DRAFT INTERNATIONAL STANDARD

ISO/IEC DIS 24772-1
ISO/IEC JTC 1/SC 22/WG23 N1338

Date: 2023-11-06

Edition 1

Secretariat: ANSI

Programming languages — Avoiding vulnerabilities in programming languages – Part 1: Language-independent catalogue of vulnerabilities

ISO copyright office

CP 401 • Ch. de Blandonnet 8

CH-1214 Vernier, Geneva

Phone: +41 22 749 01 11

Fax: +41 22 749 09 47

Email: copyright@iso.org

Website: www.iso.org

Published in Switzerland

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC ##, *[name of subcommittee]*.

This first edition of ISO/IEC 24772-1 cancels and replaces ISO/IEC TR 24772-1:2019, which has been technically revised.

The main changes are as follows:

- This International Standard replaces a Technical Report on the same topic
- A new subclause, 3.6 has been added to the terms and definitions to address specific vulnerabilities
- Clause 4 has been expanded to explain how this document is used with programming language standards, safety standards, and security standards
- Clause 5 has been amended to provide general vulnerability issues and primary avoidance mechanisms
- The titles of some clause 6 vulnerabilities have been renamed to better capture the actual vulnerability,
- The clause “Fault tolerance and failure strategies” was moved from clause 6 (6.37) to clause 7 (7.31) to reflect that the vulnerability is more about the system design of fault tolerance and failure recovery strategies than language-oriented.
- A new language vulnerability was added, 6.65 Modifying constants [UJO]

- Application vulnerabilities (clause 7) were reorganized to gather like vulnerabilities together
- New Application vulnerabilities were added to expose issues with time management in realtime systems, in normal systems and in networked systems
- Guidance in clauses 6.x.5 and 7.x.4 were replaced with “avoidance mechanisms” to clarify that it is the responsibility of the implementation team to create design and coding standards, and that some of the avoidance mechanisms stated only apply to specific scenarios.
- A new Annex B “Selected principles for language designers” has been added to collate material from clauses 6.x.6 in a single place.

A list of all parts in the ISO/IEC 24772 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

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 can therefore give rise to vulnerabilities, a result of which software programs can execute differently than intended by the writer. In some cases, these vulnerabilities can endanger the safety of a system or be exploited by attackers to compromise the security or privacy of a system.

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

This document provides users of programming languages with a language-independent overview of potential vulnerabilities in their usage and ways to avoid or mitigate them. Other parts in the ISO/IEC 24772 series describe how the language-independent analysis of this document apply to the specific programming language addressed by that document.

This document is intended to catalogue avoidance mechanisms 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. These mechanisms 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.

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

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

Specific audiences for this document include developers, maintainers and regulators of:

— Safety-critical applications that might cause loss of life, human injury, or damage to the environment.

— Security-critical applications that must ensure properties of confidentiality, integrity, and availability.

— Mission-critical applications that must avoid loss or damage to property or finance.

— Business-critical applications where correct operation is essential to the successful operation of the business.

— Scientific, modelling and simulation applications that require high confidence in the results of possibly complex, expensive and extended calculation.

This document can be relevant to other developers as well. A weakness in a non-critical application may provide the route by which an attacker gains control of a system or otherwise disrupts co-hosted applications that are critical. All developers can use this document to ensure that common vulnerabilities are removed or at least minimized from all applications.

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

While this document does not discuss specification or design issues, there is recognition that boundaries among the various activities are not clear-cut. This document 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.

This document is inherently incomplete, as 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.

Programming Languages —Avoiding vulnerabilities in programming languages – Part 1: Language-independent catalogue of vulnerabilities

1 Scope

This document specifies software programming language vulnerabilities that should be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, the description of the vulnerabilities and description of avoidance mechanisms are 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 documents are referred to in the text in such a way that some or all of their content constitutes requirements 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.

IEC 61508-1, *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 1: General requirements*

IEC 61508-3, *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements*

ISO/IEC 27001, *Information technology -- Security techniques -- Information security management systems -- Requirements*

ISO/IEC 27002, *Information technology -- Security techniques -- Code of practice for information security controls*

ISO/IEC/IEEE 60559, *Information technology -- Microprocessor Systems -- Floating-Point arithmetic*

ISO/IEC 10967-1:2012, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*

ISO/IEC 10967-2:2001, *Information technology — Language independent arithmetic — Part 2: Elementary numerical functions*

3 Terms and definitions

3.1 General

For the purposes of this document, the terms and definitions given in ISO and IEC terminology databases and in this clause apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.2 Communication

3.2.1
protocol
set of rules and supporting structures for the interaction of concurrent entities, such as tightly embedded interactions of threads or loosely coupled arrangements such as message communication spanning computer systems and networks.

3.2.2
stateless protocol
communication or cooperation between threads where no state is preserved in the protocol itself, such as the HTTP protocol or direct access to a shared resource.

3.3 Execution model

3.3.1
thread
sequential stream of execution such as a single thread in a process or a process in an operating system

3.3.2
thread activation
creation and setup of a thread up to the point where the thread begins execution

3.3.3
activated thread
thread that is created and then begins execution as a result of thread activation

3.3.4
activating thread
thread that exists first and makes the library calls or contains the language syntax that causes another thread to be activated, whether or not the activating thread waits for successful activation, checks for correct activation, or awaits termination of the activated thread

3.3.5
static thread activation
creation and initiation of a thread at program initiation, by an operating system or runtime kernel, or by another thread as part of a declarative part of the thread before it begins execution,

3.3.6
dynamic thread activation
creation and initiation of a thread by another thread (including the main program) as an executable, repeatable command, statement or subprogram call

3.3.7
thread abort

request to stop and shut down a thread immediately whether that request comes from an operating system, another thread via the operating system, or a request via shared data or communicating channel to have the thread cease execution

3.3.8
termination directing thread
thread, including an operating system thread, that requests the termination of one or more threads

3.3.9
thread termination
completion and orderly shutdown of a thread, where the thread is permitted to make data objects consistent, release any acquired resources, and notify any dependent threads that it is terminating

3.3.10
terminated thread
thread that has been halted from any further execution

3.3.11
master thread
thread that initiates other threads and that eventually waits for a terminated thread before it can take further execution steps (including termination of itself)

3.3.12
process
single execution of a program, or portion of an application which may execute independently, or which may interact in programmed ways with other processes and which may share resources such as memory, processor and filing system with other processes

3.4 Properties

3.4.1
predictable execution
property of the program such that all possible executions have results that can be predicted from the source code

3.5 Safety

3.5.1
safety hazard
potential source of material or environmental damage, physical injury, or damage to the health of people.

3.5.2
safety-critical
software or applications where failure can cause very serious consequences such as human injury or death

3.6 Vulnerabilities

3.6.1
application vulnerability
security vulnerability or safety hazard or defect

3.6.2
language vulnerability
property or feature of a programming language that through its presence or absence can contribute to, or that is strongly correlated with, application vulnerabilities in programs written in that language.

3.6.3
security vulnerability
weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat

3.7 Specific vulnerabilities

3.7.1
failure
malfunction of the system or component which has as subcategories omission failure, commission failure, timing failure and value failure

3.7.2
omission failure
service that is requested but never rendered

3.7.3
commission failure
service that initiates unexpected actions

3.7.4
timing failure
service that is not rendered before an imposed deadline

3.7.5
value failure
service that delivers incorrect or tainted results

3.7.6
dangling reference
reference to an object whose lifetime has ended due to explicit deallocation or the stack frame in which the object resided has been freed due to exiting the dynamic scope

3.7.7
unspecified functionality
code that may be executed, but whose behaviour does not contribute to the requirements of the application

4 Using this document

4.1 Purpose of this document

This document describes language vulnerabilities and mechanisms to avoid them.

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

This document can be used by the following:

— Programmers familiar with the vulnerabilities of a specific language can reference this document for more generic descriptions and their manifestations in less familiar languages.

— Tool vendors can select from this document vulnerabilities to be addressed by their tools.

— Individual organizations might wish to write their own coding standards intended to reduce the number of vulnerabilities in their software products. This document can assist in the selection of vulnerabilities to be addressed in those coding 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.

— Scientists, engineers, economists, statisticians, or others who write computer programs can read this document to become more familiar with the issues that may affect their work.

— Educators can use the document as a reference for dangerous vulnerabilities in programming and for guidance to avoid or mitigate them.

There are a number of ways to avoid a vulnerability:

— One can avoid the particular coding constructs that are found to be problematic.

— Static analysis tools can be used to detect anomalous situations such as usage of a tool that refuses to pass a harmful construct. For instance, this includes a compiler that provides warnings if a construct is problematic.

— A programming language can be chosen that avoids or mitigates a class of vulnerabilities.

— One can write specific runtime checks to detect situations that may lead to problematic behaviour.

— Automated analysis tools can be used to enforce coding standards.

— Verification and validation methods such as focused human review of code can be undertaken.

This document gathers descriptions of programming language vulnerabilities, as well as selected application vulnerabilities, which have occurred in the past and are likely to occur again. Every vulnerability discussed here has been experienced in at least one programming language or runtime environment. Some vulnerabilities occur in all programming languages, while others are mitigated by the features or capabilities of some programming environments.

Each vulnerability and its possible mitigations are described in this document in a language-independent manner, though illustrative examples may be language specific. In addition, separate language-specific documents ISO/IEC 24772-2. 24772-2, etc. (called Parts) for particular languages describe the vulnerabilities and their mitigations in a manner specific to each language. For example, ISO/IEC 24772-2 describes programming language vulnerabilities for the Ada programming language.

Throughout this document, avoidance mechanisms are described to prevent the vulnerabilities from occurring. These avoidance mechanisms are expressed in imperative terms so that coding guidelines based on this document can use the terminology verbatim. The imperative forms of avoidance mechanisms used herein does not express requirements imposed by this document. In particular, the suggested avoidance mechanisms are occasionally contradictory to each other as they provide alternatives.

Because new vulnerabilities are always being discovered, new descriptions can be needed in future editions to identify the new vulnerability descriptions. For that reason, a scheme of unique, random identifiers was chosen as permanent identification as opposed to subclause numbering which can change between editions. Each description has been assigned an arbitrarily generated, unique three-letter code. Tool vendors can use the three-letter codes as a succinct way to “profile” the selection of vulnerabilities considered by their tools.

4.2 Applying this document

This document is expected to be used in the creation of software that is safe, secure and trusted within the context of the system in which it is fielded. IEC 61508–3: defines Safety-related software as software that is used to implement safety functions in a safety-related system. Notwithstanding that in some domains a distinction is made between safety-related software (can lead to any harm) and safety-critical software (life threatening), this document uses the term safety-critical for all vulnerabilities that can result in safety hazards. Similarly, ISO/IEC 27001 defines security-related systems, and this document uses the term security-critical systems for all vulnerabilities that can result in security hazards.

This document is expected to be used in conjunction with some of the following documents, depending upon the planned application of the software:

- International functional safety standards IEC 61508-1 and IEC 61508-3;
- International security standards ISO/IEC 27001 and ISO/IEC 27002 and application-related ISO/IEC 27000 family of standards;
- National safety or security standards;
- Sector-specific standards such as MISRA C for automotive sector;
- Corporate or organizational standards and directives.

In particular, this document provides answers for questions raised in the construction of:

- Safety-critical applications;
- Security-critical applications;
- Mission-critical / business-critical applications;
- Scientific, modeling and simulation applications that have social impact.

Organizations using this document for system/application development following relevant standards in their safety/security/application domains to

- Determine the criticality of the system, including safety levels, security and privacy,

- Analyze failure modes of the system,
- Identify and analyze external events and how they can affect the system, or
- Identify and analyze attack surfaces of the system.

To use this document effectively, organizations are expected to:

- Identify the programming language(s) to be used in programming the applications in the system.
- Identify and analyze weaknesses in the product or system, including systems, subsystems, modules, and individual components.
- Identify and analyze sources of programming errors.
- Determine acceptable programming paradigms and practices to avoid vulnerabilities using the documentation provided in clause 5.2, 6 and clause 7.
- Map the identified acceptable programming practices into organizational coding standards.
- Select and deploy tooling and processes to enforce coding rules or practices.
- Implement controls (in keeping with the requirements of the safety, security and privacy needs of the system) that enforce these practices and procedures to ensure that the vulnerabilities do not affect the safety and security of the system under development.

In choosing avoidance and mitigation mechanisms, organizations should consult the language-dependent documents of the ISO/IEC 24772 series applicable to their chosen programming language(s), such as ISO/IEC 24772-2 for Ada[17] and ISO/IEC 24772-3 for C[18].

Tool vendors that follow this document provide tools that diagnose the vulnerabilities described here document to their users those vulnerabilities that cannot be diagnosed by the tool.

Programmers and software designers that follow this document adopt the architectural and coding guidelines of their organization and choose appropriate mitigation techniques when a vulnerability is not avoidable.

4.3 Structure of this document

The rest of the document is organized as follows:

Clause 5 *Vulnerability issues*, explains how many of the vulnerabilities common to programming languages occur. The issues discussed are not vulnerabilities but are language characteristics that can lead to mistakes and to vulnerabilities that can be exploited. Subclause 5.2 provides a summary list of the top 20 approaches to avoiding the most common vulnerabilities in a tabular form with references to the applicable more detailed descriptions provided in clauses 6 and 7. For many that cannot invest the resources to research all of the vulnerabilities documented in clauses 6, 7, and 8, implementing the documented mechanisms in 5.2 will already provide significant benefit to their projects.

