TECHNICAL REPORT

ISO/IEC TR 24772

First edition
2010-10-01

# Information technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages through language selection and use

*Technologies de l'information — Langages de programmation — Conduite pour éviter les vulnérabilités dans les langages de programmation à travers la sélection et l'usage de la langue*

Reference number
ISO/IEC TR 24772:2010(E)

## Contents

## Foreword

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

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

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

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

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

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

## Introduction

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)
- Concurrency
- Numerical analysis (although some simple items regarding the use of floating point are described in PLF)
- Scripting languages
- Inter-language operability
- Language-specific annexes

# 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**
**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.

**3.1.2**
**application vulnerability**
security vulnerability or safety hazard, or defect

**3.1.3**
**security vulnerability**
weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat

**3.1.4**
**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".

**Note 2**: IEC 61508 is titled "Functional safety of electrical/electronic/ programmable electronic safety-related systems", with part 4 being "Definitions and abbreviations". Hence within IEC 61508 the "safety" context of "safety hazard" is assumed.

**Note 3:** IEC 61508 cites ISO/IEC Guide 51 as the source for the definition.

**Note 4**: 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).

**3.1.5**
**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.

**Note 2:** 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.

**3.1.6**
**software quality**
degree to which software implements the requirements described by its specification and the degree to which the characteristics of a software product fulfil its requirements

**3.1.7**
**predictable execution**
property of the program such that all possible executions have results that can be predicted from the source code

**Note 1:** All the relevant language-defined implementation characteristics and knowledge of the universe of execution must be known.

**Note 2:** In some environments, this would raise issues regarding numerical stability, exceptional processing, and concurrent execution.

**Note 3:** Predictable execution is an ideal that must be approached keeping in mind the limits of human capability, knowledge, availability of tools, and other factors. Neither this Technical Report nor any standard ensures predictable execution. Rather this Technical Report provides advice on improving predictability. The purpose of this Technical Report is to assist a reasonably competent programmer to approach the ideal of predictable execution.

**Note 4**: The following terms are used in relation to “Predictable execution”.

- **Unspecified behaviour:** A situation where the implementation of a language will have to make some choice from a finite set of alternatives, but that choice is not in general predictable by the programmer, for example, the order in which sub-expressions are evaluated in an expression in many languages.
- **Implementation-defined behaviour:** A situation where the implementation of a language will have to make some choice, and it is required that this choice be documented and available to the programmer.
- **Undefined behaviour:** A situation where the definition of a language can give no indication of what behaviour to expect from a program – it can be some form of catastrophic failure (a ‘crash’) or continued execution with some arbitrary data.

**Note 5**: This Technical Report includes a clause on **Unspecified functionality**. This notion is related to neither unspecified behaviour, which is a characteristic of an application, nor the language used to develop the application.

## 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 Not in Scope

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 to be assured is not treated.

The specification of an application is *not* within the scope.

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.

## 4.2 Approach

Guidelines based on this Technical Report are likely to be highly leveraged in that they are likely to affect many times more people than the number that worked on them. Therefore guidelines such as these have the potential to make large savings, for a small cost, or to generate large unnecessary costs, for little benefit. For this reason, the writers of this Technical Report have taken a cautious approach to identifying programming language vulnerabilities. New vulnerability descriptions can be added over time, as experience and evidence are accumulated.

## 4.3 Intended Audience

The intended audience for this Technical Report is those who are concerned with assuring the existence of a critical property in 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.

As described in the following paragraphs, 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.

That should not be taken to mean that other developers can ignore this Technical Report. A weakness in an application that of itself has no direct criticality may provide the route by which an attacker gains control of a system or may otherwise disrupt co-hosted applications that are safety, security or mission critical.

It is hoped that such developers would use this Technical Report to ensure that common vulnerabilities are removed or at least minimized from all applications.

Several specific audiences for this International Technical Report have been identified and are described below.

### 4.3.1 Safety-Critical Applications

Users who may benefit from this Technical Report include those developing, qualifying, or maintaining a system where it is critical to prevent behaviour that might lead to:

- loss of human life or human injury, or
- damage to the environment.

### 4.3.2 Security-Critical Applications

Users who may benefit from this Technical Report include those developing, qualifying, or maintaining a system where it is critical to exhibit security properties of:

- confidentiality,
- integrity, and
- availability.

### 4.3.3 Mission-Critical Applications

Users who may benefit from this Technical Report include those developing, qualifying, or maintaining a system where it is critical to prevent behaviour that might lead to:

- property loss or damage, or
- economic loss or damage.

### 4.3.4 Business-Critical Systems

Users who may benefit from this Technical Report include those developing, qualifying, or maintaining a system whose correctness and integrity are essential for successful operation of a business or enterprise.

### 4.3.5 Modeling and Simulation Applications

People who may benefit from this Technical Report include those who are primarily experts in areas other than programming but need to use computation as part of their work. Such people include scientists, engineers, economists, and statisticians. They require high confidence in the applications they write and use because of the increasing complexity of the calculations made (and the consequent use of teams of programmers each contributing expertise in a portion of the calculation), or to the costs of invalid results, or to the expense of individual calculations implied by a very large number of processors used and/or very long execution times needed to complete the calculations. These circumstances give a consequent need for high reliability and motivate the need felt by these programmers for the guidance offered in this Technical Report.

## 4.4 How to Use This Document

This Technical Report gathers language-independent descriptions of programming language vulnerabilities, as well as selected application vulnerabilities, which have occurred in the past and are likely to occur again. 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 Technical Report 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.

The main part of this Technical Report contains descriptions that are intended to be language-independent to the greatest possible extent. Future editions will include annexes that 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 following clauses include suggestions for ways of avoiding the vulnerabilities. It should be noted that these include techniques that can be applied during development, and those that must be implemented as run-time checks. The former are likely to be appropriate to all applications. For some applications, it is relatively more important to ensure that potential run-time errors are eliminated during development because there may be insufficient opportunity to recover from them. For long-running simulations, run-time checks may increase the run-time to the point where the prediction loses value, or to the point where the cost of calculation becomes prohibitive given the value of the simulation results. 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.

Annexes A through C are templates and guidelines that were used in the identification, selection, and creation of the vulnerabilities that were generated for clauses 6 and 7. They can be used to guide the generation of future language vulnerabilities and application vulnerabilities.

Annex D, *Vulnerability Outline 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 E, *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. Future revisions of this Technical Report are planned to contain language-specific annexes that are developed using Annex E.

