.

variable: A computational entity that refers to an object, which is called the value of the variable.

variable binding: An association between a variable and an object which is referred to by the variable.

F.3 Type System [IHN]

F.3.1 Applicability to language

Ruby employs a dynamic type system usually referred to as "duck typing". In this system the class or type of an object is less important than the interface, or methods, it defines. Two different classes may respond to the same methods, which mean instances of each class will handle the same method call. Usually an object is not implicitly changed into another type.

Automatic conversion occurs for some built-in types in certain situations. For example with the addition of a float and an integer, the integer will be converted automatically to a float. In the examples below, the result of an operation is indicated by a Ruby comment starting with `=>`.

```
a = 2
b = 2.0
a + b #=> 4.0
```

Another instance of automatic conversion is when an integer becomes too large to fit within a machine word. On a 32-bit machine Ruby `Fixnums` have the range -2^{30} to $2^{30}-1$. When an integer becomes such that it no longer fits within said range it is converted to a `Bignum`. `Bignums` are arbitrary length integers bounded only by memory limitations.

Explicit conversion methods exist in Ruby to convert between types. The integer class contains the methods `to_s` and `to_f` which return the integer represented as a `string` object and `float` object, respectively.

```
10.to_s   #=> "10"
10.to_f   #=>  10.0
```

Strings likewise support conversion to integer and float objects.

```
"5".to_i  #=> 5
"5".to_f  #=> 5.0
```

Duck typing grants programmers of Ruby great flexibility. Strict typing is not imposed by the language, but if a programmer chooses, he or she can write programs such that methods mandate the class of the objects on which they operate. This is discouraged in Ruby. If an object is called with a method it does not know, an exception will be raised.

F.3.2 Guidance to language users

- Knowledge of the types or objects used is a must. Compatible types are ones which can be intermingled and convert automatically when necessary. Incompatible types must be converted to a compatible type before use.
- Do not check for specific classes of objects unless there is good justification.
- Provide code to catch exceptions resulting from mismatches between objects and methods.

F.4 Bit Representations [STR]

F.4.1 Applicability to language

Ruby abstracts internal storage of integers. Users do not need to concern themselves about the size (in bits) of an integer. Since integers grow as needed the user does not need to worry about overflow. Ruby provides a mechanism to inspect specific bits of an integer through the `[]` method. For example to read the 10th bit of a number:

```
number = 42
number[10]   #=> 0
number = 1024
number[10]   #=> 1
```

Note that the bits returned are not required to correspond to the internal representation of the number, just that it returns a consistent representation of the number in that implementation.

Ruby supports a variety of bitwise operators. These include ~ (not), `&` (and), `|` (or), ^ (exclusive or), << (shift left), and >> (shift right). Each of these operators works with integers of any size.

Ruby offers a pack method for the Array class (`Array#pack`) which produces a binary sequence dictated by the user supplied template. In this way members of an array can be converted to different bit representations. For instance an option for numbers is to store them in one of three ways: native-endian, big-endian, and little-endian. In this way bit sequences can be constructed for a particular interaction or purpose. There is a similar unpack method which will extract data given a template and bit sequence.

F.4.2 Guidance to language users

- For values created within Ruby the user need not concern themselves with the internal representation of data. In most situations using specific binary representations makes code harder to read and understand.
- Network packets that go on the wire are one case where bit representation is important. In situations like this be sure to use the `Array#pack` to produce network endian data[17].
- Binary files are another situation where bit representation matters. The file format description should indicate big-endian or little-endian preference.

[17] Network APIs use big-endian data.

F.5 Floating-point Arithmetic [PLF]

F.5.1 Applicability to language

Ruby supports the use of floating-point arithmetic with the Float class. The precision of floats in Ruby is implementation defined, however if the underlying system supports IEC 60559, the representation of floats shall be the 64-bit double format as specified in IEC 60559, 3.2.2.

Floating-point numbers are usually approximations of real numbers and as such some precision is lost. This is problematic when performing repeated operations. For example adding small values to numbers sometimes results in accumulation errors. Testing numbers for equality is sometimes unreliable as well. For this reason floating-point numbers should not be used to terminate loops.

F.5.2 Guidance to language users

- Guidance in clause 6.5 applies here.

F.6 Enumerator Issues [CCB]

F.6.1 Applicability to language

Ruby does not provide enumerations. Instead provides a facility for named symbols. These symbols are unique representations with no value associated. In Ruby, symbols are lightweight objects which need not be defined ahead of time. For example,

```
travel(:north)
```

is a valid use of the symbol `:north`. (Ruby's literal syntax for symbols is a colon followed by a word.) There is no danger of accidentally getting to the "value" of an enumeration. So this:

```
travel(:north + :south)
```

is not allowed. Symbols do not support addition, or any method which alters the symbol.

Sometimes it is helpful to have values associated with enumerations. In Ruby this can be accomplished by using a hash. For example,

```
traffic_light = {
   :green => "go"
   :yellow => "caution"
   :red => "stop"}
traffic_light[:yellow]
```

In this way values can be associated with the symbols. Members of a hash are accessed using the same bracket syntax as members of arrays. Note only integers can be used in array indexing, thus non-standard use of a symbol as an array index will raise an exception.

F.6.2 Guidance to language users

- Use symbols for enumerators rather than named constants.
- Do not define named constants to represent enumerators.

F.7 Numeric Conversion Errors [FLC]

F.7.1 Applicability to language

Integers in the Ruby language are of unbounded length (the actual limit is dependent on the machine's memory). When an integer exceeds the word size for the machine there is no rollover and no errors occur. Instead Ruby converts the integer from one type to another. When possible, integers in Ruby are stored in a `Fixnum` object. `Fixnum` is a class which has limited integer range, yet is able to store the number efficiently in one machine word. Typically on a 32-bit machine the range is usually -2^{30} to $2^{30}-1$. These ranges are implementation defined.