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. In using clause 6, it is important to be aware of how a listed vulnerability is presented by the programming language, tool environment, and operating system that is being used. To help, this document is supported by a set of Standards or Technical Reports (Parts), i.e. 24772-2 (for Ada[17]), 24772-3 (for C[18]), etc. Each additional Part:

- is named for a particular programming language,
- lists the vulnerabilities described in clause 6,
- describes how each vulnerability appears (or does not appear) in that specific language, and
- specifies how it may be mitigated in that language, whenever possible.

Clause 7 *Application vulnerabilities*, provides descriptions of selected vulnerabilities, generally unrelated to programming language features, 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.

Mitigations for vulnerabilities listed in clause 7 will not include the use of programming language-specific features or choices but will consist of alternate design choices or programming techniques.

Annex A is a categorization of the vulnerabilities of this document by general topic areas.

Annex B summarizes information for language designers in 6.X.6.

Annex C, provides a template for the writing of programming language specific Parts 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.

Throughout this document the font courier is used for tokens typically present in programming languages, such as `false` and `true`, but also for representative program samples from actual programming languages.

5 General vulnerability issues and primary avoidance mechanisms

5.1 General vulnerability issues

5.1.1 Predictable execution

There are many reasons why software might not execute as expected by its developers, its users or other stakeholders. Reasons include incorrect specifications, configuration management errors and a myriad of others. This document focuses on the usage of programming languages in ways that render the execution of the code less predictable, or the usage of design paradigms that weaken the application and make it susceptible to attack.

Achieving predictable execution is complicated by that fact that software may be used

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

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

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

This document focuses on a particular class of vulnerabilities, *language vulnerabilities*. These are properties of programming languages that can contribute to (or are strongly correlated with) *application vulnerabilities*—security weaknesses, safety hazards, or defects.

An example may clarify the relationship. The programmer's use of a string copying function that does not check length may be exploited by an attacker to place incorrect return values on the program stack, hence passing control of the execution to code provided by the attacker. The string copying function is the language vulnerability and the resulting weakness of the program in the face of the stack attack is the application vulnerability. The programming language vulnerability enables the application vulnerability. The language vulnerability can be avoided by using a string copying function that sets and enforces appropriate bounds on the length of the string to be copied. By using a bounded copy function the programmer improves the predictability of the code's execution.

The primary purpose of this document is to survey common programming language vulnerabilities; this is done in clause 6. Each description explains how an application vulnerability can result.

In clause 7, additional application vulnerabilities are described. These are selected because they are associated with language weaknesses even if they do not directly result from language vulnerabilities. For example, a programmer might have stored a password in plain text (see 7.17 *Insufficiently protected stored credentials [XYM]*) because the programming language did not provide a suitable library function for storing the password in a non-recoverable format.

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

5.1.2 Sources of unpredictability in language specification

5.1.2.1 Incomplete or evolving specification

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

5.1.2.2 Undefined behaviour

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

5.1.2.3 Unspecified behaviour

The behaviour of some features may be incompletely defined. The language implementer would have to choose from a finite set of choices, but the choice may not be apparent to the programmer. In such cases, different compilers or the same compiler with different options processing the code selected may lead to different results. Some of the possible results may be harmful.

5.1.2.4 Implementation-defined behaviour

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

5.1.2.5 Difficult features

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

5.1.2.6 Inadequate language support

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

5.1.3 Sources of unpredictability in language usage

5.1.3.1 Porting and interoperation

The behaviour of a program can change when it is recompiled using a different compiler, recompiled using different compilation options, executed with different libraries, executed on a different platform, or even interfaced with different systems. Such changes result from different choices for unspecified and implementation-defined behaviour, differences in library function, and differences in underlying hardware and operating system support. The problem is far worse if the original programmer chose to use implementation-dependent extensions to the language rather than staying with the standardized language.

5.1.3.2 Compiler selection and usage

Nearly all software has defects and compilers are no exception. Therefore, the compiler should be carefully selected from trusted sources and qualified prior to use. Perhaps less obvious, though, is the use of compiler options. Different compiler options can cause differences in generated code. A careful selection of settings can improve the predictability of code, such as a setting that causes the flagging of any usage of an implementation-defined behaviour.

5.2 Primary avoidance mechanisms

Each vulnerability listed in clauses 6 and 7 provides a set of ways that the vulnerability can be avoided or mitigated. Many of the mitigations and avoidance mechanisms are common. This subclause documents the most effective and most common mitigations, together with references to which vulnerabilities they apply.

Number	Software developers can …	References
1	Validate input, not make assumptions about the values of parameters and check parameters for valid ranges and values in the calling and/or called functions before performing any operations.	6.6[FLC] 7.13[XZP] 7.18[XZN] 7.28[CCM]
2	When functions return error values, check the error return values before processing any other returned data.	6.36[OYB] 6.60[CGT]
3	Enable compiler static analysis checking and resolve compiler warnings.	6.8 [HBC] 6.10[XYW] 6.14[XYK] 6.15 [FIF] 6.16[PIK] 6.17[NIA] 6.18[WXQ] 6.19[YZS] 6.22[LAV] 6.25[KOA] 6.26[XYQ] 6.27[CLL] 6.29[TEX] 6.30 [XZH] 6.34[QTR] 6.36[OYB] 6.38[YAN] 6.39[XYL] 6.47[DJS] 6.54[BRS] 6.56[EWF] 6.57[FAB] 6.60[CGT] 6.61[CGX] 6.62[CGS] 7.28[CCM]
4	Run a static analysis tool to detect anomalies not caught by the compiler.	6.3[STR] 6.6[FLC] 6.7[CJM] 6.8[HBC] 6.10[XYW] 6.14[XYK] 6.15[FIF] 6.16[PIK] 6.17[NIA] 6.18[WXQ] 6.19[YZS] 6.22[LAV] 6.25[KOA] 6.26[XYQ] 6.27[CLL] 6.29[TEX] 6.30 [XZH] 6.34[QTR] 6.36[OYB] 6.38[YAN] 6.39[XYL] 6.47[DJS] 6.54[BRS] 6.56[EWF] 6.57[FAB] 6.60[CGT] 6.61[CGX] 6.62[CGS] 7.28[CCM]
5	Perform explicit range checking: when it cannot be shown statically that ranges will be obeyed; when range checking is not provided by the implementation; or if automatic range checking is disabled.	6.6[FLC] 6.8[HBC] 6.16[PIK]
6	Allocate and free resources, such as memory, threads or locks, at the same level of abstraction.	6.14[XYK]
7	Avoid constructs that have unspecified but bounded behaviour, and if the construct is needed, test for all possible behaviours.	6.24[XYK] 6.56[EWF]
8	Make error detection, error reporting, error correction, and recovery an integral part of a system design.	6.36[OYB]
9	Use only those features of the programming language that enforce a logical structure on the program.	6.31[EWD]
10	Avoid using features of the language which are not specified to an exact behaviour or that are undefined, implementation-defined or deprecated.	6.55[BQF] 6.56[EWF] 6.57[FAB] 6.58[MEM]

		6.59[CGA]
11	Avoid using libraries without proper signatures.	6.34[QTR]
12	Prohibit the modification of loop control variables inside the loop body.	6.29[TEX]
13	Prohibit assignments within Boolean expressions, even if allowed by the language.	6.25[KOA]
14	Prohibit dependence on side effects of a term in the expression itself.	6.31[EWD] 6.24[SAM]
15	Use names that are clear and visually unambiguous and be consistent in choosing names.	6.17[NIA]
16	Use careful programming practice when programming border cases.	6.6[FLC] 6.29[TEX] 6.30 [XZH]
17	Beware of short-circuiting behaviour when expressions with side effects are used on the right side of a short-circuited Boolean expression, since a left-hand expression evaluating to `false`, dictates that the right-hand expression, including function calls with side effects, will not be evaluated.	6.24[SAM] 6.25[KOA]
18	Avoid fall-through from one case (or switch) statement into the following case statement: if a fall-through is necessary then provide a comment to inform the reader that it is intentional.	6.27[CLL]
19	Avoid using floating-point arithmetic when integers would suffice, especially for counters associated with program flow, such as loop control variables.	6.4[PLF]
20	Sanitize, erase, or encrypt data that will be visible to others (for example, freed memory, transmitted data).	7.11[EWR] 7.12[HTS]

6 Programming language vulnerabilities

6.1 General

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

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

Descriptions of how vulnerabilities are manifested in particular programming languages are provided in a separate Part in the ISO 24772 series. In each case, the behaviour of the language is assumed to be as specified by the language (International or non-International) standard cited in the respective Part. Clearly, programs could have different vulnerabilities in a non-standard implementation. Examples of non-standard implementations include

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

The vulnerability descriptions in this document are written in a language-independent manner except when specific languages are used in examples. Language-specific Parts may be consulted for language specific descriptions. This document is also supported by coding guidelines or standards for specific languages. Where applicable, general references to such documents and explicit references to coding guidelines are provided in subclause X.2 of this clause and in clause 7.

In general, clause 6 will use the terminology that is most natural to the description of each individual vulnerability. Hence, terminology may differ from description to description.

6.2 Type system [IHN]

6.2.1 Description of application vulnerability

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

6.2.2 Related coding guidelines

JSF AV Rules [24]: 148 and 183

MISRA C [29]: 4.6, 10.1, 10.3, and 10.4

MISRA C++ [30]: 3-9-2, 5-0-3 to 5-0-14

CERT C++ Secure Coding Standard [6]

CERT C guidelines [31]: DCL07-C, DCL11-C, DCL35-C, EXP05-C and EXP32-C

Ada Quality and Style Guide [1]: 3.4

6.2.3 Mechanism of failure

The type of a data object informs the compiler how values are 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.

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 runtime checks.

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 type. 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 is converted to the float type before it is added to the data value. This is an implicit type conversion. If, on the other hand, the conversion must be specified by the program, for example,

```
         a := a + float(i)
```

then it is an explicit type conversion.

Type equivalence is the strictest form of type compatibility; two types are equivalent if they are compatible without using implicit or explicit conversion. 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, such as the C bounds-checking interface [22]. Therefore, a programmer skilled in one language may very well code inadvertent type errors when using a different language.

Programs should 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 increases as a power of the problem size. (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 [17] fragment declares two distinct floating-point types:

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

The declarations make it impossible to add a value of type Celsius to a value of type Fahrenheit without explicit conversion. Even explicit conversions also require additional numeric calculations that respect the relationship of the real-world units being converted. For example, `F = CC` (where F is Fahrenheit and CC is Celsius) only works in the special case when `CC = -40`, otherwise one needs

```
    F = convert_to_fahrenheit(CC)
```

which performs `9*C/5+32`.

As another example, the following Pascal code

```
      type AltitudeInFeet = -1500.. 45000;
```

defines the operating range of a plane and lets the compiler decide on the appropriate underlying representation in contrast to a predefined type `integer` which might be represented in 16 bits (insufficient for the purpose) or 32 bits, depending on the target architecture.

6.2.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages that support multiple types and allow conversions between types.

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. They can

- 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 tool capabilities to preclude or detect the occurrence of implicit type conversions, such as those in mixed type arithmetic. If this is not possible, human review can assist in searching for implicit conversions.
- Avoid explicit type conversion of data values except when there is no alternative. Documenting such occurrences makes the justification available to maintainers.
- Use the most restricted data type that suffices to accomplish the job; for example, using 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, enables tooling to check if all possible choices have been covered.
- Respect the implied unit systems, when converting explicitly from one numeric type to another.
- Treat every compiler, tool, or run-time diagnostic concerning type compatibility as a serious issue and avoid resolution of the issue by modifying the code to include an explicit conversion, without further analysis; instead software developers should examine the underlying design to determine if the type error is a symptom of a deeper problem.
- Never ignore instances of implicit type conversion; since if the conversion is necessary, it should be changed to an explicit conversion and the rationale should be document 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.
- Create types that more accurately model the problem domain, with corresponding safe operations and conversions in lieu of using primitive types.
- Minimize the use of predefined numeric types whose ranges and precisions are implementation defined; instead, using types whose ranges and precision are guaranteed.

6.2.6 Implications for language design and evolution

In future language design and evolution activities, software designers should consider the following items:

- Standardizing 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;

- Providing a mechanism for selecting data types with sufficient capability for the problem at hand;
- Providing a way for the computation to determine the limits of the data types actually selected;
- Providing compiler options or other mechanisms to provide the highest possible degree of checking for type errors.

6.3 Bit representations [STR]

6.3.1 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 (whether the highest order bit is called bit 0 or bit n) 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.

6.3.2 Related coding guidelines

JSF AV Rules [24] 147, 154 and 155

MISRA C [29]: 1.1, 6.1, 6.2, and 10.1

MISRA C++ [30]: 5-0-21, 5-2-4 to 5-2-9, and 9-5-1

CERT C++ guidelines [6]: EXP38-C, INT00-C, INT07-C, INT12-C, INT13-C, and INT14-C

Ada Quality and Style Guide [1]: 7.6.1 to 7.6.9, and 7.3.1

See also Hogaboom [13]

6.3.3 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 $28=2^{16}+2^{8}+2^{4}$ 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 (e.g. arithmetic and logical operations) to reference the bits or output the bit, since storage ordering of the bits may not be what the programmer expects.

For the C programming language [18], Hogaboom [13] discusses generic bit manipulation in C. The C++ programming language [19] also shares many of C's characteristics but also provides higher level constructs that help the programmer avoid associated vulnerabilities.