# 5 Vulnerability issues

*Software vulnerabilities* are unwanted characteristics of software that may allow software to behave in ways that are unexpected by a reasonably sophisticated user of the software. The expectations of a reasonably sophisticated user of software may be set by the software's documentation or by experience with similar software. Programmers introduce vulnerabilities into software by failing to understand the expected behaviour (the software requirements), or by failing to correctly translate the expected behaviour into the actual behaviour of the software.

This Technical Report does not discuss a programmer's understanding of software requirements or the completeness of the requirements. This Technical Report does not discuss software engineering issues per se. This Technical Report does not discuss configuration management, build environments, code-checking tools, nor software testing. This Technical Report does not discuss the classification of software vulnerabilities according to safety or security concerns. This Technical Report does not discuss the costs of software vulnerabilities, or the costs of preventing them.

This Technical Report does discuss a reasonably competent programmer's failure to translate the understood requirements into correctly functioning software. This Technical Report does discuss programming language features known to contribute to software vulnerabilities. That is, this Technical Report discusses issues arising from those features of programming languages found to increase the frequency of occurrence of software vulnerabilities. The intention is to provide guidance to those who wish to specify coding guidelines for their own particular use.

A programmer writes source code in a programming language to translate the understood requirements into working software. The programmer selects and codes constructs specified by a programming language with the intention of achieving a written expression of the desired behaviour.

A program's expected behaviour might be stated in a complex technical document, which can result in a complex sequence of features of the programming language. Software vulnerabilities occur when a reasonably competent programmer fails to understand the totality of the effects of the language features combined to construct the software. The overall software may be a very complex technical document itself (written in a programming language whose definition is also a complex technical document).

The recommendations contained in this Technical Report might also be considered to be code quality issues. Both kinds of issues might be addressed through the use of a systematic development process, use of development/analysis tools and thorough testing.

## 5.1 Issues arising from incomplete or evolving language specifications

While there are many millions of programmers in the world, there are only several hundreds of authors engaged in designing and specifying those programming languages defined by international standards. The design and specification of a programming language is very different from programming. Programming involves selecting and sequentially combining features from the programming language to (locally) implement specific steps of the software's design. In contrast, the design and specification of a programming language involves (global) consideration of all aspects of the programming language. This must include how all the features will interact with each other, and what effects each will have, separately and in any combination, under all foreseeable circumstances. Thus, language design has global elements that are not generally present in any local programming task.

The creation of the abstractions which become programming language standards therefore involve consideration of issues unneeded in many cases of actual programming. Therefore perhaps these issues are not routinely considered when programming in the resulting language. These global issues may motivate the definition of subtle distinctions or changes of state not apparent in the usual case wherein a particular language feature is used. Authors of programming languages may also desire to maintain compatibility with older versions of their language while adding more modern features to their language and so add what appears to be an inconsistency to the language.

For example, some languages may allow a subprogram to be invoked without specifying the correct signature of the subprogram. This may be allowed to keep compatibility with earlier versions of the language where such usage was permitted, and despite the knowledge that modern practice demands the signature be specified. Specifically, the programming language C does not require a function prototype be within scope[1]. The programming language Fortran does not require an explicit interface. Thus, language usage is improved by coding standards specifying that the signature be present.

A reasonably competent programmer therefore may not consider the full meaning of every language feature used, as only the desired (local or subset) meaning may correspond to the programmer's immediate intention. In consequence, a subset meaning of any feature may be prominent in the programmer's overall experience.

Further, the combination of features indicated by a complex programming goal can raise the combinations of effects, making a complex aggregation within which some of the effects are not intended.

### 5.1.1 Compiler Selection

Compiler selection is important to ensure a system operates safely and securely. Compilers are important as they are the intermediary between the human readable source code and the machine readable binary code. This crucial step is often overlooked and compilers, unless coming from a trusted source and developed according to agreed standards, should be treated as any other commercial off the shelf software that has an unknown pedigree.

Often, developers analyze the source code to detect any code that can negatively impact security or safety. This aims to solve one part of the problem. After the source has been compiled, there is a need to be sure that the compiler did not insert any logic (maliciously or inadvertently) into the binary that compromises the systems security or safety. This is especially important because this type of vulnerability will be inserted into every piece of software that the compiler processes.

To combat against this, developers of security or safety critical systems should only use compilers from a trusted source. The trusted source should also provide evidence that the compiler is free from anomalous behaviour; similar to the way RTCA's DO-178B defines qualifiable tools. In addition, developers of critical software can

[1] This feature has been deprecated in the 1999 version of the ISO C Standard [4].

perform source to binary traceability to ensure the compiler has not inserted any undesired logic into the binary code.

If a compiler has many options, then developers should consider if the options to be used in the project are the same options used when the compiler was validated.

### 5.1.2 Issues arising from unspecified behaviour

While every language standard attempts to specify how software written in the language will behave in all circumstances, there will always be some behaviour that is not specified completely. In any circumstance, of course, a particular compiler will produce a program with some specific behaviour (or fail to compile the program at all). Where a programming language construct is insufficiently defined, different compilers may generate different behaviours from the same source code. The authors of language standards often have an interpretations or defects process in place to treat these situations once they become known, and, eventually, to specify one particular behaviour. However, the time needed by the process to produce corrections to the language standard is often long, as careful consideration of the issues involved is needed.

When programs are compiled with only one compiler, the programmer may not be aware when behaviour not specified by the standard has been produced. Programs relying upon behaviour not specified by the language standard may behave differently when they are compiled with different compilers. An experienced programmer may choose to use more than one compiler, even in one environment, to obtain diagnostics from more than one source. In this usage, any particular compiler must be considered to be a different compiler if it is used with different options (which can give it different behaviour), or is a different release of the same compiler (which may have different default options or may generate different code), or is on different hardware (which may have a different instruction set). In this usage, a different computer may be the same hardware with a different operating system, with different compilers installed, with different software libraries available, with a different release of the same operating system, or with a different operating system configuration.

### 5.1.3 Issues arising from implementation-defined behaviour

In some situations, a programming language standard may specifically allow compilers to support a range of behaviours to a given language feature or combination of features. This may enable a more efficient execution on a wider range of hardware, or enable use of the programming language in a wider variety of circumstances.

To allow use on a wide range of hardware, for example, many languages do not specify the amount of storage reserved for language-defined entities such as variables. The degree to which a diligent programmer may obtain information on the amount of storage reserved for entities varies among languages.

The authors of language standards are encouraged to provide lists of all allowed variations of behaviour (as many already do). Such a summary will benefit applications programmers, those who define applications coding standards, and those who make code analysis tools.