Once calculations exceed this range, integers are stored in a Bignum object. Bignum class allows any length (memory providing) integer. This all takes place without the user's explicit instruction. The result of any Bignum calculation may be returned as an integer if the value can be represented as an integer.

Ruby converts integers to floating point with the user's explicit intent. Loss of precision can occur converting from a large magnitude integer to a floating point number. This does not generate an error.

F.7.2 Guidance to language users

- Be aware that use of Bignums can have performance and storage implications.

F.8 String Termination [CJM]

This vulnerability is not applicable to Ruby since strings are not terminated by a special character.

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

This vulnerability is not applicable to Ruby since array indexing is checked.

F.10 Unchecked Array Indexing [XYZ]

This vulnerability is not applicable to Ruby since array indexing is checked.

F.11 Unchecked Array Copying [XYW]

This vulnerability is not applicable to Ruby since arrays grow.

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

This vulnerability is not applicable to Ruby since users cannot manipulate pointers.

F.13 Pointer Arithmetic [RVG]

This vulnerability is not applicable to Ruby since users cannot manipulate pointers.

F.14 Null Pointer Dereference [XYH]

This vulnerability is not applicable to Ruby since users cannot create or dereference null pointers.

F.15 Dangling Reference to Heap [XYK]

This vulnerability is not applicable to Ruby since users cannot explicitly allocate and explicitly deallocate memory.

F.16 Arithmetic Wrap-around Error [FIF]

This vulnerability is not applicable to Ruby since integers are unbounded.

F.17 Using Shift Operations for Multiplication and Division [PIK]

This vulnerability is not applicable to Ruby since logic shifts on integers will not modify the sign bit or lose significant bits if the size of the value grows.

F.18 Sign Extension Error [XZI]

This vulnerability is not applicable to Ruby since users cannot explicitly convert a signed integer to a larger integer without modifying the value.

F.19 Choice of Clear Names [NAI]

F.19.1 Applicability to language

Ruby is susceptible to errors resulting from similar looking names. Ruby provides scoping of local variables. However, this can be confusing. Local variables cannot be accessed from another method, but local variables can be accessed from a block. Ruby features variable prefixes for non-local variables. The dollar sign signifies a global variable. A single "`@`" symbol signifies a variable scoped to the current object. A double at symbol signifies a class wide variable, accessible from any instance of said class.

F.19.2 Guidance to language users

- Use names that are clear and visually unambiguous.
- Be consistent in choosing names.

F.20 Dead Store [WXQ]

F.20.1 Applicability to language

Ruby is susceptible to errors of accidental assignments resulting from typos of variable names. Since variables do not need to declared before use such an assignment may go unnoticed. Such behaviour is indicative of programmer error.

F.20.2 Guidance to language users

- Check that each assignment is made to the intended variable identifier.
- Use static analysis tools, as they become available, to mechanically identify dead stores in the program.

F.21 Unused Variable [YZS]

F.21.1 Applicability to language

Ruby is susceptible to this vulnerability. Ruby does not permit the declaration of variables, but "declares" parameters, which might never be read or written, hence providing storage space useful to an attacker.

F.21.2 Guidance to language users

- Enable detection of unused variables in the processor.

F.22 Identifier Name Reuse [YOW]

F.22.1 Applicability to language

Ruby employs various levels of scope which allow users to name variables in different scopes with the same name. This can cause confusion in situations where the user is unaware of the scoping rules, especially in the use of blocks.

Modules provide a way to group methods and variables without the need for a class. To use these module and method names must be completely specified. For example:

```
Base64::encode(text)
```

However modules can be included, thus putting the contents of the module within the current scope. So:

```
include Base64
encode(text)
```

can cause clashes with names already in scope. When this occurs the current scope takes precedence, but the user may not realize this resulting in unknown errors.

F.22.2 Guidance to language users

- Ensure that a definition does not occur in a scope where a different definition is accessible.

- Know what a module defines before including. If any definitions conflict, do not include the module, instead use the fully qualified name to refer to any definitions in the module.

F.23 Namespace Issues [BJL]

F.23.1 Applicability to language

This is indeed an issue for Ruby. The interpreter will resolve names to the most recent definition as the one to use, possibly redefining a variable. Scoping provides some means of protection, but there are some cases where confusion arises. A method definition cannot access local variables defined outside of its scope, yet a block can access these variables. For example:

```
x = 50
def power(y)
   puts x**y
end
power(2) #=> NameError: undefined local variable or method 'x'
```

But the following can access the x variable as defined:

```
x = 50
def execute_block(y)
   yield y
end
execute_block(2) {|y| x**y} #=> 2500
```

F.23.2 Guidance to language users

- Avoid unnecessary includes.
- Do not access variables outside of a block without justification.

F.24 Initialization of Variables [LAV]

This vulnerability is not applicable to Ruby since variables cannot be read before they are assigned.

F.25 Operator Precedence/Order of Evaluation [JCW]

F.25.1 Applicability to language

Ruby provides a rich set of operators containing over fifty operators and twenty levels of precedence. Confusion arises especially with operators which mean something similar, but are for different purposes. For example,

```
x = flag_a or flag_b
```

The Ruby language understands this as equivalent to:

```
(x = flag_a) or flag_b
```

The above assigns the value of `flag_a` to `x`. If `flag_a` evaluates to false, then the value of the entire expression is `flag_b`. The intent of the programmer was most likely assign true to x if either `flag_a` or `flag_b` are true:

```
x = flag_a || flag_b
```

F.25.2 Guidance to language users

- Use parenthesis around operators which are known to cause confusion and errors.
- Break complex expressions into simpler ones, storing sub-expressions in variables as needed.