Packing of bits in an integer is not inherently problematic. However, an understanding of the intricacies of bit level programming is a must. Some computers or other devices number the bits smallest-to-largest while others number them largest-to-smallest.

> Note Some programmers think of this as left-to-right and right-to-left. Common terminology discusses shifting bits left-to-right or right-to-left where the sign bit (if present) is considered to be the left-most bit.

Storage organization can cause problems when interfacing with external devices that number the bits in opposite order. One problem arises when incorrect assumptions are made when interfacing with external data sources or sinks and the ordering of the bits or words are not the same on both sides. Programmers may inadvertently use the sign bit in a bit field and then may not be aware that an arithmetic shift (sign extension) is being performed when right shifting causing the sign bit to be extended into other fields. Alternatively, a left shift can cause the sign bit to be one. Bit manipulations can also be problematic when the manipulations are done on binary encoded records that span multiple words. Knowledge of the storage and ordering of the bits is essential when doing bit-wise operations across multiple words as bytes may be stored in big-endian or little-endian format.

6.3.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that allow bit manipulations.

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: They can

— Explicitly document any reliance on bit ordering such as explicit bit patterns, shifts, or bit numbers;

— Understand the way bit ordering is done on the host system and on the systems with which the bit manipulations will be interfaced;

— Where supported by the language, use bit fields in preference to binary, octal, or hexadecimal representations;

— Avoid bit operations on signed operands;

— Localize and document code associated with explicit manipulation of bits and bit fields;

— Use static analysis tools that identify and report reliance upon bit ordering or bit representation.

6.3.6 Implications for language design and evolution

In future language design and evolution activities, for languages that are commonly used for bit manipulations, consider creating a standardized *API* (Application Programming Interface) for bit manipulations that is independent of word size and machine instruction set.

6.4 Floating-point arithmetic [PLF]

6.4.1 Description of application vulnerability

Most real numbers cannot be represented exactly in a computer. To represent real numbers, most computers use ISO/IEC/IEEE 60559. If ISO/IEC/IEEE 60559 is not followed, then the bit representation for a floating-point number can vary from compiler to compiler and on different platforms, however, relying on a particular representation can cause problems when a different compiler is used or the code is reused on another platform. Regardless of the representation, many real numbers can only be approximated since representing the real number using a binary representation may well require an endlessly repeating string of bits or more binary digits than are available for representation. A floating-point number is only an approximation, even though it may be an extremely good one. Floating-point representation of a real number or a conversion to floating-point can cause surprising results and unexpected consequences to those unaccustomed to the idiosyncrasies of floating-point arithmetic.

Many algorithms that use floating point can have anomalous behaviour when used with certain values. The most common results are erroneous results or algorithms that never terminate for certain segments of the numeric domain, or for isolated values. Those without training or experience in numerical analysis may not be aware of which algorithms, or, for a particular algorithm, of which domain values need attention.

In some hardware, precision for intermediate floating-point calculations may be different than that suggested by the data type, causing different rounding results when moving to standard precision modes.

6.4.2 Related coding guidelines

JSF AV Rules[24]: 146, 147, 184, 197, and 202

MISRA C [29]: 1.1 and 14.1

MISRA C++ [30]: 0-4-3, 3-9-3, and 6-2-2

CERT C guidelines [31]: FLP00-C, FP01-C, FLP02-C and FLP30-C

Ada Quality and Style Guide[1]: 5.5.6 and 7.2.1 through 7.2.8

6.4.3 Mechanism of failure

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

`0.0001100110011001100110011001100110011001100110011`...

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

Accumulating floating point values through the repeated addition of values, particularly relatively small values, can provide unexpected results. Using an accumulated value to terminate a loop can result in an unexpected number of iterations. Rounding and truncation can cause tests of floating-point numbers against other values to yield unexpected results. Another cause of floating-point errors is reliance upon comparisons of floating point values or the comparison of a floating point value with zero. Tests of equality or inequality can vary due to rounding or truncation errors, which may propagate far from the operation of origin. Even comparisons of constants may fail when a different rounding mode was employed by the compiler and by the application. Differences in magnitudes of floating-point numbers can result in no change of a very large floating-point number when a relatively small number is added to or subtracted from it.

Manipulating bits in floating-point numbers is also very implementation dependent if the implementation is not ISO/IEC/IEEE 60559 compliant or in the interpretation of `NAN`'s. Typically, special representations are specified for positive zero and negative zero; infinity and subnormal numbers very close to zero. Relying on a particular bit representation is inherently problematic, especially when a new compiler is introduced or the code is reused on another platform. The uncertainties arising from floating-point can be divided into uncertainty about the actual bit representation of a given value (such as, big-endian or little-endian) and the uncertainty arising from the rounding of arithmetic operations (for example, the accumulation of errors when imprecise floating-point values are used as loop indices).

Most floating-point implementations are binary. Decimal floating-point numbers are available on some hardware and that capability has been standardized in ISO/IEC/IEEE 60559:2011 but one should aware what precision guarantees the implementation programming language makes. In general, fixed-point arithmetic may be a better solution to common problems involving decimal fractions (such as financial calculations).

Implementations (libraries) for different precisions are often implemented in the highest precision. This can yield different results in algorithms such as exponentiation than if the programmer had performed the calculation directly.

Floating-point systems have more than one rounding mode. Round to the nearest even number is the default for almost all implementations. Repeatedly rounding iterative calculations towards zero or away from zero can result in a loss of precision and can cause unexpected outcome.

Some floating-point functions can return an arbitrary sign when the result is exactly zero. Tests that use the sign of a number rather than its relationship to zero can return unexpected results.

See also Goldberg [10].

6.4.4 Applicable language characteristics

This vulnerability description is intended to be applicable to all languages with floating-point operations, since floating-point variables can be subject to rounding or truncation errors.

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. They can

— Unless the program's use of floating-point is trivial, obtain the assistance of an expert in numerical analysis and in the hardware properties of the target system to check the stability and accuracy of the algorithm employed.

— Avoid the use of floating-point expressions in a Boolean test for equality unless it can be shown that the logic implemented by the equality test cannot be affected by prior rounding errors. Instead, use coding that determines the difference between the two values to determine whether the difference is acceptably small enough so that two values can be considered equal. If the two values are very large, the "small enough" difference can be a very large number.

— Verify that the underlying implementation is compliant with ISO/IEC/IEEE 60559 or that it includes subnormal numbers (fixed point numbers that are close to zero); and be aware that implementations that do not have this capability can underflow to zero in unexpected situations.

— Be aware that infinities, NAN and subnormal numbers may be possible and give special consideration to tests that check for those conditions before using them in floating point calculations.

— Use library functions with known numerical characteristics.

— Avoid the use of a floating-point variable as a loop counter, but if it is necessary to use a floating-point value for loop control, use inequality to determine the loop control (that is, <, <=, > or >=).

— Understand the floating-point format used to represent the floating-point numbers to provide some understanding of the underlying idiosyncrasies of floating-point arithmetic.

— Avoid manipulating the bit representation of a floating-point number; instead prefer built-in language operators and functions that are designed to extract the mantissa, exponent or sign.

— Avoid the use of floating-point for exact values such as monetary amount, and instead use floating-point only when necessary, such as for fundamentally inexact values such as measurements or values of diverse magnitudes.

— Consider the use of fixed-point arithmetic /libraries or decimal floating point when appropriate.

— Use known precision modes to implement algorithms.

— Avoid changing the rounding mode from RNE (round nearest even).

- Avoid reliance on the sign of the floating-point `Min` and `Max` operations when both numbers are zero.
- When adding (or subtracting) sequences of numbers, sort and add (or subtract) them from smallest to largest in absolute value, or use a suitable compensated summation algorithm to avoid loss of precision.

6.4.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- If a language does not already adhere to or only adheres to a subset of ISO/IEC/IEEE 60559], adhere completely to ISO/IEC/IEEE 60559;
- Providing a means to generate diagnostics for code that attempts to test equality of two floating point values;
- Standardizing their data type to ISO/IEC 10967-1:2012 and ISO/IEC 10967-2:2001.

6.5 Enumerator issues [CCB]

6.5.1 Description of application vulnerability

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

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

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

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

6.5.2 Related coding guidelines

MISRA C [29]: 8.12, 9.2, and 9.3

MISRA C++ [30]: 8-5-3

CERT C guidelines [31]: INT09-C

Ada Quality and Style Guide [1]: 3.4.2

See also Holzmann [14] rule 6.

6.5.3 Mechanism of failure

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

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

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

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

6.5.4 Applicable language Characteristics

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

- Languages that permit incomplete mappings between enumerator specification and value assignment, or that provide a positional-only mapping require additional static analysis tools and annotations to help identify the complete mapping of every literal to its value.

- Languages that provide a trivial mapping to a type such as integer require additional static analysis tools to prevent mixed type errors. They also cannot prevent invalid values from being placed into variables of such enumerator types. For example:

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

In this example, `c` may have a value not defined by the enumeration, and any further use as that enumeration will lead to erroneous results.

- Some languages provide no enumeration capability, leaving it to the programmer to define named constants to represent the values and ranges.

6.5.5 Avoiding the vulnerability or mitigating its effects

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

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

- In code that performs different computations depending on the value of an enumeration, ensure that each possible enumeration value is covered, or provide a default that raises an error or exception.

- Use an enumerated type to select from a limited set of choices and use tools that statically detect omissions of possible values in an enumeration. For languages with a complete enumeration abstraction, this is enforced by the compiler.

6.5.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- For languages that currently permit arithmetic and logical operations on enumeration types, providing a mechanism to ban such operations program-wide;

— For languages that provide automatic defaults or that do not enforce static matching between enumerator definitions and initialization expressions, providing a mechanism to enforce such matching.

6.6 Conversion errors [FLC]

6.6.1 Description of application vulnerability

Certain contexts in various languages may require exact matches with respect to types.

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

Type conversion seeks to follow these exact match rules while allowing programmers some flexibility in using values such as: structurally equivalent types in a name-equivalent language, types whose value ranges may be distinct but intersect (for example, subranges), and distinct types with sensible/meaningful corresponding values (for example, integers and floats).

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

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

See also 6.44 on polymorphic variables [BKK] for up-casting errors.

6.6.2 Related coding guidelines

CWE [7]: 192. Integer Coercion Error

MISRA C [29]: 7.2, 10.1, 10.3, 10.4, 10.6-10.8, and 11.1-11.8

MISRA C++ [30]: 2-13-3, 5-0-3, 5-0-4, 5-0-5, 5-0-6, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-2-5, 5-2-9, and 5-3-2

CERT C guidelines [31]: FLP34-C, INT02-C, INT08-C, INT31-C, and INT35-C

6.6.3 Mechanism of failure

Conversion errors result in data integrity issues and may also result in a number of safety and security vulnerabilities.

When the conversion results in no change in representation but a change in value for the new type, this may result in a value that is not expressible in the new type, or that has a dramatically different order or meaning. One such situation is the change of sign between the origin and destination (negative -> positive or positive -> negative), which changes the relative order of members of the two types and could result in memory access failures if the values are used in address calculations. Numeric type conversions can be less obvious because some languages will silently convert between numeric types.

Vulnerabilities typically occur when appropriate range checking is not performed, and unanticipated values are encountered. An Ariane 5 [2][27] launcher failure occurred due to an improperly handled conversion error resulting in the processor being shut down.

Conversion errors can also result in security issues. An attacker may input a particular numeric value to exploit a flaw in the program logic. The resulting erroneous value may then be used as an array index, a loop iterator, a length, a size, state data, or in some other security-critical manner. For example, a truncated integer value may be used to allocate memory, while the actual length is used to copy information to the newly allocated memory, resulting in a buffer overflow, as specified in ISO/IEC 60559.

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

Explicit conversions between entities of different unit systems without the application of the correct conversion factors can lead to incorrect computations. For example, the first Martian lander failed due to an improper conversion from meters to feet resulting in the loss of the lander.

6.6.4 Applicable language characteristics

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

- Languages that perform implicit type conversion (coercion);
- Languages that permit conversions between subtypes of a polymorphic type. See 6.44;
- Weakly typed languages that do not strictly enforce typing rules;
- Languages that support logical, arithmetic, or circular shifts on integer values;
- Languages that do not generate exceptions on problematic conversions;

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. They can

- If range checking is not provided by the language, use explicit range checks, type checks or value checks to validate the correctness of all values originating from a source that is not trusted. However, it is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program, see Jones, Derrick [23].
- Use explicit range checks to protect each operation. Because of the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, this approach can be prohibitively labour intensive and expensive to implement.
- Choose a language that generates exceptions on erroneous data conversions.
- Design objects and program flow such that multiple or complex explicit type conversions are unnecessary.
- Understand any explicit type conversion that must be used to reduce the plausibility of error in use.
- Use static analysis tools to identify whether or not unacceptable conversions will occur, to the extent possible.
- Avoid the use of *plausible but wrong* default values when a calculation cannot be completed correctly; instead either generating an error or producing a value that is out of range and is certain to be detected.
- Take care that any error processing does not lead to a denial-of-service vulnerability.
- Always respect the implied unit systems, when converting explicitly from one numeric type to another.

6.6.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Providing mechanisms to prevent programming errors due to conversions;
- Making all type-conversions explicit or at least generating warnings for implicit conversions where loss of data might occur.