### 5.1.4 Issues arising from undefined behaviour

In some situations, a programming language standard may specify that program behaviour is undefined. While the authors of language standards naturally try to minimize these situations, they may be inevitable when attempting to define software recovery from errors, or other situations recognized as being incapable of precise definition.

An example of undefined behaviour, in many languages, is the use of the value of a variable to which there has not yet been an assignment.

## 5.2 Issues arising from human cognitive limitations

The goal in the creation of a programming language is to provide a tool that may be used by the writers of software to manipulate data and produce a desired result. Some programming languages are general purpose languages, while others are targeted to specific tasks or needs. Even general purpose languages may be created with a specific user in mind, for example C was created by engineers for programmers implementing system software, while COBOL was created for business analysts and business programmers.

All humans are different. Constructs that may be easily understood by mathematicians may be confusing to a business analyst. Also, similar constructs in different languages (the use of '=' in COBOL for comparison, and in C for assignment), and similar constructs within the same language (the use of parentheses for both function parameter lists, macro parameter lists, and for array subscripts) can add to this confusion. In addition, many languages provide multiple syntaxes to accomplish the same task, and coders will choose those syntaxes that make the most sense to them, again adding additional confusion in program creation and maintenance.

Humans are also fallible, and can only comprehend a limited number of interactions within a process before that process must be subdivided into smaller segments. In addition, stress, whether internal pressures and deadlines or external influences totally unrelated to the task at hand, can interfere with the process of software creation. These factors combined with language constructs that may be confusing can lead to failures due to human fallibility. These failures can include:

- Cognitive shortcomings, which are failures of design and implementation. These may be
  - faulty reasoning and
  - incomplete solutions due to lack of time and effort in comprehending the task.
- Knowledge shortcomings, which are failures of training and environment. These can include
  - incomplete and/or incorrect knowledge of appropriate language semantics,
  - incomplete and/or incorrect knowledge of how the chosen syntax will be executed by a particular implementation, and
  - incomplete and/or incorrect knowledge of the internal and external interactions of the various software components involved.
- Judgment shortcomings, which are failures of choice and will, and may include
  - selection of simpler but vulnerable constructs in place of more robust but more time consuming solutions and
  - selection of terse constructs, which may be less understandable, in place of verbose, maintainable solutions.

This technical report identifies issues related to the use of programming languages that can increase the likelihood of errors due to cognitive limitations, and recommends ways that can be used to mitigate or eliminate these errors. Some of the mechanisms recommended in this technical report include reducing the cognitive effort necessary in reading existing source code, reducing the amount of knowledge needed by readers of existing source code, and reducing the probability that incorrect developer knowledge will result in unpredictable code execution.

## 5.3 Issues arising from a lack of predictable execution

If a reasonably competent programmer has a good understanding of the intent of a program after reading source code as far as a particular line of code, the programmer ought to have a good understanding of the state of the program after reading the next line of code. However, some features, or, more likely, some combinations of features, of programming languages are associated with relatively decreased rates of the programmer's maintaining their understanding as they read through a program. It is these features and combinations of features that are indicated in this Technical Report, along with ways to increase the programmer's understanding as code is read.

Here, the term understanding means the programmer's recognition of all effects, including subtle or unintended changes of state, of any language feature or combination of features appearing in the program. This view does not imply that programmers only read code from beginning to end. It is simply a statement that a line of code changes the state of a program, and that a reasonably competent programmer ought to understand the state of the program both before and after reading any line of code. It is only to a first approximation that code is read and understood line by line.

## 5.4 Issues arising from the lack of portability and interoperability

The representation of characters, the representation of true/false values, the set of valid addresses, the properties and limitations of any (fixed-point or floating-point) numerical quantities, and the representation of programmer-defined types and classes may vary among hardware, among languages (affecting inter-language software development), and among compilers of a given language. These variations may be the result of hardware differences, operating system differences, library differences, compiler differences, or different configurations of the same compiler (as may be set by environment variables or configuration files). In each of these circumstances, there is an additional burden on the programmer because part of the program's behaviour is indicated by a factor that is not a part of the source code. That is, the program's behaviour may be indicated by a factor that is invisible when reading the source code. Compilation control schemes (IDE projects, make, and scripts) further complicate this situation by abstracting and manipulating the relevant variables (target platform, compiler options, libraries, and so forth).

Many compilers of standard-defined languages also support language features that are not specified by the language standard. These non-standard features are called extensions. For portability, the programmer must be aware of the language standard, and use only constructs with standard-defined semantics. The motivation to use extensions may include the desire for increased functionality within a particular environment, or increased efficiency on particular hardware. There are well-known software engineering techniques for minimizing the ill effects of extensions; these techniques should be a part of any coding standard where they are needed, and they should be employed whenever extensions are used. These issues are software engineering issues and are not further discussed in this Technical Report.

Some language standards define libraries that are available as a part of the language definition. Such libraries are an intrinsic part of the respective language and are called intrinsic libraries. There are also libraries defined by other sources; these are called non-intrinsic libraries.

The use of non-intrinsic libraries to broaden the software primitives available in a given development environment is a useful technique, allowing the use of trusted functionality directly in the program. Libraries may also allow the program to bind to capabilities provided by an environment. However, these advantages are potentially offset by any lack of skill on the part of the designer of the library (who may have designed subtle or undocumented changes of state into the library's behaviour), and implementer of the library (who may not have implemented the library identically on every platform), and even by the availability of the library on a new platform. The quality of the documentation of a third-party library is another factor that may decrease the reliability of software using a library in a particular situation by failing to describe clearly the library's full behaviour. If a library is missing on a new platform, its functionality must be recreated to port any software depending upon the missing library**.** The re-creation may be burdensome if the reason the library is missing is because the underlying capability for a particular environment is missing. The pedigree of the library must also be assured, the library must come from a trusted source without adulteration in transmission or storage.

Using a non-intrinsic library usually requires that options be set during compilation and linking phases, which constitute a software behaviour specification beyond the source code. Again, these issues are software engineering issues and are not further discussed in this Technical Report**.**

## 5.5 Issues arising from inadequate language intrinsic support

Many languages are created to facilitate programming within an application domain. Some languages are specifically designed for programming of business applications, numerical computation or systems programming. Problems can arise when, for example, a language being used to implement a real-time, multi-threaded system lacks key features that are needed such as a way of enforcing mutual exclusion. Such features can be provided by the programming environment in the form of libraries, but the definition of such libraries may be proprietary and inclined to change in later releases. A vendor may even decide to withdraw support entirely for such a library. Also, such a library may not be verified and validated to the same standard as the compiler and the application being developed.