F.26 Side-effects and Order of Evaluation [SAM]

F.26.1 Applicability to language

In Ruby method invocations can change the state of the receiver (object whose method is invoked). This occurs not just for input and output for which side-effects are unavoidable, but also for routine operations such as mutating strings, modifying arrays, or defining methods. Ruby has adopted a naming convention which indicates destructive methods (those which modify the receiver) instead of creating a new object which is a modified copy. For example,

```
array = [1, 2, 3]  #=> [1, 2, 3]
array.slice(1..2)  #=> [2, 3]
array              #=> [1, 2, 3]
array.slice!(1..2)    #=> [2, 3]
array              #=> [1]
```

The method name with the exclamation signifies the object itself will be modified, whereas the other method does not modify it. Sometimes though the method is understood by the user to modify the object or cause side-effects. For example,

```
array = [1, 2, 3]
array.concat([4, 5, 6])
array #=> [1, 2, 3, 4, 5, 6]
```

These behaviours are documented and with little effort the user will be able recognize which methods cause side-effects and what those effects are.

The order of evaluation in Ruby is left to right. Order of evaluation and order of precedence are different. Precedence allows the familiar order of operations for expressions. For example,

```
a + b * c
```

`a` is evaluated, followed by `b` and `c`, then the value of `b` and the value of `c` are multiplied and added to the value of `a`. This is a subtle point which matters only if `a`, `b`, or `c` cause side effects. The following illustrates this:

```
def a; print "A"; 1; end
def b; print "B"; 2; end
```

```
def c; print "C"; 3; end
a + b * c #=> 7, and "ABC" is printed to standard output
```

F.26.2 Guidance to language users

- Read method documentation to be aware of side-effects.
- Do not depend on side-effects of a term in the expression itself.

F.27 Likely Incorrect Expression [KOA]

F.27.1 Applicability to language

Ruby has operators which are typographically similar, yet which have different meanings. The assignment operator and comparison operators are examples of these. Both are expressions and can be used in conditional expressions.

```
if a = 3 then #…
if a == 3 then #…
```

The first example assigns the value 3 to the variable a. 3 evaluates to true and the conditional is executed. The second checks that the variable `a` is equal to the value 3 and executes the conditional if true.

Another instance is the use of assignments in Boolean expressions. For instance,

```
a = x or b = y
```

This expression assigns the value `x` to `a`. If `x` is false then the value of y will be assigned to `b`. This should be avoided as the second assignment will not always occur. This could possibly be the intention of the programmer, but a more clear way to write the code which accomplishes that is:

```
a = x
b = y if a
```

There is no confusion here as the second assignment clearly has an if-modifier. This is common and well understood in the Ruby language.

F.27.2 Guidance to language users

- Avoid assignments in conditions.
- Do not perform assignments within Boolean expressions.

F.28 Dead and Deactivated Code [XYQ]

F.28.1 Applicability to language

Dead and deactivated, as in any programming language with code branching, can be a problem in Ruby. The existence of code which can never be reached is not a problem itself. Its existence indicates the possibility of a coding error. Code coverage tools can help analyse which portions of code can and cannot be reached.

In particular the developer should ensure each branch can evaluate to true or false. If a condition only ever evaluates to true, then only one branch will be taken. This situation creates dead code.

F.28.2 Guidance to language users

- Use analysis tools to identify unreachable code.

F.29 Switch Statements and Static Analysis [CLL]

F.29.1 Applicability to language

Ruby provides a case statement. This construct is similar to C's switch statement with a few important differences. Cases do not "flow through" from one to the next. Only one case will be executed. An else case can be provided, but is not required. If no cases match then the value of the case statement is nil.

F.29.2 Guidance to language users

- Include an else clause, unless the intention of cases not covered is to return the value nil.
- Multiple expressions (separated by commas) may be served by the same when clause.

F.30 Demarcation of Control Flow [EOJ]

This vulnerability is not applicable to Ruby since control constructs require an explicit termination symbol.

F.31 Loop Control Variables [TEX]

F.31.1 Applicability to language

Ruby allows the modification of loop control variables from within the body of the loop. This is usually not performed, as the exact results are not always clear.

F.31.2 Guidance to language users

- Do not modify loop control variables inside the loop body

F.32 Off-by-one Error [XZH]

F.32.1 Applicability to language

Like any programming language which supplies equality operators and array indexing, Ruby is vulnerable to off-by-one-errors. These errors occur when the developer creates an incorrect test for a number range or does not index arrays starting at zero.

Some looping constructs of the language alleviate the problem, but not all of them. For example this code

```
for i in 1..5
   print i
end#=> 12345
```

In addition to this is the usual confusion associated between <, <=, >, and >= in a test

Also unique to Ruby is the confusion of these particular loop constructs:

```
5.times {|x| p x}
```

and

```
1.upto(5) {|x| p x}
```

Each loop executes the code block five times. However the values passed to the block differ. With `5.times` the loop starts with the value `0` and the last value passed to the block is `4`. However in the case of `1.upto(5)`, it starts by passing `1`, and ends by passing `5`.

F.32.2 Guidance to language users

- Use careful programming practice when programming border cases.
- Use static analysis tools to detect off-by-one errors as they become available.
- Instead of writing a loop to iterate all the elements of a container use the `each` method supplied by the object's class.

F.33 Structured Programming [EWD]

F.33.1 Applicability to language

Ruby makes structured programming easy for the user. Its object-oriented nature encourages at least a minimum amount of structure. However, it is still possible to write unstructured code. One feature which allows this is the break statement. The statement ends the execution of the current innermost loop. Excessive use of this may be confusing to others as it is not standard practice.

F.33.2 Guidance to language users

While there are some cases where it might be necessary to use relatively unstructured programming methods, they should generally be avoided. The following ways help avoid the above named failures of structured programming:

- Instead of using multiple return statements, have a single return statement which returns a variable that has been assigned the desired return value.
- In most cases a break statement can be avoided by using another looping construct. These are abundant in Ruby.
- Use classes and modules to partition functionality.