6.7 String termination [CJM]

6.7.1 Description of application vulnerability

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

6.7.2 Related coding guidelines

CWE [7]: 170. Improper Null Termination

CERT C guidelines [31]: STR03-C, STR31-C, STR32-C, and STR36-C

6.7.3 Mechanism of failure

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

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

6.7.4 Applicable language characteristics

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

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

6.7.5 Avoiding the vulnerability or mitigating its effects

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

- Avoid relying solely on the string termination character;
- Use library calls that do not rely on string termination characters such as

  ```
  strncpy
  ```

  instead of

  ```
  strcpy
  ```

  in the standard C library;

- Use static analysis tools that detect errors in string termination.

6.7.6 Implications for language design and evolution

In future language design and evolution activities, consider the following items:

- Eliminate library calls that make assumptions about string termination characters;
- Check bounds when an array or string is accessed, such as the C Bounds Checking Library [22];
- Specify a string construct that does not need a string termination character.

6.8 Buffer boundary violation (buffer overflow) [HCB]

6.8.1 Description of application vulnerability

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

6.8.2 Related coding guidelines

CWE [7]:

120. Buffer copy without Checking Size of Input (‘Classic Buffer Overflow’)

122. Heap-based Buffer Overflow

124. Boundary Beginning Violation (‘Buffer Underwrite’)

129. Unchecked Array Indexing

131. Incorrect Calculation of Buffer Size

787. Out-of-bounds Write

805. Buffer Access with Incorrect Length Value

JSF AV [31} : Rule 15 and 25

MISRA C [29]: 21.1

MISRA C++ [30]: 5-0-15 to 5-0-18

CERT C guidelines [31]: ARR30-C, ARR32-C, ARR33-C, ARR38-C, MEM35-C and STR31-C

6.8.3 Mechanism of failure

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

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

- A read access will return a value that has no relationship to the intended value, such as, the value of another variable or uninitialized storage.

- An out-of-bounds read access may be used to obtain information that is intended to be confidential.

- A write access will not result in the intended value being updated and may result in the value of an unrelated object (that happens to exist at the given storage location) being modified, including the possibility of changes in external devices resulting from the memory location being hardware-mapped.

- When an array has been allocated storage on the stack, an out-of-bounds write access may modify internal runtime housekeeping information (for example, a function's return address) which might change a program’s control flow.

- An inadvertent or malicious overwrite of function pointers that may be in memory, causing them to point to an unexpected location or the attacker's code. Even in applications that do not explicitly use function pointers, the run-time will usually store pointers to functions in memory. For example, object methods in object-oriented languages are generally implemented using function pointers in a data structure or structures that are kept in memory. The consequence of a buffer boundary violation can be targeted to cause arbitrary code execution; this vulnerability may be used to subvert any security service.

6.8.4 Applicable language characteristics

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

- Languages that do not detect and prevent an array being accessed outside of its declared bounds, by means of an index, by pointer, or by using the physical memory address to access memory locations.

- Languages that do not automatically allocate storage when accessing an array element for which storage has not already been allocated;

- Languages that provide bounds checking but permit the check to be suppressed;

- Languages that allow a copy or move operation without an automatic length check ensuring that source and target locations are of at least the same size. The destination target can be larger than the source being copied.

6.8.5 Avoiding the vulnerability or mitigating its effects

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

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

- Use static analysis to verify that all array accesses are within the permitted bounds. Such analysis may require that source code contain certain kinds of information, for example, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified.

- Perform sanity checks on all calculated expressions used as an array index or for pointer arithmetic.

- Ascertain whether the compiler can insert bounds checks while still meeting the performance requirements of the program and direct the compiler to insert such checks where appropriate.

NOTE 1 Some guideline documents recommend only using variables having an unsigned data type when indexing an array, on the basis that an unsigned data type can never be negative. This recommendation simply converts an indexing underflow to an indexing overflow because the value of the variable will wrap to a large positive value rather than a negative one. Also, some languages support arrays whose lower bound is greater than zero, so an index can be positive and be less than the lower bound. Some languages support zero-sized arrays, so any reference to a location within such an array is invalid.

NOTE 2 In the past, the implementation of array bound checking has sometimes incurred what has been considered to be a high runtime overhead (often because unnecessary checks were performed). It is now practical for translators to perform sophisticated analysis that significantly reduces the runtime overhead (because runtime checks are only made when it cannot be shown statically that no bound violations can occur).

6.8.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

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

6.9 Unchecked array indexing [XYZ]

6.9.1 Description of application vulnerability

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

6.9.2 Related coding guidelines

CWE [7]:

129. Unchecked Array Indexing

676. Use of Potentially Dangerous Function

JSF AV Rules [24]: 164 and 15

MISRA C [29]: 21.1

MISRA C++ [30]: 5-0-15 to 5-0-18

CERT C guidelines [31]: ARR30-C, ARR32-C, ARR33-C, and ARR38-C

Ada Quality and Style Guide [1]: 5.5.1, 5.5.2, 7.6.7, and 7.6.8

6.9.3 Mechanism of failure

A single fault could allow both an overflow and underflow of the array index. An index overflow exploit might use buffer overflow techniques, but this can often be exploited without having to provide "large inputs." Array index overflows can also trigger out-of-bounds read operations, or operations on the wrong objects; that is, buffer overflows are not always the result. Unchecked array indexing, depending on its instantiation, can be responsible for any number of related issues. Most prominent of these possible flaws is the buffer overflow condition, with consequences ranging from denial of service, and data corruption, to arbitrary code execution.

The most common situation leading to unchecked array indexing is the use of loop index variables as buffer indexes. If the end condition for the loop is subject to a flaw, the index can grow or shrink unbounded, therefore causing a buffer overflow or underflow. Another common situation leading to this condition is the use of a function's return value, or the resulting value of a calculation directly as an index into a buffer. Unchecked array indexing can result in the corruption of relevant memory and perhaps instructions, lead to the program halting, if the values are outside of the valid memory area. If the memory corrupted is data, rather than instructions, the system might continue to function with improper values. If the corrupted memory can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow.

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

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

6.9.4 Applicable language characteristics

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

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

6.9.5 Avoiding the vulnerability or mitigating its effects

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

- Include sanity checks to ensure the validity of any values used as index variables;
- Consider choosing a language that is not susceptible to these issues;
- When available, use whole array operations whenever possible;
- Do not suppress bounds checks provided by the language unless it has been statically verified that out-of-bounds accesses will not occur.

6.9.6 Implications for language designers

In future language design and evolution activities, language designers should consider the following items:

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

6.10 Unchecked array copying [XYW]

6.10.1 Description of application vulnerability

When the size and addresses of both the source and destination of an array or compound object are not checked before the copy operation begins, the results can be catastrophic to program integrity. The classic case of buffer overflow happens when some number of bytes (or other units of storage) are copied from one buffer to another and the amount being copied is greater than is allocated for the destination buffer. Data corruption can happen when the program, or the programmer, does not check for overlap between the source and target.

The first situation, overflow of a buffer in a sensitive region of a system, has been exploited as a classic attack vector to render systems inoperable or to take them over.

The second situation, that of overlap, can result in data corruption, which is likely to result in incorrect functioning of the system with potentially disastrous consequences to the containing system.

6.10.2 Related coding guidelines

CWE [7]: 121. Stack-based Buffer Overflow

JSF AV Rules [24]: 15

MISRA C [29]: 21.1

MISRA C++ [30]: 5-0-15 to 5-0-18

CERT C guidelines [31]: ARR33-C and STR31-C

Ada Quality and Style Guide [1]: 7.6.7 and 7.6.8

6.10.3 Mechanism of failure

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

When the source and target areas overlap, some libraries do not produce the expected outcome of copying the value of the source area into the target area, because they do not identify the situation and save into a temporary first to isolate the overlapped ranges.

The arguments to these library functions include the addresses of the contents of the two storage areas and the number of bytes (or some other measure) to copy. Passing the appropriate combination of incorrect start addresses or number of bytes to copy makes it possible to read or write outside of the storage allocated to the source/destination area. When passed incorrect parameters the library function performs one or more unchecked array index accesses, as described in *6.9 Unchecked array indexing* [XYZ] .

6.10.4 Applicable language characteristics

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

- Languages that contain standard library functions for performing bulk copying of storage areas;
- The same range of languages having the characteristics listed in *6.9 Unchecked array indexing [XYZ]* .

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. They can:

- Only use library functions that perform checks on the arguments to ensure no buffer overrun can occur and perform checks on the argument expressions prior to calling the Standard library function to ensure that no buffer overrun will occur;
- Use static analysis to verify that the appropriate library functions are only called with arguments that do not result in a buffer overrun or overlap;

  NOTE Such analysis may require that source code contain certain kinds of information, for example, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified as annotations or language constructs.

- Sanitize all input data so that excessively large input data that could result in overflows is rejected;
- Avoid suppressing any bounds checks provided by the language.

6.10.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Providing libraries that perform checks on the parameters to ensure that no buffer overrun can occur;
- Providing full array assignment.

6.11 Pointer type conversions [HFC]

6.11.1 Description of application vulnerability

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

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

6.11.2 Related coding guidelines

CWE [7]:

136. Type Errors

188. Reliance on Data/Memory Layout

JSF AV Rules [24]: 182 and 183

MISRA C [29]: 11.1-11.8

MISRA C++ [30]: 5-2-2 to 5-2-9

CERT C guidelines [31]: INT11-C and EXP36-A

Ada Quality and Style Guide [1]: 7.6.7 and 7.6.8

See also Hatton [11] rule 13: Pointer casts

6.11.3 Mechanism of failure

If a pointer’s type is not appropriate for the data or function being accessed, data can be corrupted, or privacy can be broken by inappropriate read or write operation using the indirection provided by the pointer value.

With a suitable type-definition, large portions of memory can be maliciously or accidentally read or modified. Such modification of data objects will generally lead to value faults of the application. Modification of code elements such as function pointers or internal data structures for the support of object-orientation can affect control flow. This can make the code susceptible to targeted attacks by causing invocation via a pointer-to-function that has been manipulated to point to an attacker's malicious code.

6.11.4 Applicable language characteristics

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

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

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. They can:

- Treat all compiler pointer-conversion warnings as serious errors;
- Adopt programming guidelines, preferably augmented by static analysis, that restrict pointer conversions, such as the rules itemized above from JSF AV [24], CERT [31], Hatton [11] or MISRA C [29];
- Use other means of assurance such as proofs of correctness, analysis with tools, verification techniques, or other methods to verify that pointer conversions do not lead to later undefined behaviour.

6.11.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider creating a mode that provides a runtime check of the validity of all accessed objects before the object is read, written or executed.

6.12 Pointer arithmetic [RVG]

6.12.1 Description of application vulnerability

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

6.12.2 Related coding guidelines

JSF AV [24] Rule: 215

MISRA C [29]: 18.1-18.4

MISRA C++ [30]: 5-0-15 to 5-0-18

CERT C guidelines [31]: EXP08-C

6.12.3 Mechanism of failure

Pointer arithmetic used incorrectly can produce:

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

6.12.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that allow pointer arithmetic.

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. They can

- Avoid using pointer arithmetic for accessing anything except composite types;
- Prefer indexing for accessing array elements rather than using pointer arithmetic in languages that permit the dual modes of access;
- Limit pointer arithmetic calculations to the addition and subtraction of integers.

6.12.6 Implications for language design and evolution

[None]

6.13 Null pointer dereference [XYH]

6.13.1 Description of application vulnerability

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

6.13.2 Related coding guidelines

CWE [7]:

476. NULL Pointer Dereference

JSF AV [24]: Rule 174

CERT C guidelines [31]: EXP34-C

Ada Quality and Style Guide [1]: 5.4.5

6.13.3 Mechanism of failure

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

6.13.4 Applicable language characteristics

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

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

6.13.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects by ensuring that prior to dereferencing a pointer, its value is not equal to `NULL`.

6.13.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider a language feature that would check a pointer value for `NULL` before performing an access.

6.14 Dangling reference to heap [XYK]

6.14.1 Description of application vulnerability

Memory designated by a dangling reference, see 3.7.6, can be reused as soon as the referenced object has been deleted; therefore, any subsequent access through the dangling reference may affect an apparently arbitrary location of memory, corrupting data or code.

This description concerns dangling references to the heap. The description of dangling references to stack frames can be found in 6.33 *Dangling reference to stack frame [DCM]*. In many languages, references are called pointers; the issues are identical.

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

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

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

Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or dereferencing `NULL` pointers or pointers that are not initialized.

6.14.2 Related coding guidelines

CWE [7]:

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

416. Use After Free

MISRA C [29]: 18.1-18.6

MISRA C++ [30]: 0-3-1, 7-5-1, 7-5-2, 7-5-3, and 18-4-1

CERT C guidelines [31]: MEM01-C, MEM30-C, and MEM31.C

Ada Quality and Style Guide [1]: 5.4.5, 7.3.3, and 7.6.6

6.14.3 Mechanism of failure

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

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

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

- An error condition or other exceptional circumstances that unexpectedly cause an object to become undefined;
- Developer confusion over which part of the program is responsible for freeing the memory.

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

6.14.4 Applicable language characteristics

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

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

6.14.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can;

- Use an implementation that checks whether a pointer is used that designates a memory location that has already been freed;
- Use a coding style that does not permit deallocation;
- In complicated error conditions, be sure that clean-up routines respect the state of allocation properly, such as if the language is object-oriented, ensure that object destructors delete each chunk of memory only once, and ensure that all pointers are set to `NULL` once the memory they point to have been freed;
- Use a static analysis tool that is capable of detecting some situations when a pointer is used after the storage it refers to is no longer a pointer to valid memory location;
- Allocate and free memory at the same level of abstraction, and ideally in the same code module

6.14.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Providing implementations of the free function that can tolerate multiple frees on the same reference/pointer or frees of memory that was never allocated. Such an operation is called an idempotent operation;

- For properties that cannot be checked at compile time, providing an assertion mechanism for checking properties at run-time, with the option to inhibit assertion checking if efficiency is a concern;
- Providing a storage allocation interface that will allow the called function to set the pointer used to NULL after the referenced storage is deallocated.