Some potential problems may be preventable through the use of stronger types, or the use of controls such as array bounds checking or integer checking to avoid overflows. These stronger restraints on a language have a cost both in performance and in the flexibility to perform certain operations. Language designers must strike a balance between restraints in the language, performance and flexibility causing some languages to lean heavily toward one or more extremes in pursuit of some language attributes. The intrinsic support provided by a language can help considerably in avoiding vulnerabilities, but such support can cause the utility of programming within a particular application domain to diminish.

## 5.6 Issues arising from language features prone to erroneous use

Certain language constructs are relatively simple and straightforward to use. Other ones are complex to use or easily misused in a legal, but unintended way. Programmers may use floating point variables and pointers without fully understanding the nuances of the data representation. Rarely needed constructs or constructs that can be substituted for a series of simpler constructs can be used without a complete understanding of the full effects of the constructs.

Syntactic language features that are not intolerant of common typo errors can produce some problems that are notoriously difficult to find. One common example of this is that C permits an unintentional assignment to be performed in a Boolean expression by the accidental use of a single "=" (assignment) instead of the intended "==" test for equality. It then allows the resulting value to be treated as a Boolean.

# 6 Programming Language Vulnerabilities

## 6.1 General

The standard for a programming language provides definitions for that language's constructs. This Technical Report will in general use the terminology that is most natural to the description for each individual vulnerability, relying upon the individual standards for terminology details. In general, the reader should be aware that "method", "function", and "procedure" could denote similar constructs in different languages, as can "pointer" and "reference". Situations described as "undefined behaviour" in some languages are known as "unbounded behaviour" in others.

## 6.2 Obscure Language Features [BRS]

### 6.2.1 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.

### 6.2.2 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

### 6.2.3 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.

### 6.2.4 Applicable language characteristics

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

### 6.2.5 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.

### 6.2.6 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.

## 6.3 Unspecified Behaviour [BQF]

### 6.3.1 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.

### 6.3.2 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].

### 6.3.3 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 a 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.

### 6.3.4 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.

### 6.3.5 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.

### 6.3.6 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.

## 6.4 Undefined Behaviour [EWF]

### 6.4.1 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.

### 6.4.2 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].

### 6.4.3 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.

### 6.4.4 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.

### 6.4.5 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).

### 6.4.6 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.

## 6.5 Implementation-defined Behaviour [FAB]

### 6.5.1 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.

### 6.5.2 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 Quaility and Style Guide: 7.1.5 and 7.1.6
See: Unspecified Behaviour [BQF] and Undefined Behaviour [EWF].

### 6.5.3 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.

### 6.5.4 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.

### 6.5.5 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.

### 6.5.6 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.

## 6.6 Deprecated Language Features [MEM]

### 6.6.1 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.

### 6.6.2 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 Quaility and Style Guide: 7.1.1

### 6.6.3 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.

### 6.6.4 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.

### 6.6.5 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.

### 6.6.6 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.

## 6.7 Pre-processor Directives [NMP]

### 6.7.1 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.

### 6.7.2 Cross reference

Holzmann-8
JSF SV 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

### 6.7.3 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.

### 6.7.4 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.

### 6.7.5 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.

### 6.7.6 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.

## 6.8 Choice of Clear Names [NAI]

### 6.8.1 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 some time 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.

### 6.8.2 Cross reference

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

### 6.8.3 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 easily missed in human reviews.

### 6.8.4 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, postconditions, 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.

### 6.8.5 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.

### 6.8.6 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.

## 6.9 Choice of Filenames and other External Identifiers [AJN]

### 6.9.1 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 among variations of the same operating system. For example, Microsoft XP 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.

### 6.9.2 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

### 6.9.3 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.

### 6.9.4 Applicable language characteristics

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

- Any language providing for use of an *API* (Application Programming Interface) for external access of resources with varied naming conventions. In practice, this means all languages.
- A particular language interface to a system should be consistent in its processing of filenames or external identifiers. Consistency is only the first consideration. Even though it is consistent, it may consistently do something that is unexpected by the developer of the software interfacing with the system.

### 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:

- 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.

### 6.9.6 Implications for standardization

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

- Language APIs for interfacing with external identifiers should be compliant with ISO/IEC 9945:2003 (IEEE Std 1003.1-2001).

## 6.10 Unused Variable [XYR]

### 6.10.1 Description of application vulnerability

A variable's value is assigned but never used, making it a dead store. As a variant, a variable is declared but neither read nor written to in the program, making it an unused variable. This type of error suggests that the design has been incompletely or inaccurately implemented.

Unused variables by themselves are innocuous, but can be combined with other vulnerabilities such as index bounds errors and buffer overflows and may mask errors or provide hidden channels.

### 6.10.2 Cross reference

CWE:

563. Unused Variable

MISRA C++ 2008: 0-1-4 and 0-1-6

CERT C guidelines: MSC13-C

### 6.10.3 Mechanism of failure

A variable is declared, but never used. It is likely that the variable is simply vestigial, but it is also possible that the unused variable points out a bug. This is likely to suggest that the design has been incompletely or inaccurately implemented.

A variable is assigned a value but this value is never used thereafter. The assignment is then generally referred to as a dead store. Note that this may be acceptable if the variable is a volatile variable, for which the assignment of a value triggers some external event.

A dead store is indicative of careless programming or of a design or coding error; either the use of the value was forgotten (almost certainly an error) or the assignment was performed even though it was not needed (unless there is a justification for it).

An unused variable or a dead store is very unlikely to be the cause of a vulnerability. However, since compilers diagnose unused variables routinely and dead stores occasionally, their presence is often an indication that compiler warnings are either suppressed or are being ignored by programmers. This observation does not hold for automatically generated code, where it is commonplace to find unused variables and dead stores, introduced to keep the generation process simple and uniform.

### 6.10.4 Applicable language characteristics

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

- Dead stores are possible in any programming language that provides assignment. (Pure functional languages do not have this issue.)
- Unused variables (in the technical sense above) are possible only in languages that provide variable declarations.

### 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:

- Enable detection of unused variables and dead stores in the compiler. The default setting may be to suppress these warnings.

### 6.10.6 Implications for standardization

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

- Languages should consider requiring mandatory diagnostics for unused variables.