F.34 Passing Parameters and Return Values [CSJ]

F.34.1 Applicability to language

Ruby uses call by reference. Each variable is a named reference to an object. Return values in Ruby are merely the object of the last expression, or a return statement. Note that Ruby allows multiple return values by way of array. The following is valid:

```
return angle, velocity#=> [angle, velocity]
```

or less verbosely:

```
[angle, velocity]  #as the last line of the method
```

While pass by reference is a low over-head way of passing parameters, sometimes confusion can arise for programmers. If an object is modified by a method, then the possibility exists that the original object was modified. This may not the intended consequence. For example,

```
def pig_latin(word)
 word = word[1..-1] << word[0] if !word[/^[aeiouy]/]
 word << "ay"
end
```

The above method modifies the original object if it is that string starts with a vowel. The effect is the value outside the scope of the method is modified. The following revised method avoids this by calling the `dup` method on the object `word`:

```
def pig_latin_revised(word)
 word = word[/^[aeiouy]/] ? word.dup : word[1..-1] << word[0]
 word << "ay"
end
```

F.34.2 Guidance to language users

- Methods which modify their parameters should have the exclamation mark suffix. This is a standard Ruby idiom alerting users to the behaviour of the method.
- Make local copies of parameters inside methods if they are not intended to be modified.

F.35 Dangling References to Stack Frames [DCM]

This vulnerability is not applicable to Ruby since users cannot create dangling references.

F.36 Subprogram Signature Mismatch [OTR]

F.36.1 Applicability to language

Subprogram signatures in Ruby only consist of an arity count and name. A mismatch in the number of parameters will thus be caught before a call is executed. The type of each parameter is not enforced by the interpreter. This is

considered strength of Ruby, in that an object that responds to the same methods can imitate an object of another type. If an object does not respond to a method an error will be thrown. Also if the implementer chooses they can query the object to test its available methods and choose how to proceed.

F.36.2 Guidance to language users

- The Ruby interpreter will provide error messages for instances of methods called with an inappropriate number of arguments.

F.37 Recursion [GDL]

F.37.1 Applicability to language

Recursion can exhaust the finite stack space within a program. When this happens in Ruby, a "SystemStackError: stack level too deep" error occurs, which can be caught.

For methods which have the possibility of exhausting the stack, they should be implemented in an imperative style instead of the more mathematical, perhaps elegant, recursive manner.

There is no set amount of recursion an interpreter must support. Recursive methods which run successfully inside one conforming Ruby implementation may or may not successfully run inside a different implementation.

F.37.2 Guidance to language users

- When possible, design algorithms in an imperative manner.
- Test recursive methods extensively in the intended interpreter for stack overflow errors**.**

F.38 Ignored Error Status and Unhandled Exceptions [OYB]

F.38.1 Applicability to language

Ruby provides the class Exception which is used to communicate between raise methods (methods which throw an exception) and rescue statements. Exception objects carry information about the exception including its type, possibly a descriptive string, and optional trace back.

Given this information the programmer can deal with exception appropriately within rescue statements. In some cases this might be program termination, while in other cases an error may be par for the course.

F.38.2 Guidance to language users

- Extend Ruby's exception handling for your specific application.
- Use the language's built-in mechanisms (`rescue`, `retry`) for dealing with errors.

F.39 Termination Strategy [REU]

F.39.1 Applicability to language

Ruby standard does not explicitly state a termination strategy. The behaviour is unspecified. Differing implementations therefore can have different strategies.

F.39.2 Guidance to language users

- Consult implementation documentation concerning termination strategy.
- Do not assume each implementation behaves handles termination in the same manner.

F.40 Type-breaking Reinterpretation of Data [AMV]

This vulnerability is not applicable to Ruby since every data has a single interpretation.

F.41 Memory Leak [XYL]

This vulnerability is no applicable to Ruby since users cannot explicitly allocate memory.

F.42 Templates and Generics [SYM]

This vulnerability is not applicable to Ruby since it does not include templates or generics.

F.43 Inheritance [RIP]

F.43.1 Applicability to language

Ruby allows classes to inherit from one parent class. In addition to this modules can be included in a class. The class inherits the module's instance methods, class variables, and constants. Including modules can silently redefine methods or variables. Caution should be exercised when including modules for this reason. At most a class will have one direct superclass.

F.43.2 Guidance to language users

- Provide documentation of encapsulated data, and how each method affects that data.
- Inherit only from trusted sources, and, whenever possible check the version of the superclass during initialization.
- Provide a method that provides versioning information for each class.

F.44 Extra Intrinsics [LRM]

This vulnerability is not applicable to Ruby since the most recent definition of a method is selected for use.

F.45 Argument Passing to Library Functions [TRJ]

F.45.1 Applicability to language

The original Ruby interpreter is written in the C programming language. Because of this many libraries for Ruby have been written to interface with the Ruby and C. The library designer should make the library validate any input before its use.

F.45.2 Guidance to language users

- Develop wrappers around library functions that check the parameters before calling the function.
- Use only libraries known to have been consistent and validated interface requirements.

F.46 Inter-language Calling [DJS]

F.46.1 Applicability to language

Ruby is susceptible to this vulnerability when used in a multi-lingual environment.

There is not a standard definition for interactions with other languages. The Ruby language does not mandate calling or return conventions for example. Two conforming Ruby processors may differ enough that binary libraries designed for one may not work in the other.

F.46.2 Guidance to language users

- Implementations may provide a framework for inter-language calling. Be familiar with the data layout and calling mechanism of said framework.
- Use knowledge of all languages used to form names acceptable in all languages involved.
- Ensure the language in which error checking occurs is the one that handles the error.

F.47 Dynamically-linked Code and Self-modifying Code [NYY]

F.47.1 Applicability to language

Dynamically-linked code might be a different version at runtime than what was tested during development. This may lead to unpredictable results. Self-modifying code can be written in Ruby.

F.47.2 Guidance to language users

- Verify dynamically linked code being used is the same as that which was tested.
- Do not write self-modifying code.