6.15 Arithmetic wrap-around error [FIF]

6.15.1 Description of application vulnerability

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

- the type is signed or unsigned;
- the specification of the language semantics and/or implementation choices;
- the computation *wraps around* to an unexpected value.

This vulnerability is related to 6.16 Using shift operations for multiplication and division [PIK].

6.15.2 Related coding guidelines

CWE [7]:

128. Wrap-around Error

190. Integer Overflow or Wraparound

JSF AV Rules [24]: 164 and 15

MISRA C [29]: 7.2, 10.1, 10.3, 10.4, 10.6, 10.7, and 12.4

MISRA C++ [30]: 2-13-3, 5-0-3 to 5-0-10, and 5-19-1

CERT C guidelines [31]: INT30-C, INT32-C, and INT34-C

6.15.3 Mechanism of failure

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

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

The precise consequences of wrap-around differ depending on:

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

- Implementation decisions.

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

6.15.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that do not trigger an exception condition when a wrap-around error occurs.

6.15.5 Avoiding the vulnerability or mitigating its effects

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

- Determine applicable upper and lower bounds for the range of all variables and use language mechanisms or static analysis to determine that values are confined to the proper range;

- Analyze the software using static analysis to identify unexpected consequences of arithmetic operations.

6.15.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider providing facilities to specify either an error, a saturated value, or a modulo result when numeric overflow occurs. Ideally, the selection among these alternatives could be made by the programmer.

6.16 Using shift operations for multiplication and division [PIK]

6.16.1 Description of application vulnerability

Using shift operations as a surrogate for multiply or divide may produce an unexpected value when the sign bit is changed or when value bits are lost. This vulnerability is related to *6.15 Arithmetic wrap-around error [FIF]*.

6.16.2 Related coding guidelines

CWE [7]:

128. Wrap-around Error

190. Integer Overflow or Wraparound

JSF AV Rules[31]: 164 and 15

MISRA C [29]: 7.2, 10.1, 10.3, 10.4, 10.6, 10.7, and 12.4

MISRA C++ [30]: 2-13-3, 5-0-3 to 5-0-10, and 5-19-1

CERT C guidelines [31]: INT30-C, INT32-C, and INT34-C

6.16.3 Mechanism of failure

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

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

6.16.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that permit logical shift operations on variables of arithmetic type.

6.16.5 Avoiding the vulnerability or mitigating its effects

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

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

6.16.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Not providing logical shifting on arithmetic values;
- Flagging all occurrences of logical shifts for reviewers.

6.17 Choice of clear names [NAI]

6.17.1 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. Typing errors can lead to unintended bindings. The problem is amplified if a language does not require explicit declarations of names.

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 programming 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 natural language and dialect specific, such as a pidgin, and may be meaningless to humans that speak other dialects.

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

The problems described above are different from overloading or overriding where the same name is used intentionally (and documented) to access closely linked sets of subprograms. This is also different than using reserved names which can lead to a conflict with the reserved use and the use of which may or may not be detected at compile time.

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

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

6.17.2 Related coding guidelines

JSF AV Rules [24]: 48, 49, 50, 51,52

MISRA C [29]: 1.1

CERT C guidelines [31]: DCL02-C

Ada Quality and Style Guide [1]: 3.2

6.17.3 Mechanism of Failure

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

6.17.4 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 treat letter case as significant. Some languages do not differentiate between names with differing case, while others do.

6.17.5 Avoiding the vulnerability or mitigating its effects

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

— Use static analysis tools to show the target of calls and accesses and to produce alphabetical lists of names, and possibly followed with human review to detect the names that are sorted at an unexpected location or which look almost identical to an adjacent name in the list;

— Use a language with a requirement to declare names before use or use available tool or compiler options to enforce such a requirement;

— Avoid names that conflict with (unreserved) keywords or language-defined library names for the language being used;

— Avoid names that only differ by characters that may be confused visually in the alphabet used in development, such as for the Roman alphabet characters such as 'O' and '0', 'l' (lower case 'L'), 'I' (capital '**i**') and '1', 'S' and '5', 'Z' and '2', and 'n' and 'h';

- Avoid names that only differ in the use of upper and lower case to other names;
- In languages with optional declarations of variables, always use explicit declarations of the variables to assist compiler checking;
- Use language features such as preconditions and postconditions or named parameter passing to facilitate the detection of accidentally incorrect function names.

6.17.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- providing an option to impose the declaration of names before use;
- Requiring that implementations use all the characters of a name when comparing names, instead of some fixed number of leading characters.

6.18 Dead store [WXQ]

6.18.1 Description of application vulnerability

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

This vulnerability is very similar to 6.19 *Unused variable [YZS]*.

6.18.2 Related coding guidelines

CWE [7]: 563. Unused Variable

MISRA C++ [30]: 0-1-4 and 0-1-6

CERT C guidelines [31]: MSC13-C

6.18.3 Mechanism of failure

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

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

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

A dead store is justifiable if, for example:

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

6.18.4 Applicable language characteristics

This vulnerability description is intended to be applicable to any programming language that provides assignment or initialized declarations.

6.18.5 Avoiding the vulnerability or mitigating its effects

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

- Use static analysis to identify any dead stores in the program and to ensure that there is a justification for each one;
- Avoid declaring variables of compatible types in nested scopes with similar names;
- If variables are intended to be accessed by other execution threads or external devices, mark them as volatile;
- To prevent potential leakage of sensitive information, assign some information-free value to the volatile object after the last intended read.

6.18.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider providing (possibly optional) warning messages for dead store.

6.19 Unused variable [YZS]

6.19.1 Description of application vulnerability

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

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

This vulnerability is similar to 6.18 *Dead store [WXQ]* if the variable is initialized but never used.

6.19.2 Related coding guidelines

CWE [7]: 563. Unused Variable

MISRA C++ [30]: 0-1-3

CERT C guidelines [31]: MSC13-C

6.19.3 Mechanism of failure

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

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

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

6.19.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that provide variable declarations.

6.19.5 Avoiding the vulnerability or mitigating its effects

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

- Enable detection of unused variables in the compiler;
- Use static analysis to identify any unused variables in the program and ensure that there is a documented justification for them.

6.19.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider providing (possibly optional) warning messages for unused variables.

6.20 Identifier name reuse [YOW]

6.20.1 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).

6.20.2 Related coding guidelines

JSF AV Rules [24]: 120, 135, 136 and 137,

MISRA C [29]: 5.3, 5.8, 5.9, 21.1, 21.2

MISRA C++ [30]: 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 [31]: DCL01-C and DCL32-C

Ada Quality and Style Guide [1]: 5.6.1 and 5.7.1

6.20.3 Mechanism of failure

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

For instance, in the following code fragment:

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

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

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

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

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

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

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

A related problem exists in languages that allow overloading or overriding of keywords or standard library function identifiers. Such overloading can lead to confusion about which entity is intended to be referenced. For issues of overriding and overloading methods in object-oriented programming, see 6.41 *Inheritance [RIP]*.

It is an important principle that definitions for new identifiers do not use a name that is already visible within the scope containing the new definition, or alternately, that language-specific facilities check for and prevent inadvertent overloading of names are used.

6.20.4 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.

6.20.5 Avoiding the vulnerability or mitigating its effects

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

- 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, including using a language-specific project coding convention 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 available language features, 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.

6.20.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Requiring mandatory diagnostics for entities with the same name in nested scopes;
- Requiring mandatory diagnostics for entity names that exceed the length that the implementation uses to define uniqueness;
- Requiring mandatory diagnostics for overloading or overriding of keywords or standard library function identifiers.

6.21 Namespace issues [BJL]

6.21.1 Description of Application Vulnerability

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

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

6.21.2 Related coding guidelines

MISRA C++ [30]: 7-3-1, 7-3-3, 7-3-5, 14-5-1, and 16-0-2

6.21.3 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 both namespaces to obtain names for direct usage, for an example.

```
use N1, N2; – presumed to make all names in N1 and N2
            -- directly visible in the scope of intended use
...
X := A + B;
```

The semantics of the above example are intuitive and unambiguous.

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

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

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

If a language supports overloading of subprograms, the notion of ‘same name’ used in the above example is extended to mean not only the same name, but also the same signature of the subprogram. For vulnerabilities associated with overloading and overriding, see *6.20 Identifier name reuse [YOW]* and *6.41 Inheritance [RIP]*. In the context of namespaces, however, adding signature matching to the name binding activity, 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, but 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.

6.21.4 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.

6.21.5 Avoiding the Vulnerability or Mitigating its Effects

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

- Avoid wholesale import directives, i.e. directives that give all imported names the same visibility level as each other and/or the same visibility level as local names (provided that the language offers the respective capabilities);
- Use only selective *single name* import directives or using fully qualified names (provided that the language offers the respective capabilities)

6.21.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Avoiding preference rules among mutable namespaces;
- Providing mechanisms such that ambiguities are invalid and avoidable by the user, for example, by using names qualified by their originating namespace.

6.22 Missing initialization of variables [LAV]

6.22.1 Description of application vulnerability

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

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

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

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

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

6.22.2 Related coding guidelines

CWE [7]: 457. Use of Uninitialized Variable

JSF AV Rules [24]: 71, 143, and 147

MISRA C [29]: 9.1, 9.2, and 9.3

MISRA C++ [30]: 8-5-1

CERT C guidelines [31]: DCL14-C and EXP33-C

Ada Quality and Style Guide [1]: 5.9.6

6.22.3 Mechanism of failure

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

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

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

The general problem of showing that all program objects are initialized is intractable.

6.22.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that permit variables to be read before they are assigned.

6.22.5 Avoiding the vulnerability or mitigating its effects

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

- Carefully structure programs to show that all variables are set before first read on every path throughout each subprogram;
- Use static analysis tools to show that all objects are set before use, and since the general problem is intractable, keep initialization algorithms simple so that they can be analyzed;
- When declaring and initializing the object together, if the language does not require the compiler to statically verify that the declarative structure and the initialization structure match, use static analysis tools to help detect any mismatches;
- Use dynamic tools where available to detect uninitialized variables during testing.

- When an object is visible from multiple modules, identify a module that must set the value before reads can occur from any other module that can access the object, and ensure that this module is executed first;
- When an object can be accessed concurrently, including by interrupts and co-routines, identify where early initialization occurs and show statically that the correct order is set, i.e. via program structure, not by timing, OS precedence, or chance;
- Consider initializing each object at declaration, or immediately after subprogram execution commences and before any branches;
- If the subprogram must commence with conditional statements, show statically that every variable declared and not initialized earlier is initialized on each branch;
- Ensure that the initial object value is a sensible value for the logic of the program;

  Note: So-called junk initialization (such as, for example, setting every variable to zero) prevents the use of tools to detect otherwise uninitialized variables;

- Define or reserve fields or portions of the object to only be set when fully initialized, understanding however, that this approach has the effect of setting the variable to possibly mistaken values while defeating the use of static analysis to find the uninitialized variables;
- When setting compound objects, if the language provides mechanisms to set all components together, use those in preference to a sequence of initializations as this facilitates coverage analysis; otherwise use tools that perform such coverage analysis and document the initialization.;
- Avoid performing partial initializations unless there is no choice and document any deviations from full initialization;
- Where default assignments of multiple components are performed, explicitly declare all component names and/or ranges to help static analysis and to identify component changes during maintenance;
- Use named assignments in preference to positional assignment where the language has named assignments so that such named assignments can be used to build reviewable assignment structures that can be analyzed by the language processor for completeness; otherwise use comments and secondary tools to help show correct assignment where the language only supports positional assignment notation.

6.22.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

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

6.23 Operator precedence and associativity [JCW]

6.23.1 Description of application vulnerability

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

Experience and experimental evidence show that developers can have incorrect beliefs about the relative precedence of many binary operators, as documented by D. Jones [23].

6.23.2 Related coding guidelines

JSF AV Rules [24]: 204 and 213

MISRA C [29]: 10.1, 12.1, 13.2, 14.4, 20.7, 20.10, and 20.11

MISRA C++ [30]: 4-5-1, 4-5-2, 4-5-3, 5-0-1, 5-0-2, 5-2-1, 5-3-1, 16-0-6, 16-3-1, and 16-3-2

CERT C guidelines [31]: EXP00-C

Ada Quality and Style Guide [1]: 7.1.8 and 7.1.9

6.23.3 Mechanism of failure

In C and C++, the bitwise operators (bitwise logical and bitwise shift) are sometimes thought of by the programmer having similar precedence to arithmetic operations, so just as one might correctly write

```
x – 1 == 0 //x minus one is equal to zero
```

a programmer might erroneously write

```
x & 1 == 0 // mentally meaning "(x and-ed with 1) is equal to zero"
```

whereas the operator precedence rules of C and C++ actually bind the expression as `x & (1==0)`,

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

Examples from an opposite extreme can be found in programs written in APL, which is noteworthy for the absence of any distinctions of precedence. One commonly made mistake is to write

```
a * b + c,
```

intending to produce

```
(a * b) + c,
```

whereas APL's uniform right-to-left associativity produces

```
(b + c) * a.
```

6.23.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages whose precedence and associativity rules are sufficiently complex that developers may not fully remember them.