## 6.11 Identifier Name Reuse [YOW]

### 6.11.1 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).

### 6.11.2 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 Quaility and Style Guide: 5.6.1 and 5.7.1

### 6.11.3 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).

An example of how interpretations of a programming language can differ, in the following code fragment:

```
int j = 100;
{
  for (int j = 0; j < 10; j++) ;
  std::cout << j << std::endl; // What is the value of j
}
```

According to ISO 14882:2003 (C++) standard the value printed for `j` should be 100, but in some implementations that do not conform to the current version of the standard it will be 10, as the loop counter `j` remains in-scope after the end of the loop statement.

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). (See, Choice of Filenames and other External Identifiers [AJN].)

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.

### 6.11.4 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.

### 6.11.5 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.

### 6.11.6 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.

## 6.12 Namespace Issues [BJL]

### 6.12.1 Description of Application Vulnerability

If a language provides separate, non-hierarchical 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 behavior 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.

### 6.12.2 Cross references

[None]

### 6.12.3 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 can not 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 is, 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 attacher.

### 6.12.4 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.

### 6.12.5 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)

### 6.12.6 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 illegal and avoidable by the user, for example, by using names qualified by their originating namespace.

## 6.13 Type System [IHN]

### 6.13.1 Description of application vulnerability

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

### 6.13.2 Cross reference

JSF AV Rule: 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 Quaility and Style Guide: 3.4

### 6.13.3 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. This 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 increase 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.

### 6.13.4 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.

### 6.13.5 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.

### 6.13.6 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.

## 6.14 Bit Representations [STR]

### 6.14.1 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.

### 6.14.2 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 Quaility and Style Guide: 7.6.1 through 7.6.9, and 7.3.1

### 6.14.3 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.

### 6.14.4 Applicable language characteristics

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

- Languages that allow bit manipulations.

### 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:

- 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.

### 6.14.6 Implications for standardization

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

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

## 6.15 Floating-point Arithmetic [PLF]

### 6.15.1 Description of application vulnerability

Most real numbers cannot be represented exactly in a computer.  To represent real numbers, most computers use 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.

### 6.15.2 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 Quaility and Style Guide: 5.5.6 and 7.2.1 through 7.2.8

### 6.15.3 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 IEEE 754 is a commonly used representation for floating-point data types, it is not universally used or required by all computer languages. Some languages predate IEEE 754 and make the support for the standard optional. One IEEE 754 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 IEEE 754, 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).

### 6.15.4 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.

### 6.15.5 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.

### 6.15.6 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 ANSI/IEEE 754 should consider adhering completely to the standard. Examples of standardization that should be considered:
  - C should consider requiring ANSI/IEEE 754 for floating-point arithmetic, rather than providing it as an option, as is the case in ISO/IEC 9899:1999[4].
  - Java should consider fully adhering to ANSI/IEEE 754 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.

## 6.16 Enumerator Issues [CCB]

### 6.16.1 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.

### 6.16.2 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 Quaility and Style Guide: 3.4.2

### 6.16.3 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 undefinable values can be propagated.

If arrays are indexed by enumerations containing nondefault 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 or switch/case statements can result in illegal accesses and possibly unbounded behaviours.

### 6.16.4 Applicable language Characteristics

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

- Languages that do not require full coverage of an enumeration in a switch/case statement.
- Languages that provide a default choice in a switch/case statement.
- 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 illegal 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.

### 6.16.5 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.
- When a language requires full coverage of an enumeration in a switch/case statement, a default choice should not be provided. For languages that do not require full coverage, then a default choice with a possible error generation should be provided to ensure that there is full coverage.
- When named syntax is available for representation setting, coverage analysis can eliminate the order issues and the incomplete coverage issues as long as no default choice is given.

### 6.16.6 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.

## 6.17 Numeric Conversion Errors [FLC]

### 6.17.1 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.

### 6.17.2 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

### 6.17.3 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.

### 6.17.4 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.

### 6.17.5 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.

### 6.17.6 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.

## 6.18 String Termination [CJM]

### 6.18.1 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.

### 6.18.2 Cross reference

CWE:
 170. Improper Null Termination
CERT C guidelines: STR03-C, STR31-C, STR32-C, and STR36-C

### 6.18.3 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.

### 6.18.4 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.

### 6.18.5 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.

### 6.18.6 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.

## 6.19 Boundary Beginning Violation [XYX]

### 6.19.1 Description of application vulnerability

A buffer underwrite condition occurs when an array is indexed outside its lower bounds, or pointer arithmetic results in an access to storage that occurs before the beginning of the intended object.

### 6.19.2 Cross reference

CWE:
124. Boundary Beginning Violation ('Buffer Underwrite')
129. Unchecked Array Indexing
JSF AV Rule: 25
MISRA C 2004: 21.1
MISRA C++ 2008: 5-0-15 to 5-0-18
CERT C guidelines: ARR30-C, ARR32-C, and ARR38-C

### 6.19.3 Mechanism of failure

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

- 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.
- 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.

### 6.19.4 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 arithmetic).
- Languages that do not automatically allocate storage when accessing an array element for which storage has not already been allocated.

### 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:.

- Use of implementation provided functionality to automatically check array element accesses and prevent out-of-bounds accesses.
- 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 type when indexing an array, on the basis that an unsigned 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 language 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).

### 6.19.6 Implications for standardization

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

- Languages that use pointer types should consider specifying a standardized feature for a pointer type that would enable array bounds checking, if such a pointer type is not already in the standard.

## 6.20 Unchecked Array Indexing [XYZ]

### 6.20.1 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.

### 6.20.2 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 Quaility and Style Guide: 5.5.1, 5.5.2, 7.6.7, and 7.6.8

### 6.20.3 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. Due to this fact, consequences range from denial of service, and data corruption, to arbitrary code execution. The most common condition 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 at runtime. 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.

### 6.20.4 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.

### 6.20.5 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.

### 6.20.6 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.

## 6.21 Unchecked Array Copying [XYW]

### 6.21.1 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.

### 6.21.2 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 Quaility and Style Guide: 7.6.7 and 7.6.8

### 6.21.3 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].

### 6.21.4 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].

### 6.21.5 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.

### 6.21.6 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.

## 6.22 Buffer Overflow [XZB]

### 6.22.1 Description of application vulnerability

A buffer overflow arises when, due to unchecked array indexing or unchecked array copying, storage outside the buffer is accessed. Usually overflows 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 overflows describe accesses to contiguous memory beyond the end of the buffer data, as may arise when arrays are copied without length checks. However, accessing before the beginning of the buffer data is equally possible, dangerous and maliciously exploitable.