F.48 Library Signature [NSQ]

F.48.1 Applicability to language

Ruby implementations which interface with libraries must have correct signatures for functions. Creating correct signatures for a large library is cumbersome and should be avoided by using tools.

F.48.2 Guidance to language users

- Use tools to create signatures.
- Avoid using libraries without proper signatures.

F.49 Unanticipated Exceptions from Library Routines [HJW]

F.49.1 Applicability to language

Ruby interfaces with libraries which could encounter unanticipated exceptions. In some situations, largely dependent on the interpreter implementation, exceptions can cause unpredictable and possibly fatal results.

F.49.2 Guidance to language users

- Use library routines which specify all possible exceptions.
- Use libraries which generate Ruby exceptions that can be `rescue`d.

F.50 Pre-processor Directives [NMP]

This vulnerability is not applicable to Ruby since it lacks a pre-processor.

F.51 Suppression of Language-defined Run-time Checking [MXB]

This vulnerability does not apply to Ruby since suppression of language defined run-time checks is not allowed. They are an integral part of the Ruby language.

F.52 Provision of Inherently Unsafe Operations [SKL]

This vulnerability does not apply to Ruby. It provides a means for "type-casting" which is safe by honoring classes with the same interface as the same type. If differences are exposed an exception will be raised by the processor. However, unlike statically typed languages, type safety in a Ruby program cannot be checked at compile-time. Therefore, maintenance of type safety in a Ruby program critically depends upon the correct coding of exception handlers to catch and treat type exceptions. Hence, Ruby programmers must pay particular attention to the class of vulnerabilities described in 6.38 and F.38 [OYB].

F.53 Obscure Language Features [BRS]

This vulnerability is not applicable to Ruby.

F.54 Unspecified Behaviour [BQF]

F.54.1 Applicability of language

Unspecified behaviour occurs where the proposed Ruby standard does not mandate a particular behaviour.

Unspecified behaviour in Ruby is abundant. In the proposed standard there are 136 instances of the phrase “unspecified behaviour.” Examples of

unspecified behaviour are:

- A `for`-expression terminated by a `break`-expression, `next`-expression, or `redo`-expression.
- Calling `Numeric#coerce(numeric)` with the value `NaN`.
- Calling `Integer#&(other)` if `other` is not an instance of the class `Integer`. This also applies to `Integer#|`, `Integer#^`, `Integer#<<`, and `Integer#>>`
- Calling `String#*(num)` if `other` is not an instance of the class `Integer`

F.54.2 Guidance to language users

- Do not rely on unspecified behaviour because the behaviour can change at each instance.
- Code that makes assumptions about the unspecified behaviour should be replaced to make it less reliant on a particular installation and more portable.
- Document instances of use of unspecified behaviour.

F.55 Undefined Behaviour [EWF]

F.55.1 Applicability to language

Undefined behaviour in Ruby is cover by sections [BQF] and [FAB].

F.55.2 Guidance to language users

- Avoid using features of the language which are not specified to an exact behaviour.

F.56 Implementation-defined Behaviour [FAB]

F.56.1 Applicability to language

The proposed Ruby standard defines implementation-defined behaviour as: possibly differing between implementations, but defined for every implementation.

The proposed Ruby standard has documented 98 instances of implementation defined behaviour. Examples of implementation defined behaviour are:

- Whether a singleton class can have class variables or not.
- The direct superclass of `Object`.
- The visibility of **`Module#class_variable_get`.**
- `Kernel.p(* args)` return value.

F.56.2 Guidance to language users

- The abundant nature of implementation-defined behaviour makes it difficult to avoid. As much as possible users should avoid implementation defined behaviour.
- Determine which implementation-defined implementations are shared between implementations. These are safer to use than behaviour which is different for every.

F.57 Deprecated Language Features [MEM]

This vulnerability is not applicable to Ruby since one edition of the standard exists.

Annex G
(*informative*)
Vulnerability descriptions for the language SPARK

G.1 Identification of standards and associated documentation

See C.1, plus the references below. In the body of this annex, the following documents are referenced using the short abbreviation that introduces each document, optionally followed by a specific section number. For example "[SLRM 5.2]" refers to section 5.2 of the SPARK Language Definition.

[SLRM] SPARK Language Definition: "SPARK95: The SPADE Ada Kernel (Including RavenSPARK)" Latest version always available from www.altran-praxis.com.

[SB] "High Integrity Software: The SPARK Approach to Safety and Security." John Barnes. Addison-Wesley, 2003. ISBN 0-321-13616-0.

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

[LSP] "A behavioral notion of subtyping." Barbara Liskov and Jeannette Wing. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16, Issue 6 (November 1994), pp. 1811 - 1841.

G.2 General terminology and concepts

The SPARK language is a contractualized subset of Ada, specifically designed for high-assurance systems. SPARK is designed to be amenable to various forms of static analysis that prevent or mitigate the vulnerabilities described in this TR.

Many terms and concepts applicable to Ada also apply to SPARK. See C.2.

This section introduces concepts and terminology which are specific to SPARK and/or relate to the use of static analysis tools.

Soundness

This concept relates to the absence of false-negative results from a static analysis tool. A false negative is when a tool is posed the question "Does this program exhibit vulnerability X?" but incorrectly responds "no." Such a tool is said to be **unsound** for vulnerability X. A sound tool effectively finds **all** the vulnerabilities of a particular class, whereas an unsound tool only finds some of them.

The provision of soundness in static analysis is problematic, mainly owing to the presence of unspecified and undefined features in programming languages. Claims of soundness made by tool vendors should be carefully evaluated to verify that they are reasonable for a particular language, compilers and target machines. Soundness claims are always underpinned by assumptions (for example, regarding the reliability of memory, the correctness of compiled code and so on) that should also be validated by users for their appropriateness.