6.23.5 Avoiding the vulnerability or mitigating its effects

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

- Adopt programming guidelines (preferably augmented by static analysis), for example, use the language-specific rules cross-referenced within 6.24 *Side effects and order of evaluation of operands [SAM]*;

- Use parentheses around binary operator combinations that are known to be a source of error (for example, mixed arithmetic/bitwise and bitwise/relational operator combinations);

- Break up complex expressions and use temporary variables to make the intended order clearer.

6.23.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- In the language definition, avoiding the provision of precedence or of a particular associativity for operators that are not typically ordered with respect to one another in arithmetic;

- Requiring full parenthesization to avoid misinterpretation.

6.24 Side-effects and order of evaluation of operands [SAM]

6.24.1 Description of application vulnerability

Some programming languages allow subexpressions to cause side-effects, such as assignment, increment, decrement, or broader effects even on the execution environment. For example, some programming languages permit such side-effects, and if, within one expression, two or more side-effects modify the same object, undefined behaviour results, for example, from C, `i = v[i++]`.

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

For example, in a robot scenario, the logical expression

```
Robot.Turn_Left(Angle) and Robot.Drive (Distance)
```

will have wildly different effects depending upon the order of evaluation of the subexpressions.

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

6.24.2 Related coding guidelines

JSF AV Rules [24]: 157, 158, 204, 204.1, and 213

MISRA C [29]: 12.1, 13.2, 13.5 and 13.6

MISRA C++ [30]: 5-0-1

CERT C guidelines [31]: EXP10-C, EXP30-C

Ada Quality and Style Guide [1]: 7.1.8 and 7.1.9

6.24.3 Mechanism of failure

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

(All examples here use the syntax of C or Java for brevity; the effects can be created in any language that allows functions with side-effects in the places where C allows the increment operations.)

Consider

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

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

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

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

or

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

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

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

where even if parentheses are placed around the i++ subexpressions, undefined behaviour still remains.

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

6.24.4 Applicable language characteristics

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

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

6.24.5 Avoiding the vulnerability or mitigating its effects

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

- Make use of one or more programming guidelines, which (a) prohibit unspecified or undefined behaviours, and (b) can be enforced by static analysis; (See JSF AV and MISRA rules in this clause);
- Keep expressions simple to reduce potential side effects, support static analysis, improve human comprehension, and reduce errors;
- Ensure that each expression results in the same value (including side effects), regardless of the order of evaluation or execution of terms of the expression.

6.24.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider language features that will eliminate or mitigate this vulnerability, such as pure functions.

6.25 Likely incorrect expression [KOA]

6.25.1 Description of application vulnerability

Certain expressions are symptomatic of what is likely to be a mistake made by the programmer. The statement is not contrary to the language standard but is unlikely to be intended. The statement may have no effect and effectively is a null statement or may introduce an unintended side-effect. A common example arises in languages that use == for equality and = for assignment and allow assignments as expressions: leading to the use of = in a Boolean expression where the programmer intended to perform an equality test using ==. It is valid and possible that the programmer intended to do an assignment within the `if` expression, but due to this being a common error, a programmer doing so would be using a poor programming practice. A less likely occurrence, but still possible is the substitution of == for = in what is supposed to be an assignment statement, but which effectively becomes a null statement. These mistakes may survive testing only to manifest themselves in deployed code where they may be maliciously exploited.

Other easily confused operators in languages are the logical operators such as `&&` for the bitwise operator `&`, or vice versa.

6.25.2 Related coding guidelines

CWE [7]:

480. Use of Incorrect Operator

481. Assigning instead of Comparing

482. Comparing instead of Assigning

570. Expression is Always False

571. Expression is Always True

JSF AV Rules [24]: 160

MISRA C [29]: 2.2, 13.3-13.6, and 14.3

MISRA C++ [30]: 0-1-9, 5-0-1, 6-2-1, and 6-5-2

CERT C guidelines [31]: MSC02-C and MSC03-C

6.25.3 Mechanism of failure

Substitution of = in place of == in a Boolean test in languages that use this syntax is an easy mistake to make. Other instances can be the result of intricacies of the language definition that specifies what part of an expression must be evaluated. For instance, having an assignment expression in a Boolean statement is likely assuming that the complete expression will be executed in all cases. However, this is not always the case as sometimes the truth-value of the Boolean expression can be determined after only executing some portion of the expression. Consider:

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

If `(a==b)` is determined to be `true`, then there is no need for the subexpression `(c=(d-1))` to be executed and as such, the assignment `(c=(d-1))` will not occur.

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

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

6.25.4 Applicable language characteristics

This vulnerability description is intended to be applicable to all languages, since all languages are susceptible to likely incorrect expressions.

6.25.5 Avoiding the vulnerability or mitigating its effects

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

- Simplify expressions;
- Avoid assignment expressions in function parameters, as sometimes the assignments can be executed in an unexpected order and instead, perform all assignments before the function call;
- Avoid assignments within a Boolean expression, and if intended move the assignment to before the Boolean expression for clarity and robustness;
- Use static analysis tools that detect and warn of expressions that include assignment within an expression;
- Annotate code that includes assignment within an expression to show that it is intentional and include rationale for the side effect;
- Avoid the use of statements that have no program effect, but if necessary, document with comments the rationale for the usage in each instance.

6.25.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following:

- Prohibiting assignments used as function parameters.
- Prohibiting assignments within a Boolean expression.
- Avoiding situations where easily confused symbols (such as = and ==, or `;` `:`, or `!=` and `/=`) are valid in the same context. For example, = is not generally valid in an `if` statement in Java because it does not normally return a `Boolean` value.

6.26 Dead and deactivated code [XYQ]

6.26.1 Description of application vulnerability

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

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

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

Dead and deactivated code is considered separately from the description of 6.19 *Unused variable [YZS]*.

6.26.2 Related coding guidelines

CWE [7]:

561. Dead Code

570. Expression is Always False

571. Expression is Always True

JSF AV Rules [24]: 127 and 186

MISRA C [29]: 2.1 and 4.4

MISRA C++ [30]: 0-1-1 to 0-1-10, 2-7-2, and 2-7-3

CERT C guidelines [31]: MSC07-C and MSC12-C

6.26.3 Mechanism of failure

Dead code in an application can never be executed, either because statically there is no call path to the code (for example, a function that is never called) or dynamically because the execution paths to the code can never be executed, as in

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

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

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

The presence of dead code is not in itself an error. There may also be legitimate reasons for its presence, for example:

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

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

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

However, it may be the case that, because of some other error, the code is rendered unreachable. Therefore, it is important to understand and document why dead code is present.

Be aware that some defensive code, such as that created to catch hardware error, may be optimized away by the compiler. Use of optimization fences such as *volatile* accesses (consult language and compiler manuals) may help.

6.26.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that allow code to exist in a program or executable, which can never be executed.

6.26.5 Avoiding the vulnerability or mitigating its effects

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

- Identify any dead code in the application using static analysis or testing with specialized tools;
- Remove dead code from an application unless its presence serves a documented purpose;

  NOTE 1 When a developer identifies code that is dead because a conditional consistently evaluates to the same value, this can be indicative of an earlier bug or indicative of inadequate path coverage in the test regimen. Investigation is essential to ascertain why the same value is occurring.

- For any deactivated code, provide a justification as to why it is present;
- Ensure that any code that was expected to be unused is documented as deactivated code;
- For code that appears to be dead code but is in reality accessible only by asynchronous events or error handlers, or present for debugging purposes, prevent the optimizations that remove the code in question;

  NOTE 2 Examples include the judicious use of *volatile* accesses, pragmas, or compiler switches.

- Apply standard branch coverage measurement tools and ensure by 100 % coverage that all branches are neither dead nor deactivated.

6.26.6 Implications for language design and evolution

[None]

6.27 Switch statements and lack of static analysis [CLL]

6.27.1 Description of application vulnerability

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

6.27.2 Related coding guidelines

JSF AV Rules [24]: 148, 193, 194, 195, and 196

MISRA C [29]: 16.3-16.6

MISRA C++ [30]: 6-4-3, 6-4-5, 6-4-6, and 6-4-8

CERT C guidelines [31]: MSC01-C

Ada Quality and Style Guide [1]: 5.6.1 and 5.6.10

6.27.3 Mechanism of failure

The fundamental challenge when using a `switch` statement is to make sure that all possible cases are, in fact, treated correctly. In most cases, this is not enforced by the language or the compiler. Possible consequences include:

- Not handling a case;
- Handling a case by a default clause instead of the specific case handling needed;
- Not detecting out-of-bounds cases;
- Jumping to ‘arbitrary’ code.

In particular, the last of these consequences can be exploited by malicious attacks.

6.27.4 Applicable language characteristics

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

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

6.27.5 Avoiding the vulnerability or mitigating its effects

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

- Ensure that every valid choice has a branch that covers the choice;
- Avoid default branches where it can be statically shown that each choice is covered by a branch;
- Use a default branch that initiates error processing where coverage of all choices by branches cannot be statically shown;
- Use a restricted set of enumeration values to improve coverage analysis where the language provides such capability;
- Avoid “flowing through” from one case to another;
- In cases where flow-through is necessary and intended, use an explicitly coded branch to clearly mark the intent and provide comments explaining the intention to help reviewers and maintainers;
- Perform static analysis to determine if all cases are, in fact, covered by the code;

  NOTE The use of a default case can hamper the effectiveness of static analysis since the tool cannot determine if omitted alternatives were or were not intended for default treatment.

- Use other means of mitigation including manual review, bounds testing, tool analysis, verification techniques, and proofs of correctness to show coverage.

6.27.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider language specifications that require compilers to ensure that a complete set of alternatives is provided in cases where the value set of the switch variable can be statically determined.

6.28 Non-demarcation of control flow [EOJ]

6.28.1 Description of application vulnerability

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

6.28.2 Related coding guidelines

JSF AV Rules [24]: 59 and 192

MISRA C [29]: 15.6 and 15.7

MISRA C++ [30]: 6-3-1, 6-4-1, 6-4-2, 6-4-3, 6-4-8, 6-5-1, 6-5-6, 6-6-1 to 6-6-5, and 16-0-2

Ada Quality and Style Guide[1]: 3, 5.6.1 through 5.6.10

6.28.3 Mechanism of failure

Programmers may rely on indentation to determine inclusion of statements within constructs. Testing of the software may not reveal that statements that appear to be included in a construct (due to formatting) but are actually outside of it because of the absence of a terminator. Moreover, for a nested `if-then-else` statement the programmer may be confused about which `if` statement controls the `else` part directly. This can lead to unexpected results.

6.28.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that contain loops and conditional statements that are not explicitly terminated by an `end` construct.

6.28.5 Avoiding the vulnerability or mitigating its effects

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

— Where the language does not provide demarcation of the end of a control structure, adopt a convention for marking the closing of a construct that can be checked by a tool, to ensure that program structure is apparent;

— Adopt programming guidelines (preferably augmented by static analysis). For example, consider the rules documented in 6.29.2 *Loop control variable abuse [TEX]*;

— Use other means of assurance, such as proofs of correctness, analysis with tools, and dynamic verification techniques;

— Use pretty-printers and syntax-aware editors to highlight such problems, but be aware that such tools sometimes disguise such errors;

— Where the language permits single statements after loops and conditional statements but permits optional compound statements, for example

in C

```
        if (...) statement else statement;
```

or Pascal

```
      if expression then statement else statement;
```

always use the compound version (i.e. C's `{... }` or Pascal's `begin... end`).

6.28.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Adding a mode that strictly enforces compound conditional and looping constructs with explicit termination, such as `end if` or a closing bracket;
- Creating syntax for explicit termination of loops and conditional statements;
- Providing syntax to terminate named loops and conditionals and to determine if the structure as named matches the structure as inferred.

6.29 Loop control variable abuse [TEX]

6.29.1 Description of application vulnerability

Many languages support a looping construct whose number of iterations is controlled by the value of a loop control variable. Looping constructs provide a method of specifying an initial value for this loop control variable, a test that terminates the loop and the quantity by which it is decremented or incremented on each loop iteration.

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

Some languages, such as C-based languages do not explicitly specify which of the variables appearing in a loop header is the control variable for the loop. MISRA C [29] and MISRA C++ [30] have proposed algorithms for deducing which, if any, of these variables is the loop control variable in the programming languages C and C++ (these algorithms could also be applied to other languages that support a C-like for-loop).

6.29.2 Related coding guidelines

JSF AV [24] Rule: 201

MISRA C [29]: 14.2

MISRA C++ [30]: 6-5-1 to 6-5-6

6.29.3 Mechanism of failure

The mechanism of failure is that changes to a loop control variable inside the loop body may cause the loop to unexpectedly:

- Exit prematurely;
- Execute forever;
- Not cover the complete range of values documented by the loop header.

Readers of source code often make assumptions about what has been written. A common assumption is that a loop control variable is not modified in the body of the loop. A programmer may write incorrect code based on this assumption. Similarly, reviewers, who are often domain specialists and not programmers, also make assumptions about written code and assume that loop control variables are not changed by the loop body.

6.29.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that allow a loop control variable to be modified in the body of its associated loop.

6.29.5 Avoiding the vulnerability or mitigating its effects

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

- Avoid modifying a loop control variable in the body of its associated loop body;
- Use a static analysis tool that identifies the modification of a loop control variable.

6.29.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the addition of an identifier type for loop control that cannot be modified by anything other than the loop control construct.