### 6.22.2 Cross reference

CWE:
  122. Heap-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, STR31-C and MEM35-C

### 6.22.3 Mechanism of failure

Overwriting adjacent data (or data at arbitrarily computed locations) outside the area allocated for an array leads to value failures of the application. The program statements causing the buffer overflow are often difficult to find. But not only data storage can be corrupted. Buffer overflow may also inadvertently or even maliciously overwrite function pointers that may be in memory, pointing them to the attacker's code. Even in applications that do not explicitly use function pointers, the run-time will usually store function pointers in memory. For example, object methods in object-oriented languages are generally implemented using function pointers in data structures that are kept in memory. Since the consequence of a buffer overflow can be targeted to cause arbitrary code execution, this vulnerability can be used to subvert any security service.

### 6.22.4 Applicable language characteristics

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

- Copying of arrays can be done without an automatic length check ensuring that source and target locations are of the same size.
- Indexing of array elements can be done without an automatic check that the indexing is within the bounds of the array.
- Accesses might violate the physical bounds of the entire array or violate the logical bounds of a particular extent. The vulnerability is somewhat mitigated, if the former violation is checked for and detected by the implementation although the latter is not.
- Languages that provide bounds checking but permit the check to be suppressed.
- The bounds of an array are not automatically extended to accommodate accesses that might otherwise have been beyond the bounds. (This may or may not match the programmer's intent.)

### 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:

- Use a language or compiler that performs automatic bounds checking on elements accesses and automatic length checking on copying entire arrays.
- Use an abstraction library to add checks on top of library functions that copy arrays without length checks.
- Checks that prevent overflows can be disabled in some languages to increase performance. This option should be used rarely.
- Implementation-defined checks that prevent overflows can be enabled in some languages that do not require such checks. This option should be used whenever feasible.

### 6.22.6 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. This capability may need to be disabled at times for performance reasons.

## 6.23 Pointer Casting and Pointer Type Changes [HFC]

### 6.23.1 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.

### 6.23.2 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 Quaility and Style Guide: 7.6.7 and 7.6.8

### 6.23.3 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.

### 6.23.4 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.

### 6.23.5 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.

### 6.23.6 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.

## 6.24 Pointer Arithmetic [RVG]

### 6.24.1 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.

### 6.24.2 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

### 6.24.3 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.

### 6.24.4 Applicable language characteristics

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

- Languages that allow pointer arithmetic.

### 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:

- Use pointer arithmetic only for indexing objects defined as arrays.
- Prefer indexing for accessing array elements rather than using pointer arithmetic.
- Limit pointer arithmetic calculations to the addition and subtraction of integers.

### 6.24.6 Implications for standardization

[None]

## 6.25 Null Pointer Dereference [XYH]

### 6.25.1 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.

### 6.25.2 Cross reference

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

### 6.25.3 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.

### 6.25.4 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.

### 6.25.5 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.

### 6.25.6 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.

## 6.26 Dangling Reference to Heap [XYK]

### 6.26.1 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.

### 6.26.2 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 Quaility and Style Guide: 5.4.5, 7.3.3, and 7.6.6

### 6.26.3 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.

### 6.26.4 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.

### 6.26.5 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.

### 6.26.6 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.

## 6.27 Templates and Generics [SYM]

### 6.27.1 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.

### 6.27.2 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

### 6.27.3 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 illegal, 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.

### 6.27.4 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.

### 6.27.5 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.

### 6.27.6 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.

## 6.28 Inheritance [RIP]

### 6.28.1 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.

### 6.28.2 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 Quaility and Style Guide: 9 (complete clause)

### 6.28.3 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.

### 6.28.4 Applicable language characteristics

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

- Languages that allow single and multiple inheritances.

### 6.28.5 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.

### 6.28.6 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.

## 6.29 Initialization of Variables [LAV]

### 6.29.1 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.

### 6.29.2 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
CERT C guidelines: DCL14-C and EXP33-C
MISRA C++ 2008: 8-5-1
Ada Quaility and Style Guide: 5.9.6

### 6.29.3 Mechanism of failure

Uninitialized objects may have illegal values, legal but wrong values, or legal 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.

### 6.29.4 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.

### 6.29.5 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.
- 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.

### 6.29.6 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.

## 6.30 Wrap-around Error [XYY]

### 6.30.1 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 if so specified by the language semantics "wraps around" to either a very small, negative, or undefined value. Using shift operations as a surrogate for multiply or divide may produce a similar error.

### 6.30.2 Cross reference

CWE:
  128. Wrap-around Error
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

### 6.30.3 Mechanism of failure

Due to how arithmetic is performed by computers, if a variable is incremented 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.

Shift operations may also produce values that cannot be easily predicted as a result of the different representations of negative integers on various hardware, and, when treating signed quantities, of the differences in behaviour between logical shifts and arithmetic shifts (the particular effect of filling with the sign bit).

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.

### 6.30.4 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.
- Languages that do not fully specify the distinction between arithmetic and logical shifts.

### 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:

- 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.
- 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.

### 6.30.6 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.

## 6.31 Sign Extension Error [XZI]

### 6.31.1 Description of application vulnerability

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

### 6.31.2 Cross reference

CWE:

194. Incorrect Sign Extension

MISRA C++ 2008: 5-0-4

CERT C guidelines: INT13-C

### 6.31.3 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.

### 6.31.4 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.

### 6.31.5 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.

### 6.31.6 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.

## 6.32 Operator Precedence/Order of Evaluation [JCW]

### 6.32.1 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

### 6.32.2 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 Quaility and Style Guide: 7.1.8 and 7.1.9

### 6.32.3 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`".

### 6.32.4 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.

### 6.32.5 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 C++ [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.

### 6.31.6 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.

## 6.33 Side-effects and Order of Evaluation [SAM]

### 6.33.1 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.

### 6.33.2 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 Quaility and Style Guide: 7.1.8 and 7.1.9

### 6.33.3 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.

### 6.33.4 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.

### 6.33.5 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.

### 6.33.6 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.

## 6.34 Likely Incorrect Expression [KOA]

### 6.34.1 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 legal 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.

### 6.34.2 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

### 6.34.3 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

### 6.34.4 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.

### 6.34.5 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.

### 6.34.6 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 legal in the same context. For example, = is not generally legal in an `if` statement in Java because it does not normally return a boolean value.

## 6.35 Dead and Deactivated Code [XYQ]

### 6.35.1 Description of application vulnerability

Dead and Deactivated code (the distinction is addressed in 6.35.3) 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 is undesirable because it indicates the possibility of a coding error and because it may provide a "jump" target for an intrusion.

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

### 6.35.2 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

### 6.35.3 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.

The presence of dead code is not in itself an error, but begs the question why is it there? Is its presence an indication that the developer believed it to be necessary, but some error means it will never be executed? Or is there a legitimate reason 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.