Static analysis techniques can also be **sound in theory** – where the mathematical model for the language semantics and analysis techniques have been formally stated, proved, and reviewed – but **unsound in practice** owing to defects in the implementation of analysis tools. Again, users should seek evidence to support any soundness claim made by language designers and tool vendors. A language which is **unsound in theory** can never be sound in practice.

The single overriding design goal of SPARK is the provision of a static analysis framework which is **sound in theory,** and as **sound in practice** as is reasonably possible.

In the subsections below, we say that SPARK **prevents** a vulnerability if supported by a form of static analysis which is sound in theory. Otherwise, we say that SPARK **mitigates** a particular vulnerability.

SPARK Processor

We define a "SPARK Processor" to be a tool that implements the various forms of static analysis required by the SPARK language definition. Without a SPARK Processor, a program cannot reasonably be claimed to be SPARK at all, much in the same way as a compiler checks the static semantic rules of a standard programming language.

In SPARK, certain forms of analysis are said to be **mandatory** – they are required to be implemented and programs must pass these checks to be valid SPARK. Examples of mandatory analyses are the enforcement of the SPARK language subset, static semantic analysis (e.g. enhanced type checking) and information flow analysis [IFA].

Some analyses are said to be **optional** – a user may choose to enable these additional analyses at their discretion. The most notable example of an optional analysis in SPARK is the generation of verification conditions and their proof using a theorem proving tool. Optional analyses may provide greater depth of analysis, protection from additional vulnerabilities, and so on, at the cost of greater analysis time and effort.

Failure modes for static analysis

Unlike a language compiler, a user can always choose not to, or might just forget to run a static analysis tool. Therefore, there are two modes of failure that apply to all vulnerabilities:

1. The user fails to apply the appropriate static analysis tool to their code.
2. The user fails to review or mis-interprets the output of static analysis.

G.3 Type System [IHN]

SPARK mitigates this vulnerability.

SPARK's type system is a simplification of that of Ada. Both Explicit and Implicit conversions are permitted in SPARK, as is instantiation and use of Unchecked_Conversion [SB 1.3].

A design goal of SPARK is the provision of *static type safety,* meaning that programs can be shown to be free from all run-time type failures using entirely static analysis. If this optional analysis is achieved, a SPARK program should never raise an exception at run-time.

G.4 Bit Representation [STR]

SPARK mitigates this vulnerability.

SPARK is designed to offer a semantics which is independent of the underlying representation chosen by a compiler for a particular target machine. Representation clauses are permitted, but these do not affect the semantics as seen by a static analysis tool [SB 1.3].

G.5 Floating-point Arithmetic [PLF]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.5 [PLF].

G.6 Enumerator Issues [CCB]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.6 [CCB].

G.7 Numeric Conversion Errors [FLC]

SPARK prevents this vulnerability.

SPARK is designed to be amenable to static verification of the absence of predefined exceptions, and in particular all cases covered by this vulnerability [SB 11]. All numeric conversions (both explicit and implicit) give rise to a verification condition that must be discharged, typically using an automated theorem-prover.

G.8 String Termination [CJM]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.8 [CJM].

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

SPARK prevents this vulnerability.

SPARK is designed to permit static analysis for all such boundary violations, through techniques such as theorem proving or abstract interpretation [SB 11].

SPARK programs that have been subject to this level of analysis can be compiled with run-time checks suppressed, supported by a body of evidence that such checks could never fail, and thus removing the possibility of erroneous execution.

G.10 Unchecked Array Indexing [XYZ]

SPARK prevents this vulnerability. See G.9.

G.11 Unchecked Array Copying [XYW]

SPARK prevents this vulnerability.

Array assignments in SPARK are only permitted between objects that have statically matching bounds, so there is no possibility of an exception being raised [SB 5.5, SLRM 4.1.2]. Ada's "slicing" and "sliding" of arrays is not permitted in SPARK, so this vulnerability cannot occur.

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

SPARK prevents this vulnerability.

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access (pointer) types [SB 1.3, SLRM 3.10].

G.13 Pointer Arithmetic [RVG]

SPARK prevents this vulnerability. See G.12.

G.14 Null Pointer Dereference [XYH]

SPARK prevents this vulnerability. See G.12.

G.15 Dangling Reference to Heap [XYK]

SPARK prevents this vulnerability. See G.12.

G.16 Arithmetic Wrap-around Error [FIF]

See C.16 [FIF]. In addition, SPARK mitigates this vulnerability through static analysis to show that a signed integer expression can never overflow at run-time [SB 11].

G.17 Using Shift Operations for Multiplication and Division [PIK]

SPARK is identical to Ada with respect to this vulnerability and is mitigation. See C.17 [PIK].

G.18 Sign Extension Error [XZI]

SPARK is identical to Ada with respect to this vulnerability and is mitigation. See C.18 [XZI].

G.19 Choice of Clear Names [NAI]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.19 [NAI].

G.20 Dead store [WXQ]

SPARK prevents this vulnerability through mandatory static information flow analysis [IFA], which detects dead stores. Additionally, SPARK requires variables that are used for output to the environment, or for communication between tasks to be specifically identified. IFA for such variables is modified since it is know that consecutives writes to such variables might not constitute a dead store.

G.21 Unused Variable [YZS]

SPARK mitigates this vulnerability.

As in C.21 [YZS]. Also, SPARK is designed to permit sound static analysis of the following cases [IFA]:

- Variables which are declared but not used at all.
- Variables which are assigned to, but the resulting value is not used in any way that affects an output of the enclosing subprogram. This is called an “ineffective assignment” in SPARK.

G.22 Identifier Name Reuse [YOW]

SPARK prevents this vulnerability.

This vulnerability is prevented through language rules enforced by static analysis. SPARK does not permit names in local scopes to redeclare and hide names that are already visible in outer scopes [SLRM 6.1].

G.23 Namespace Issues [BJL]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.23 [BJL].

G.24 Initialization of Variables [LAV]

SPARK prevents this vulnerability through mandatory static information flow analysis [IFA].