6.30 Off-by-one error [XZH]

6.30.1 Description of application vulnerability

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

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

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

The error may cause a bounds violation and the potential reading or writing of data and corresponding corruption of adjacent data. It can also be a serious security hole as it can permit someone to surreptitiously provide an unused location (such as 0 or the last element) that can be used for undocumented features or hidden channels.

6.30.2 Related coding guidelines

CWE [7]: 193. Off-by-one Error

6.30.3 Mechanism of failure

An off-by-one error could lead to:

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

- an incorrect conditional.

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

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

6.30.4 Applicable language characteristics

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

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

6.30.5 Avoiding the vulnerability or mitigating its effects

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

- Follow a systematic development process, use development/analysis tools and perform thorough testing are all common ways of preventing errors, and in this case, off-by-one errors;
- Use static analysis tools that warn of potential off-by-one errors;
- Where references are being made to array indices and the languages provide constructs to specify the whole array or the starting and ending indices explicitly (for example, Ada[17] provides the attributes 'First and 'Last for each dimension), use the language-provided constructs instead of numeric literals. Where the language does not provide such constructs, declare named constants and use them in preference to numeric literals;
- Where the language does not encapsulate variable length arrays, provide encapsulation through library objects and a coding standard developed that requires such arrays to only be used via those library objects, so the developer does not need to be explicitly concerned with managing bounds values.

6.30.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider providing encapsulations for arrays that:

- Prevent the need for the developer to be concerned with explicit bounds values;
- Provide the developer with symbolic access to the array start, end, and iterators.

6.31 Unstructured programming [EWD]

6.31.1 Description of application vulnerability

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

6.31.2 Related coding guidelines

JSF AV Rules [24]: 20, 113, 189, 190, and 191

MISRA C [29]: 15.1-15.3, and 21.4

MISRA C++ [30]: 6-6-1, 6-6-2, 6-6-3, and 17-0-5

CERT C guidelines [31]: SIG32-C

Ada Quality and Style Guide[1]: 3, 4, 5.4, 5.6, and 5.7

6.31.3 Mechanism of failure

Lack of structured programming can lead to:

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

6.31.4 Applicable language characteristics

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

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

6.31.5 Avoiding the vulnerability or mitigating its effects

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

- Avoid using language features that transfer control of the program flow via a jump, such as `goto`;
- Avoid using language features such as `continue` and `break` in the middle of loops;
- Avoid the use of multiple exit points from a function/procedure/method/subroutine unless it can be shown that the code with multiple exit points is superior;
- Avoid multiple entry points to a function/procedure/method/subroutine;
- Use only those features of the programming language that enforce a logical structure on the program and create program flow that follows a simple hierarchical model that employs looping constructs such as `for`, `repeat`, `do`, and `while`.

6.31.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider supporting and favouring structured programming enforced through language constructs to the extent possible.

6.32 Passing parameters and return values [CSJ]

6.32.1 Description of application vulnerability

Nearly every procedural language provides some method of abstraction that permits decomposition of the flow of control into routines, functions, subprograms, or methods. (For the purpose of this description, the term subprogram will be used.) To have any effect on the computation, the subprogram must change data visible to the calling program. It can do this by changing the value of a non-local variable, changing or setting the value of a parameter, or, in the case of a function, providing a return value. Because different languages use different mechanisms with different semantics for passing parameters, a programmer using an unfamiliar language may obtain unexpected results.

6.32.2 Related coding guidelines

JSF AV Rules [24]: 20, 116

MISRA C [29]: 8.2, 8.3, 8.13, and 17.1-17.3

MISRA C++ [30]: 0-3-2, 7-1-2, 8-4-1, 8-4-2, 8-4-3, and 8-4-4

CERT C guidelines[31]: EXP12-C and DCL33-C

Ada Quality and Style Guide [1]: 5.2 and 8.3

6.32.3 Mechanism of failure

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

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

In call by copy, the called subprogram does not share data with the calling program. Instead, formal parameters act as local variables. Values are passed between the actual arguments and the formal parameters by copying.

Some languages may control changes to formal parameters based on labels such as `in`, `out`, or `inout`. There are three cases to consider:

- ‘call by value’ for `in` parameters;
- ‘call by result’ for `out` parameters and function return values;
- ‘call by value-result’ for `inout` parameters.

For call by value, the calling program evaluates the actual arguments and copies the result to the corresponding formal parameters that are then treated as local variables by the subprogram. For call by result, the values of the locals corresponding to formal parameters are copied to the corresponding actual arguments. For call by value-result, the values are copied in from the actual arguments at the beginning of the subprogram's execution and back out to the actual arguments at its termination.

The obvious disadvantage of call by copy is that extra copy operations are needed, and execution time is required to produce the copies. Particularly if parameters represent sizable objects, such as large arrays, the cost of call by copy can be high. For this reason, many languages also provide the call by reference mechanism.

The disadvantage of call by reference is that the calling program cannot be assured that the subprogram has not changed data that was intended to be unchanged. For example, if an array is passed by reference to a subprogram intended to sum its elements, the subprogram could also change the values of one or more elements of the array. However, some languages enforce the subprogram's access to the shared data based on the labelling of actual arguments with modes—such as `in`, `out`, or `inout` or by pointers to constant objects.

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

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

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

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

An additional complication with subprograms occurs when one or more of the arguments are expressions. In such cases, the evaluation of one argument might have side-effects that result in a change to the value of another or unintended aliasing. Implementation choices regarding order of evaluation could affect the result of the computation. This particular problem is described in 6.24 *Side-effects and order of evaluation of operands [SAM]*.

6.32.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that provide mechanisms for defining subprograms where the data passes between the calling program and the subprogram via parameters and return values. This includes methods in many popular object-oriented languages.

6.32.5 Avoiding the vulnerability or mitigating its effects

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

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

  - To avoid unintentional aliasing effects, avoid using expressions or function calls as actual arguments; instead assign the result of the expression to a temporary local and pass the local;
  - Utilize tools or other forms of analysis to ensure that non-obvious instances of aliasing are absent:
- Perform reviews or analysis to determine that called subprograms fulfil their responsibilities to assign values to all output parameters.

6.32.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider providing labels, such as `in`, `out`, and `inout`, that control the subprogram's access to its formal parameters, and enforce the controlled access.

6.33 Dangling references to stack frames [DCM]

6.33.1 Description of application vulnerability

Many languages allow the address of a local variable to be stored as a value in other variables. Examples are the application of the address operator in C or C++, or of the '`Access` or '`Address` attributes in Ada[17]. In some languages, this facility is also used to model the call-by-reference mechanism by passing the address of the actual parameter by-value. An obvious safety requirement is that the stored address shall not be used after the lifetime of the local variable has expired. This situation can be described as a *dangling reference to the stack*.