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.

### 6.35.4 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.

### 6.35.5 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 endeavour to remove, as a first resort and as far as practical, dead code from an application.
- 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

### 6.35.6 Implications for standardization

[None]

## 6.36 Switch Statements and Static Analysis [CLL]

### 6.36.1 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.

### 6.36.2 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 Quaility and Style Guide: 5.6.1 and 5.6.10

### 6.36.3 Mechanism of failure

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

### 6.36.4 Applicable language characteristics

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

- Contain a construct, such as a `switch` statement, that provides a selection among alternative control flows based on the evaluation of an expression.

### 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:

- 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 its possible set of values is known statically and is small in number (as compared, for example, to the value set of an integer variable). In languages that don't provide enumerated types, a tightly constrained integer sub-type might be a good alternative. 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. (Using multiple labels on individual alternatives is not a violation of this recommendation, though.) 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.

### 6.36.6 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.

## 6.37 Demarcation of Control Flow [EOJ]

### 6.37.1 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.

### 6.37.2 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 Quaility and Style Guide: 3, 5.6.1 through 5.6.10

### 6.37.3 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.

### 6.37.4 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.

### 6.37.5 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.

### 6.37.6 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.

## 6.38 Loop Control Variables [TEX]

### 6.38.1 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/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.

### 6.38.2 Cross reference

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

### 6.38.3 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.

### 6.38.4 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.

### 6.38.5 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 loop control variable. 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).

### 6.38.6 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.

## 6.39 Off-by-one Error [XZH]

### 6.39.1 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.

### 6.39.2 Cross reference

CWE:
193. Off-by-one Error

### 6.39.3 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 illegal 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.

### 6.39.4 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.

### 6.39.5 Avoiding the vulnerability or mitigating its effects

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

- A 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.

### 6.39.6 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.

## 6.40 Structured Programming [EWD]

### 6.40.1 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.

### 6.40.2 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 Quaility and Style Guide: 3, 4, 5.4, 5.6, and 5.7

### 6.40.3 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.

### 6.40.4 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.

### 6.40.5 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.

### 6.40.6 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.

## 6.41 Passing Parameters and Return Values [CSJ]

### 6.41.1 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.

### 6.41.2 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 Quaility and Style Guide: 5.2 and 8.3

### 6.41.3 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].

### 6.41.4 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.

### 6.41.5 Avoiding the vulnerability or mitigating its effects

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

- Use 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.

### 6.41.6 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.

## 6.42 Dangling References to Stack Frames [DCM]

### 6.42.1 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. Technically, the stack frame, in which the local variable lived, has been popped and memory may have been reused for a subsequent call. Therefore, the invalidity of the stored address is very difficult to decide. This situation can be described as a "dangling reference to the stack".

### 6.42.2 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 Quaility and Style Guide: 7.6.7, 7.6.8, and 10.7.6

### 6.42.3 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", that 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.

### 6.42.4 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.

### 6.42.5 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.

### 6.42.6 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 subclause 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.

## 6.43 Subprogram Signature Mismatch [OTR]

### 6.43.1 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.

### 6.43.2 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

### 6.43.3 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.

### 6.43.4 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.

### 6.43.5 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.

### 6.43.6 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.

## 6.44 Recursion [GDL]

### 6.44.1 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.

### 6.44.2 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 Quaility and Style Guide: 5.6.6

### 6.44.3 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.

### 6.44.4 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.

### 6.44.5 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.

### 6.44.6 Implications for standardization

[None]

## 6.45 Returning Error Status [NZN]

### 6.45.1 Description of application vulnerability

Unpredicted error conditions—perhaps from hardware (such as an I/O device error), perhaps from software (such as heap exhaustion)—sometimes arise during the execution of code. Programming languages provide a surprisingly wide variety of mechanisms to deal with such errors. The choice of a mechanism that doesn't match the programming language can lead to errors in the execution of the software or unexpected termination of the program. This could lead to a simple decrease in the robustness of a program or it could be exploited in a denial of service attack.

### 6.45.2 Cross reference

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

### 6.45.3 Mechanism of failure

Even in the best-written programs, error conditions sometimes arise. Some errors occur because of defects in the software itself, but some result from external conditions in hardware, such as errors in I/O devices, or in the software system, such as exhaustion of heap space. If left untreated, the effect of the error might result in termination of the program or continuation of the program with incorrect results. To deal with the situation, designers of programming languages have equipped their languages with different mechanisms to detect and treat such errors. These mechanisms are typically intended to be used in specific programming idioms. However, the mechanisms differ among languages. A programmer expert in one language might mistakenly use an inappropriate idiom when programming in a different language with the result that some errors are left untreated, leading to termination or incorrect results. Attackers can exploit such weaknesses in denial of service attacks.

In general, languages make no distinction between dealing with programming errors (like an access to protected memory), unexpected hardware errors (like device error), expected but unusual conditions (like end of file), and even usual conditions that fail to provide the typical result (like an unsuccessful search). This description will use the term "error" to apply to all of the above. The description applies equally to error conditions that are detected via hardware mechanisms and error conditions that are detected via software during execution of a subprogram (such as an inappropriate parameter value).

### 6.45.4 Applicable language characteristics

Different programming languages provide remarkably different mechanisms for treating errors. In languages that provide a number of error detection and treatment mechanisms, it becomes a design issue to match the mechanism to the condition. This clause will describe the mechanisms that are provided in widely used languages.

The simplest case is the set of languages that provide no special mechanism for the notification and treatment of unusual conditions. In such languages, error conditions are signaled by the value of an auxiliary status variable, sometimes a subprogram parameter. The programming language C standard library functions use a variant of this approach; the error status is provided as the return value and sometimes in an additional global error value. Obviously, in such languages, it is imperative to check and act upon the status variable after every call to a subprogram that might provide an error indication. If error conditions can occur in an asynchronous manner, it is necessary to provide means to check for errors in a systematic and periodic manner.

Some languages permit the passing of a label parameter. If an error is encountered, the subprogram returns to the indicated label rather than to the point at which it was called. Similarly some languages accept the name of a subprogram to be used to handle errors. In either case, it is imperative to provide labeled code or a subprogram to deal with all possible error situations.

