

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)

AVA Documents
Document Sharing Hub
© ISO/IEC 2010

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2010

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword	vi
Introduction	vii
1 Scope	1
2 Normative references	1
3 Terms and definitions, symbols and conventions	1
3.1 Terms and definitions, symbols and conventions.....	1
3.2 Symbols and conventions.....	3
4 Basic Concepts	4
4.1 Not in Scope.....	4
4.2 Approach	4
4.3 Intended Audience	4
4.4 How to Use This Document	5
5 Vulnerability issues	8
5.1 Issues arising from incomplete or evolving language specifications	8
5.2 Issues arising from human cognitive limitations	11
5.3 Issues arising from a lack of predictable execution	12
5.4 Issues arising from the lack of portability and interoperability	12
5.5 Issues arising from inadequate language intrinsic support.....	13
5.6 Issues arising from language features prone to erroneous use.....	13
6 Programming Language Vulnerabilities	14
6.1 General	14
6.2 Obscure Language Features [BRS]	14
6.3 Unspecified Behaviour [BQF].....	15
6.4 Undefined Behaviour [EWF]	17
6.5 Implementation-defined Behaviour [FAB]	18
6.6 Deprecated Language Features [MEM].....	20
6.7 Pre-processor Directives [NMP].....	21
6.8 Choice of Clear Names [NAI].....	23
6.9 Choice of Filenames and other External Identifiers [AJN].....	25
6.10 Unused Variable [XYR]	26
6.11 Identifier Name Reuse [YOW]	27
6.12 Namespace Issues [BJL].....	30
6.13 Type System [IHN].....	31
6.14 Bit Representations [STR].....	34
6.15 Floating-point Arithmetic [PLF]	35
6.16 Enumerator Issues [CCB]	38
6.17 Numeric Conversion Errors [FLC]	40

6.18	String Termination [CJM]	42
6.19	Boundary Beginning Violation [XYX]	43
6.20	Unchecked Array Indexing [XYZ]	44
6.21	Unchecked Array Copying [XYW]	46
6.22	Buffer Overflow [XZB].....	47
6.23	Pointer Casting and Pointer Type Changes [HFC].....	49
6.24	Pointer Arithmetic [RVG]	50
6.25	Null Pointer Dereference [XYH]	51
6.26	Dangling Reference to Heap [XYK]	52
6.27	Templates and Generics [SYM]	54
6.28	Inheritance [RIP].....	56
6.29	Initialization of Variables [LAV].....	57
6.30	Wrap-around Error [XYY].....	59
6.31	Sign Extension Error [XZI]	60
6.32	Operator Precedence/Order of Evaluation [JCW].....	61
6.33	Side-effects and Order of Evaluation [SAM]	63
6.34	Likely Incorrect Expression [KOA]	64
6.35	Dead and Deactivated Code [XYQ]	66
6.36	Switch Statements and Static Analysis [CLL]	68
6.37	Demarcation of Control Flow [EOJ]	69
6.38	Loop Control Variables [TEX]	70
6.39	Off-by-one Error [XZH].....	71
6.40	Structured Programming [EWD]	73
6.41	Passing Parameters and Return Values [CSJ].....	74
6.42	Dangling References to Stack Frames [DCM].....	77
6.43	Subprogram Signature Mismatch [OTR].....	79
6.44	Recursion [GDL].....	80
6.45	Returning Error Status [NZN]	81
6.46	Termination Strategy [REU]	84
6.47	Extra Intrinsics [LRM].....	85
6.48	Type-breaking Reinterpretation of Data [AMV]	87
6.49	Memory Leak [XYL].....	89
6.50	Argument Passing to Library Functions [TRJ].....	90
6.51	Dynamically-linked Code and Self-modifying Code [NYY]	91
6.52	Library Signature [NSQ]	92
6.53	Unanticipated Exceptions from Library Routines [HJW]	93
7	Application Vulnerabilities.....	95
7.1	Adherence to Least Privilege [XYN]	95
7.2	Privilege Sandbox Issues [XYO]	95
7.3	Executing or Loading Untrusted Code [XYS]	97
7.4	Unspecified Functionality [BVQ]	98
7.5	Distinguished Values in Data Types [KLK]	99
7.6	Memory Locking [XZX]	100

7.7	Resource Exhaustion [XZP]	101
7.8	Injection [RST]	102
7.9	Cross-site Scripting [XYT]	105
7.10	Unquoted Search Path or Element [XZQ]	108
7.11	Improperly Verified Signature [XZR]	108
7.12	Discrepancy Information Leak [XZL]	109
7.13	Sensitive Information Uncleared Before Release [XZK]	110
7.14	Path Traversal [EWR]	111
7.15	Missing Required Cryptographic Step [XZS]	113
7.16	Insufficiently Protected Credentials [XYM]	113
7.17	Missing or Inconsistent Access Control [XZN]	114
7.18	Authentication Logic Error [XZO]	115
7.19	Hard-coded Password [XYP]	117
	Annex A (informative) Guideline Selection Process	118
A.1	Selection Process	118
A.2	Cost/Benefit Analysis	118
A.3	Documenting of the selection process	119
	Annex B (informative) Template for use in proposing programming language vulnerabilities	120
B.1	6.<x> <short title> [<unique immutable identifier>]	120
	Annex C (informative) Template for use in proposing application vulnerabilities	122
C.1	7.<x> <short title> [<unique immutable identifier>]	122
	Annex D (informative) Vulnerability Outline and List	123
D.1	Vulnerability Outline	123
D.2	Vulnerability List	125
	Annex E (informative) Language Specific Vulnerability Template	127
E.1	<language>.1 Identification of standards	127
E.2	<language>.2 General terminology and concepts	127
E.3	<language>.<x> <Vulnerability Name> [<3 letter tag>]	127
	Bibliography	129

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 scope¹. 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
 - o faulty reasoning and
 - o 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
 - o incomplete and/or incorrect knowledge of appropriate language semantics,
 - o incomplete and/or incorrect knowledge of how the chosen syntax will be executed by a particular implementation, and
 - o 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
 - o selection of simpler but vulnerable constructs in place of more robust but more time consuming solutions and
 - o 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 Quality 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 compiler 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 Quality 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 attacker.

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.