G.25 Operator Precedence/Order of Evaluation [JCW]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.25 [JCW].

G.26 Side-effects and Order of Evaluation [SAM]

SPARK prevents this vulnerability.

SPARK does not permit functions to have side-effects, so all expressions are side-effect free. Static analysis of run-time errors also ensures that expressions evaluate without raising exceptions. Therefore, expressions are neutral to evaluation order and this vulnerability does not occur in SPARK [SLRM 6.1].

G.27 Likely Incorrect Expression [KOA]

SPARK is identical to Ada with respect to this vulnerability and its mitigation (see C.3.KOA) although many cases of “likely incorrect” expressions in Ada are forbidden in SPARK.

G.28 Dead and Deactivated Code [XYQ]

SPARK mitigates this vulnerability.

In addition to the advice of C.28 [XYQ], SPARK is amenable to optional static analysis of dead paths. A dead path cannot be executed in that the combination of conditions for its execution are logically equivalent to *false.* Such cases can be statically detected by theorem proving in SPARK.

G.29 Switch Statements and Static Analysis [CLL]

As in C.29 [CLL], this vulnerability is prevented by SPARK. The vulnerability relating to an uninitialized variable and the “when others” clause in a case statement is also prevented – see G.24 [LAV].

G.30 Demarcation of Control Flow [EOJ]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.30 [EOJ].

G.31 Loop Control Variables [TEX]

SPARK prevents this vulnerability in the same way as Ada. See C.31 [TEX].

G.32 Off-by-one Error [XZH]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.32 [XZH]. Additionally, any off-by-one error that gives rise to the potential for a buffer-overflow, range violation, or any other construct that could give rise to a predefined exception, will be detected by static analysis in SPARK [SB 11].

G.33 Structured Programming [EWD]

SPARK mitigates this vulnerability.

Several of the vulnerabilities in this category that affect Ada are entirely eliminated by SPARK. In particular: the use of the `goto` statement is prohibited in SPARK [SLRM 5.8], loop exit statements only apply to the most closely enclosing loop (so “multi-level loop exits” are not permitted) [SLRM 5.7], and all subprograms have a single entry and a single exit point [SLRM 6]. Finally, functions in SPARK must have exactly one return statement which must the final statement in the function body [SLRM 6].

G.34 Passing Parameters and Return Values [CSJ]

SPARK mitigates this vulnerability.

SPARK goes further than Ada with regard to this vulnerability. Specifically;

- SPARK forbids all aliasing of parameters and name [SLRM 6]
- SPARK is designed to offer consistent semantics regardless of the parameter passing mechanism employed by a particular compiler. Thus this implementation-dependent behaviour of Ada is eliminated from SPARK.

Both of these properties can be checked by static analysis.

G.35 Dangling References to Stack Frames [DCM]

SPARK prevents this vulnerability.

SPARK forbids the use of the ‘Address attribute to read the address of an object [SLRM 4.1]. The ‘Access attribute and all access types are also forbidden, so this vulnerability cannot occur.

G.36 Subprogram Signature Mismatch [OTR]

SPARK mitigates this vulnerability.

Default values for subprogram are not permitted in SPARK [SLRM 6], so this case cannot occur. SPARK does permit calling modules written in other languages so, as in C.36 [OTR], additional steps are required to verify the number and type-correctness of such parameters.

SPARK also allows a subprogram body to be written in full-blown Ada (not SPARK). In this case, the subprogram body is said to be "hidden", and no static analysis is performed by a SPARK Processor. For such hidden bodies, some alternative means of verification must be employed, and the advice of Annex Ada should be applied.

G.37 Recursion [GDL]

SPARK does not permit recursion, so this vulnerability is prevented [SLRM 6].

G.38 Ignored Error Status and Unhandled Exceptions [OYB]

SPARK mitigates this vulnerability.

In SPARK, the normal approach is to use static analysis to prove that predefined exceptions cannot be raised. User-defined exceptions are not permitted.

As recommended in C.38.2, it may be appropriate to retain a single "top-level" exception handler for each task as an additional defense.

The vulnerability relating to an ignored error status is prevented by SPARK through static information flow analysis [IFA].

G.39 Termination Strategy [REU]

SPARK mitigates this vulnerability.

SPARK permits a limited subset of Ada's tasking facilities known as the "Ravenscar Profile" [SLRM 9]. There is no nesting of tasks in SPARK, and all tasks are required to have a top-level loop which has no exit statements, so this vulnerability does not apply in SPARK.

SPARK is also amenable to static analysis for the absence of predefined exceptions [SB 11], thus mitigating the case where a task terminates prematurely (and silently) owing to an unhandled predefined exception.

G.40 Type-breaking Reinterpretation of Data [AMV]

SPARK mitigates this vulnerability.

SPARK permits the instantiation and use of Unchecked_Conversion as in Ada. The result of a call to Unchecked_Conversion is not assumed to be valid, so static verification tools can then insist on re-validation of the result before further analysis can succeed [SB 11].

At the time of writing, SPARK does not permit discriminated records, so vulnerabilities relating to discriminated records and unchecked unions are prevented.

G.41 Memory Leak [XYL]

SPARK prevents this vulnerability.

SPARK does not permit the use of access types, storage pools, or allocators, so this vulnerability cannot occur [SLRM 3]. In SPARK, all objects have a fixed size in memory, so the language is also amenable to static analysis of worst-case memory usage.

G.42 Templates and Generics [SYM]

At the time of writing, SPARK does not permit the use of generics units, so this vulnerability is currently prevented. In future, the SPARK language may be extended to permit generic units, in which case section C.42 [SYM] applies.

G.43 Inheritance [RIP]

SPARK mitigates this vulnerability.

SPARK permits only a subset of Ada's inheritance facilities to be used. Multiple inheritance, class-wide operations and dynamic dispatching are not permitted, so all vulnerabilities relating to these language features do not apply to SPARK [SLRM 3.8].