The approaches described above have the disadvantage that error checking must be provided at every call to a subprogram. This can clutter the code immensely to deal with situations that may occur rarely. For this reason, some languages provide an exception mechanism that automatically transfers control when an error is encountered. This has the potential advantage of allowing error treatment to be factored into distinct error handlers, leaving the main execution path to deal with the usual results. The disadvantages, of course, are that the language design is complicated and the programmer must deal with the conceptually more complex problem of providing error handlers that are removed from the immediate context of a specific call to a subprogram. Furthermore, different languages provide exception-handling mechanisms that differ in the manner in which various design issues are treated:

- How is the occurrence of an exception bound to a particular handler?
- What happens when no handler is local to an exception occurrence? Is the exception propagated in some manner or is it lost?
- What happens after an exception handler executes? Is control returned to the point before the call or after the call, or is the calling routine terminated in some way? If the calling routine is terminated, is there some provision for finalization, such as closing files or releasing resources?
- Are programmers permitted to define additional exceptions?
- Does the language provide default handlers for some exceptions or must the programmer explicitly provide for all of them?
- Can predefined exceptions be raised explicitly?
- Under what circumstances can error checking be disabled?

### 6.45.5 Avoiding the vulnerability or mitigating its effects

Given the variety of error handling mechanisms, it is difficult to write general guidelines. However, dealing with exception handlers can stress the capability of many static analysis tools and can, in some cases, reduce the effectiveness of their analysis. Therefore, for situations where the highest of reliability is required, the application should be designed so that exception handling is not used at all. In the more general case, exception-handling mechanisms should be reserved for truly unexpected situations and other situations (possibly hardware arithmetic overflow) where no other mechanism is available. Situations which are merely unusual, like end of file, should be treated by explicit testing—either prior to the call which might raise the error or immediately afterward.

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.
- In dealing with languages where untreated exceptions can be lost (for example, an exception that goes untreated within an Ada task), it is mandatory to deal with the exception in the local context before it is lost.
- When execution within a particular context is abandoned due to an exception, 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 condition and retry the operation. In such cases, one often treats a symptom but not the underlying problem. It is usually a better solution to finalize and terminate the current context and retreat to a context where the situation is known.
- 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 thought to be impossible.

### 6.45.6 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 (for example, label parameters, auxiliary status variables), but each of the mechanisms should be standardized.

## 6.46 Termination Strategy [REU]

### 6.46.1 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.

### 6.46.2 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 Quaility and Style Guide: 5.8 and 7.5

### 6.46.3 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.

### 6.46.4 Applicable language characteristics

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

### 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:

- 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.

### 6.46.6 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.

## 6.47 Extra Intrinsics [LRM]

### 6.47.1 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.

### 6.47.2 Cross reference

[None]

### 6.47.3 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.

### 6.47.4 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.

### 6.47.5 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.

### 6.47.6 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.

## 6.48 Type-breaking Reinterpretation of Data [AMV]

### 6.48.1 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.

### 6.48.2 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 Quaility and Style Guide: 7.6.7 and 7.6.8

### 6.48.3 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].

### 6.48.4 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.

### 6.48.5 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 form 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.

### 6.48.6 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.

## 6.49 Memory Leak [XYL]

### 6.49.1 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.

### 6.49.2 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 Quaility and Style Guide: 5.4.5, 5.9.2, and 7.3.3

### 6.49.3 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.

### 6.49.4 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.

### 6.49.5 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 (for reuse).

### 6.49.6 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 can specify that implementations must document choices for dynamic memory management algorithms, to help designers decide on appropriate usage patterns and recovery techniques as necessary.

## 6.50 Argument Passing to Library Functions [TRJ]

### 6.50.1 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.

### 6.50.2 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

### 6.50.3 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.

### 6.50.4 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.

### 6.50.5 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.

### 6.50.6 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.
- Libraries should be defined so that all parameters are validated.

## 6.51 Dynamically-linked Code and Self-modifying Code [NYY]

### 6.51.1 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.

### 6.51.2 Cross reference

JSF AV Rule: 2

### 6.51.3 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.

### 6.51.4 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.

### 6.51.5 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.

### 6.51.6 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.

## 6.52 Library Signature [NSQ]

### 6.52.1 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.

### 6.52.2 Cross reference

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

### 6.52.3 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.

### 6.52.4 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.

### 6.52.5 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.

### 6.52.6 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.

## 6.53 Unanticipated Exceptions from Library Routines [HJW]

### 6.53.1 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.

### 6.53.2 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 Quaility and Style Guide: 5.8 and 7.5

### 6.53.3 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 [NZN], Returning Error Status, are also relevant here.

### 6.53.4 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.

### 6.53.5 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.

### 6.53.6 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.

# 7 Application Vulnerabilities

## 7.1 Adherence to Least Privilege [XYN]

### 7.1.1 Description of application vulnerability

Failure to adhere to the principle of least privilege amplifies the risk posed by other vulnerabilities.

### 7.1.2 Cross reference

CWE:
    250. Design Principle Violation: Failure to Use Least Privilege
CERT C guidelines: POS02-C

### 7.1.3 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.

### 7.1.4 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.

## 7.2 Privilege Sandbox Issues [XYO]

### 7.2.1 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.

### 7.2.2 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

CERT C guidelines: POS36-C

### 7.2.3 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.

### 7.2.4 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 SeImpersonate Privilege.

## 7.3 Executing or Loading Untrusted Code [XYS]

### 7.3.1 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.

### 7.3.2 Cross reference

CWE:
114. Process Control
CERT C guidelines: PRE09-C, ENV02-C, and ENV03-C

### 7.3.3 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.

### 7.3.4 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.

## 7.4 Unspecified Functionality [BVQ]

### 7.4.1 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.

### 7.4.2 Cross reference

JSF AV Rule: 127
MISRA C 2004: 2.2, 2.3, 2.4, and 14.1
XYQ: Dead and Deactivated code.

### 7.4.3 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.

### 7.4.4 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.

## 7.5 Distinguished Values in Data Types [KLK]

### 7.5.1 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.

### 7.5.2 Cross reference

CWE:

 20: Improper input validation

 137: Representation errors

JSF AV Rule: 151

### 7.5.3 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.

### 7.5.4 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.

## 7.6 Memory Locking [XZX]

### 7.6.1 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.

### 7.6.2 Cross reference

CWE:
 591. Sensitive Data Storage in Improperly Locked Memory
CERT C guidelines: MEM06-C

### 7.6.3 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.

### 7.6.4 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.

## 7.7 Resource Exhaustion [XZP]

### 7.7.1 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.

### 7.7.2 Cross reference

CWE:
400. Resource Exhaustion

### 7.7.3 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 and MAC 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.

### 7.7.4 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.

## 7.8 Injection [RST]

### 7.8.1 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.

### 7.8.2 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’)
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

### 7.8.3 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.