SPARK is also designed to be amenable to static verification of the Liskov Substitution Principle [LSP].

G.44 Extra Intrinsics [LRM]

SPARK prevents this vulnerability in the same way as Ada. See C.44 [LRM].

G.45 Argument Passing to Library Functions [TRJ]

SPARK mitigates this vulnerability.

SPARK includes all of the mitigations of Ada with respect to this vulnerability, but goes further, allowing preconditions to be checked statically by a theorem-prover. The language in which such preconditions are expressed is also substantially more expressive than Ada's type system.

G.46 Inter-language Calling [DJS]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.46 [DJS].

G.47 Dynamically-linked Code and Self-modifying Code [NYY]

SPARK prevents this vulnerability in the same way as Ada. See C.47 [NYY].

G.48 Library Signature [NSQ]

SPARK prevents this vulnerability in the same way as Ada. See C.48 [NSQ].

G.49 Unanticipated Exceptions from Library Routines [HJW]

SPARK prevents this vulnerability in the same way as Ada. See C.49 [HJW]. SPARK does permit the use of exception handlers, so these may be used to catch unexpected exceptions from library routines.

G.50 Pre-Processor Directives [NMP]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.50 [NMP].

G.51 Suppression of Language-defined Run-time Checking [MXB]

SPARK mitigates this vulnerability through static analysis. In particular, theorem-proving can be used to verify that a run-time check can never fail, allowing such checks to be suppressed with confidence [SB 11].

G.52 Provision of Inherently Unsafe Operations [SKL]

As in Ada, SPARK allows the use of Unchecked_Conversion, so the advice of C.52 applies here.

SPARK allows provides a provision for “hidden” bodies – units not written in SPARK at all that are ignored by a SPARK Processor. These units are assumed to be written in Ada, so for these units, the advice of the entire Ada Annex should be applied.

G.53 Obscure Language Features [BRS]

SPARK mitigates this vulnerability.

The design of the SPARK subset avoids many language features that might be said to be “obscure” or “hard to understand”, such as controlled types, unrestricted tasking, anonymous access types and so on.

SPARK goes further, though, in aiming for a completely *unambiguous* semantics, removing all erroneous and implementation-dependent features from the language. This means that a SPARK program should have a single meaning to programmers, reviewers, maintainers and all compilers.

SPARK also bans the aliasing, overloading, and redeclaration of names, so that one entity only ever has one name and one name can denote at most one entity, further reducing the risk of mis-understanding or mis-interpretation of a program by a person, compiler or other tools.

G.54 Unspecified Behaviour [BQF]

SPARK prevents this vulnerability. There are no unspecified behaviours.

G.55 Undefined Behaviour [EWF]

SPARK prevents this vulnerability through subsetting and static analysis. The language is designed to exhibit no undefined behaviours.

G.56 Implementation-Defined Behaviour [FAB]

SPARK mitigates this vulnerability.

SPARK allows a number of implementation-defined features as in Ada. These include:

- The range of predefined integer types.
- The range and precision of predefined floating-point types.
- The range of System.Any_Priority and its subtypes.
- The value of constants such as System.Max_Int, System.Min_Int and so on.
- The selection of T'Base for a user-defined integer or floating-point type T.
- The rounding mode of floating-point types.

In the first four cases, static analysis tools can be configured to "know" the appropriate values [SB 9.6]. Care must be taken to ensure that these values are correct for the intended implementation. In the fifth case, SPARK defines a contract to indicate the choice of base-type, which can be checked by a pragma Assert. In the final case, additional static analysis of numerical precision must be performed by the user to ensure the correctness of floating-point algorithms.

G.57 Deprecated Language Features [MEM]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See C.57 [MEM].

G.58 Implications for standardization

See C.58.

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, *International terminology standards — Preparation and layout*

[4] ISO/IEC 9899:2011, *Programming languages* — C

[5] ISO/IEC 9899:1999/Cor.1:2001, *Technical Corrigendum 1*

[6] ISO/IEC 9899:1999/Cor.1:2004, *Technical Corrigendum 2*

[7] ISO/IEC 9899:1999/Cor.1:2007, *Technical Corrigendum 3*

[8] ISO/IEC 1539-1:2004, *Information technology — Programming languages* — Fortran — Part 1: Base lannguge

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

[10] ISO/IEC 14882:2011, *Programming languages* — C++

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

[12] Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2004 (second edition)[20].

[13] ISO/IEC TR24731–1, *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces*

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

[15] Joint Strike Fighter Air Vehicle: C++ Coding Standards for the System Development and Demonstration Program. Lockheed Martin Corporation. December 2005.

[16] Motor Industry Software Reliability Association. *Guidelines for the Use of the C++ Language in critical systems*, June 2008

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

[20] The first edition should not be used or quoted in this work.

[18] L. Hatton, Safer C: developing software for high-integrity and safety-critical systems. McGraw-Hill 1995

[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.

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

[26] *ARIANE 5: Flight 501 Failure*, Report by the Inquiry Board, July 19, 1996 http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[27] Hogaboom, Richard, *A Generic API Bit Manipulation in C*, Embedded Systems Programming, Vol 12, No 7, July 1999 http://www.embedded.com/1999/9907/9907feat2.htm

[28] Carlo Ghezzi and Mehdi Jazayeri, Programming Language Concepts, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998

[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.

[30] Seacord, R. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005. See http://www.cert.org/books/secure-coding for news and errata.

[31] John David N. Dionisio. Type Checking. http://myweb.lmu.edu/dondi/share/pl/type-checking-v02.pdf

[32] MISRA Limited. "MISRA C: 2004 Guidelines for the Use of the C Language in Critical Systems." Warwickshire, UK: MIRA Limited, October 2004 (ISBN 095241564X).

[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

[40] CERT. *CERT C++ Secure Coding Standard*. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637 (2009).

[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

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

Index