6.33.2 Related coding guidelines

CWE [7]: 562. Return of Stack Variable Address

JSF AV [24] Rule: 173

MISRA C [29]: 4.1 and 18.6

MISRA C++ [30]: 0-3-1, 7-5-1, 7-5-2, and 7-5-3

CERT C guidelines [31]: EXP35-C and DCL30-C

Ada Quality and Style Guide [1]: 7.6.7, 7.6.8, and 10.7.6

6.33.3 Mechanism of failure

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

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

struct s secret;
 array_type* ptr2;
```

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

The risk of variant 1 is the assignment of the address of `Arr` to a pointer variable that survives the lifetime of `Arr`. The fault is the subsequent use of the dangling reference to the stack, which references memory since altered by other calls and possibly validly owned by other routines. As part of a call-back, the fault allows systematic examination of portions of the stack contents without triggering an array-bounds-checking violation. Thus, this vulnerability is easily exploitable. As a fault, the effects can be most astounding, as memory gets corrupted by completely unrelated code portions.

A life-time check as part of pointer assignment can prevent the risk, and in many cases, such as the situations above, the check is statically decidable by a compiler. However, for the general case, a dynamic check is needed to ensure that the copied pointer value lives no longer than the designated object.

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

6.33.4 Applicable language characteristics

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

— The address of a local entity (or formal parameter) of a routine can be obtained and stored in a variable or can be returned by this routine as a result;

— No check is made that the lifetime of the variable receiving the address is not longer than the lifetime of the designated entity.

6.33.5 Avoiding the vulnerability or mitigating its effects

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

— Avoid using the address of locally declared entities as storable, assignable or returnable value (except where idioms of the language make it unavoidable).

— When such an address is stored, ensure that the lifetime of the variable containing the address is completely enclosed by the lifetime of the designated object;

— Never return the address of a local variable as the result of a function call.

6.33.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Not providing means to obtain the address of a locally declared entity as a storable value;
- Defining implicit checks to implement the assurance of enclosed lifetime expressed in 6.33.5.

  NOTE In many cases, the check is statically decidable, for example, when the address of a local entity is taken as part of a return statement or expression.

6.34 Subprogram signature mismatch [OTR]

6.34.1 Description of application vulnerability

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

6.34.2 Related coding guidelines

CWE [7]:

628. Function Call with Incorrectly Specified Arguments

686. Function Call with Incorrect Argument Type

683. Function Call with Incorrect Order of Arguments

JSF AV [24] Rule: 108

MISRA C [29]: 8.2-8.4, 17.1, and 17.3

MISRA C++ [30]: 0-3-2, 3-2-1, 3-2-2, 3-2-3, 3-2-4, 3-3-1, 3-9-1, 8-3-1, 8-4-1, and 8-4-2

CERT C guidelines [31]: DCL31-C, and DCL35-C

6.34.3 Mechanism of failure

When a subprogram is called, the actual arguments of the call are pushed on to the execution stack. When the subprogram terminates, the formal parameters are popped off the stack. If the number and type of the actual arguments do not match the number and type of the formal parameters, then depending upon the calling mechanism used by the language translator, the push and the pop will not be consistent and, if so, the stack will be corrupted.

Stack corruption can lead to unpredictable execution of the program and can provide opportunities for execution of unintended or malicious code.

The compilation systems for many languages and implementations can check to ensure that the list of actual parameters and any expected return match the declared set of formal parameters and return value (the *subprogram signature*) in both number and type. However, when the call is being made to an externally compiled subprogram, an object-code library, or a module compiled in a different language, the programmer must take additional steps to ensure a match between the expectations of the caller and the called subprogram.

For functions that accept a variable number of parameters, then parameter mismatches are particularly likely.

6.34.4 Applicable language characteristics

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

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

6.34.5 Avoiding the vulnerability or mitigating its effects

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

- Use language or compiler support or static analysis tools to detect mismatches in calling signatures and the actual subprogram, particularly in multilingual environments;
- Take advantage of any mechanism provided by the language to ensure that subprogram signatures match;
- Avoid any language features that permit variable numbers of actual arguments without a method of enforcing a match for any instance of a subprogram call;
- Take advantage of any language or implementation feature that guarantees matching the subprogram signature in linking to other languages or to separately compiled modules;
- Intensively review subprogram calls where the match is not guaranteed by tooling;
- Ensure that only a trusted source is used when using non-standard imported modules.

6.34.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Ensuring that the signatures of subprograms match within a single compilation unit;
- Providing features for asserting and checking the match with externally compiled subprograms.

6.35 Recursion [GDL]

6.35.1 Description of application vulnerability

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

6.35.2 Related coding guidelines

CWE [7]: 674. Uncontrolled Recursion

JSF AV [24] Rule: 119

MISRA C [29]: 17.2

MISRA C++ [30]: 7-5-4

CERT C guidelines [31]: MEM05-C

Ada Quality and Style Guide [1]: 5.6.6

6.35.3 Mechanism of failure

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

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

6.35.4 Applicable language characteristics

This vulnerability description is intended to be applicable to any language that permits the recursive invocation of subprograms.

6.35.5 Avoiding the vulnerability or mitigating its effects

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

- Minimize the use of recursion.
- Convert recursive calculations to the corresponding iterative calculation. In principle, any recursive calculation can be remodelled as an iterative calculation which will have a smaller impact on some computing resources but which may be harder for a human to comprehend. The cost to human understanding must be weighed against the practical limits of computing resource.
- Use static analysis to detect non-obvious recursive call paths such as indirect and long recursive call cycles.
- Restrict recursion to cases where the depth of recursion can be shown to be statically bounded by a tolerable number and document this number. Alternatively, monitor the depth of the recursion through a mechanism such as passing a recursion depth value that is incremented for each level of recursion, and using explicit comparison against a maximum depth limit to trigger handling of the situation.

6.35.6 Implications for language design and evolution

[None]

6.36 Ignored error status and unhandled exceptions [OYB]

6.36.1 Description of application vulnerability

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

6.36.2 Related coding guidelines

CWE [7]: 754. Improper Check for Unusual or Exceptional Conditions

JSF AV Rules [24]: 115 and 208

MISRA C [29]: 4.7

MISRA C++ [30]: 15-3-2 and 19-3-1

CERT C guidelines [31]: DCL09-C, ERR00-C, and ERR02-C

Ada Quality and Style Guide [1]: 4.1

6.36.3 Mechanism of failure

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

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

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

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

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

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

Different error handling mechanisms have different strengths and weaknesses. Dealing with exception handling in some languages can stress the capabilities of static analysis tools and can, in some cases, reduce the effectiveness of their analysis. Inversely, the use of error status variables can lead to confusingly complicated control structures, particularly when recovery is not possible locally. Therefore, for situations where the highest of reliability is required, the decision for or against exception handling deserves careful thought. In any case, it is important that exception-handling mechanisms be reserved for truly unexpected situations and other situations where no local recovery is possible. Situations which are merely unusual, like the end of file condition, are better treated by explicit testing—either prior to the call which might raise the error or immediately afterward.

In general, error detection, reporting, correction, and recovery are problematic if made a late opportunistic add-on. They are far more effective if made be an integral part of the system design.

6.36.4 Applicable language characteristics

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

6.36.5 Avoiding the vulnerability or mitigating its effects

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

- Reserve exception-handling mechanisms for truly unexpected situations and other situations where no local recovery is possible.
- Handle exceptions by the exception handlers of an enclosing construct as close as possible to the origin of the exception but as far out as necessary to be able to deal with the error., and consider preventing implicit exceptions by checking the error condition in the code prior to executing the construct that causes the exception.
- Equally, check error return values or auxiliary status variables following a call to a subprogram, unless it is demonstrated that the error condition is impossible.
- When functions return error values, check the error return values before processing any other returned data.
- For each routine, document all error conditions, matching error detection and reporting needs, and provide sufficient information for handling the error situation.
- Use static analysis tools to detect and report missing or ineffective error detection or handling.
- When execution within a particular context is abandoned due to an exception or error condition, finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.
- Retreat to a context where the fault can be handled completely (after finalizing and terminating the current context) when it is not appropriate to repair an error situation and retry the operation.
- Always enable error checking provided by the language, the software system, or the hardware in the absence of a conclusive analysis that the error condition is rendered impossible.
- Carefully review all error handling mechanisms, because of the complexity of error handling.
- In applications with the highest requirements for reliability, use defense-in-depth approaches, for example, checking and handling errors even if thought to be impossible.

6.36.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider a standardized set of mechanisms for detecting and treating error conditions, so that all languages to the extent possible can use them. This does not mean that all languages use the same mechanisms, as there will be a variety, but that each of the mechanisms be standardized.

6.37 Type-breaking reinterpretation of data [AMV]

6.37.1 Description of application vulnerability

In most cases, objects in programs are assigned locations in processor storage to hold their value. If the same storage space is assigned to more than one object—either statically or temporarily—then a change in the value of one object will have an effect on the value of the other. Furthermore, if the representation of the value of an object is reinterpreted as being the representation of the value of an object with a different type, unexpected results may occur.

6.37.2 Related coding guidelines

JSF AV Rules [24] 153 and183

MISRA C [29]: 19.1, and 19.2

MISRA C++ [30]: 4-5-1 to 4-5-3, 4-10-1, 4-10-2, and 5-0-3 to 5-0-9

CERT C guidelines [31]: MEM08-C

Ada Quality and Style Guide [1]: 7.6.7 and 7.6.8

6.37.3 Mechanism of failure

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

Unintentional or malicious reinterpretation of data can cause overwriting or disclosure of arbitrary memory regions. In addition, type-breaking reinterpretation of representation presents obstacles to human understanding of the code, the ability of tools to perform effective static analysis, and the ability of code optimizers to do their job.

Examples include:

- Providing alternative mappings of objects into blocks of storage performed either statically (such as the Fortran `common statement`) or dynamically (such as pointers).
- Union types, particularly unions that do not have a discriminant stored as part of the data structure. (Discriminants are additional components of the data structure that determine the layout of the rest of the data. If the discriminant capability is not provided by the language, then it is the programmer's responsibility to ensure consistency).
- Operations that permit a stored value to be interpreted as a different type (such as treating the representation of a pointer as an integer).

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

It is easier to avoid operations that reinterpret the same stored value as representing a different type when the language clearly identifies them. For example, Ada[17] forces the programmer to explicitly declare the conversion to be an instantiation of `Unchecked_Conversion`.

A much more difficult situation occurs when pointers are used to achieve type reinterpretation. Many languages perform type-checking of pointers and place restrictions on the ability of pointers to access arbitrary locations in storage (See 6.11 Pointer type conversions [HFC]).

6.37.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that permit multiple interpretations of the same bit pattern.

6.37.5 Avoiding the vulnerability or mitigating its effects

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

- Avoid reinterpretation performed as a matter of convenience; for example, avoid an integer pointer to manipulate character string data. When type-breaking reinterpretation is necessary, document it carefully in the code.
- When using union types, use discriminated unions in preference to non-discriminated unions.
- Avoid operations that reinterpret the same stored value as representing a different type.
- When data is reinterpreted with a different type, use language-defined capabilities to flag and check such usage (such as Ada's `'Valid` attribute), or use static analysis to show that the operation always succeeds.
- Use static analysis tools to locate situations where unintended reinterpretation occurs.
- As the presence of reinterpretation greatly complicates static analysis for other problems, consider segregating intended reinterpretation operations into distinct subprograms.

6.37.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

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

6.38 Deep vs. shallow copying [YAN]

6.38.1 Description of application vulnerability

When structures containing references as data components are copied, one must decide whether the references are being copied (*shallow copy*) or, instead, the objects designated by the references are being copied and a reference to the newly created object is used as the component value of the copied structure (*deep copy*). Almost all languages define structure-copying operations as shallow copies, i.e. the copied structure references the same object. Deep copying is algorithmically more challenging, since no object shall be copied twice although it may be reachable by multiple paths within the graph spanned by the references. Further, deep copying may be expensive in time and memory consumption. If, however, a shallow copy is made where a deep copy was needed, serious aliasing problems can arise in the objects that are part of the graphs spanned by the copied references. Subsequent modification of such an object is visible via both the old and the new structure.

An identical problem arises when array indices are stored as component values (in lieu of pointers or references) and used to access objects in an array outside the copied data structure.

6.38.2 Related coding guidelines

JSF AV [24] Rules: 76, 77, 80

Ada Quality and Style Guide [1]: Sections 5.4, 5.5

6.38.3 Mechanism of failure

Problems with shallow copying arise when values in the objects (transitively) referenced by the original or the copy are assigned to: in a deep copy, such assignments affect only the original or the copy of the graph, respectively; in a shallow copy, the value of the object is changed in both graphs, which may not have been the intention of the programmer. Consequently, the problem may manifest itself only during maintenance when, for the first time, such as assignment to a contained object is introduced, while shallow copying was originally chosen for reasons of efficiency but relying on the absence of assignments.

Knowledge of the use of shallow copying in lieu of deep copying can be exploited in attacks by causing unintended changes in data structures via the described aliasing effect.

The exposure and effects are similar to any other unintended aliasing, such as CSJ Passing Parameters and Return Values.

6.38.4 Applicable language characteristics

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

- Languages that have pointers or references as part of composite data structures.
- Languages that support arrays.

6.38.5 Avoiding the vulnerability or mitigating its effects

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

- Use shallow copying only where the aliasing caused is intended and comment usage at the usage point;
- Use deep copying if there is any possibility that the aliasing of a shallow copy would affect the application adversely;
- Use abstractions to ensure deep copies where needed, e.g. by (re-)defining assignment operations, constructors, and other operations that copy component values.

6.38.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider providing mechanisms to create abstractions that guarantee deep copying where needed.

6.39 Memory leaks and heap fragmentation [XYL]

6.39.1 Description of application vulnerability

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

6.39.2 Related coding guidelines

CWE [7]: 401. Failure to Release Memory Before Removing Last Reference (aka ‘Memory Leak’)

JSF AV [24] Rule: 206

MISRA C [29]: 4.12

CERT C guidelines [31]: MEM00-C and MEM31-C

Ada Quality and Style Guide [1]: 5.4.5, 5.9.2, and 7.3.3

6.39.3 Mechanism of failure

As a process or system runs, any memory taken from dynamic memory and not returned or reclaimed (by the runtime system, the application, or a garbage collector) after it ceases to be used, may result in future memory allocation requests failing for lack of free space.

Alternatively, memory claimed and returned can cause the heap to fragment into progressively smaller blocks, which, with the usual allocators, will result in a higher memory consumption and steadily increasing search times for blocks of suitable size, until the system spends most of the CPU-time for searching the heap for suitable blocks.

Either condition can thus result in a memory exhaustion exception, progressively slower performance by the allocating application, program termination or a system crash.

If an attacker can determine the cause of an existing memory leak or can increase the allocation rate for blocks of different sizes, the attacker will be able to cause the application to leak or fragment quickly and therefore cause the application to crash or fail to perform within acceptable time limits. Denial-of-Service attacks can thus occur.

Some modern languages have added a concept of “ownership” to simplify the lifetime management of objects allocated on the heap and to control access (such as writing).

6.39.4 Applicable language characteristics

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

- Languages that reclaim memory under programmer control can exhibit heap fragmentation and memory leaks;
- Languages that support mechanisms to dynamically allocate memory and employ garbage collection can exhibit memory leaks (and if the garbage collection is not coalescing, heap fragmentation).

6.39.5 Avoiding the vulnerability or mitigating its effects

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

- Use garbage collectors that reclaim memory no longer accessible by the application, as some garbage collectors are part of the language while others are add-ons;
- In systems with garbage collectors, set all non-local pointers or references to null, when the designated data is no longer needed, since the data transitively reachable from such a pointer or reference will not be garbage-collected otherwise, effectively causing memory leaks;
- In systems without garbage collectors, cause deallocation of the data before the last pointer or reference to the data is lost;
- Allocate and free memory at the same level of abstraction, and ideally in the same code module;

NOTE 1 Allocating and freeing memory in different modules and levels of abstraction can make it difficult for developers to match requests to free storage with the appropriate storage allocation request. This can cause confusion regarding when and if a block of memory has been allocated or freed, leading to memory leaks.

— When available, take advantage of ownership concepts to manage the heap;

— Use reference counting techniques or choose languages that use reference-counting techniques to eliminate storage leaks;

— Use Storage pools when available in combination with strong typing.

NOTE 2 Storage pools are a specialized memory mechanism where all of the memory associated with a class of objects is allocated from a specific bounded region such that storage exhaustion in one pool does not affect the code operating on other memory.

— Use storage pools of equally-sized blocks to avoid fragmentation within each storage pool and if necessary, provide application-specific (de-)allocators to achieve this functionality;

— Avoid the use of dynamically allocated storage entirely, or allocate only during system initialization and never allocate once the main execution commences, particularly in safety-critical systems (and hence for safety-critical software) and long running systems;

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

6.39.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

— Providing syntax and semantics to guarantee program-wide that dynamic memory is not used (such as the configuration pragmas feature offered by some programming languages).

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

6.40 Templates and generics [SYM]

6.40.1 Description of application vulnerability

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

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

6.40.2 Related coding guidelines

JSF AV Rules [24]: 101, 102, 103, 104, and 105

MISRA C++ [30]: 14-6-1, 14-6-2, 14-7-1 to 14-7-3, 14-8-1, and 14-8-2

Ada Quality and Style Guide [1]: 8.3.1 through 8.3.8, and 8.4.2

6.40.3 Mechanism of failure

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

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

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

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

Similar confusion can arise if the language permits specific methods of an instance of a generic to be explicitly defined, rather than using the common code, so that behaviour is not consistent for all instantiations. For example, for the same generic container class, the sort member normally sorts the elements of the container into ascending order. In some languages, a 'special case' can be created for the instantiation of the generic with a particular type. For example, the sort member for a 'float' container may be explicitly defined to provide different behaviour, say sorting the elements into descending order. Specialization that does not affect the apparent behaviour of the instantiation is not an issue.

6.40.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages that permit definitions of objects or functions to be parameterized by type, for later instantiation with specific types, such as Templates in C++, or Generics in Ada [17] or Java.

6.40.5 Avoiding the vulnerability or mitigating its effects

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

— Document the properties of an instantiating type necessary for a generic to be valid;

— If an instantiating type has the required properties, ensure that all operations of the generic are either valid or unavailable, whether currently used in the program or not;

— Avoid, or carefully document, any 'special cases' where a generic is instantiated with a specific type but does not behave as it does for other types.

6.40.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

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

6.41 Inheritance [RIP]

6.41.1 Description of application vulnerability

Inheritance, the ability to create enhanced and/or restricted object classes based on existing object classes, can introduce a number of vulnerabilities, both inadvertent and malicious. Because inheritance allows the overriding of methods of the parent class and because object-oriented systems are designed to encapsulate code and data, it can be difficult to determine where in the hierarchy an invoked method is actually defined. Also, since an overriding method does not need to call the method in the parent class that has been overridden, essential initialization and manipulation of class data may be bypassed. This can be especially dangerous in constructor methods, copy methods, or destructor methods and in particular when private data components (that is, data components not visible to methods of subclasses) of the parent class are left uninitialized or unchanged. Serious violations of type invariants can arise as a consequence.

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

6.41.2 Related coding guidelines

JSF AV Rules [24]: 78, 79, 80, 81, 86, 87, 88, 89, 89, 90, 91, 92, 93, 94, 95, 96 and 97

MISRA C++ [30]: 0-1-12, 8-3-1, 10-1-1 to 10-1-3, and 10-3-1 to 10-3-3

Ada Quality and Style Guide [1]: 9

6.41.3 Mechanism of failure

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

- Execution of malicious redefinitions, which can occur through the insertion of a class into the class hierarchy that overrides commonly called methods in the parent classes.
- Accidental redefinition, where a method is defined that inadvertently overrides a method that has already been defined in a parent class.
- Accidental failure of redefinition, when a method is incorrectly named or the parameters are not defined properly, and thus does not override a method in a parent class.
- Breaking of class invariants, which can be caused by redefining methods that initialize, copy, destroy or validate class data without including that initialization, copying, destruction, or validation in the overriding methods. This applies particularly to class invariants involving data of the parent class not visible in methods of the subclass. Inherited methods of the parent that have access to these “private” components will likely fail, if the components are left uninitialized or set inappropriately.

- Direct reading and writing of visible class members instead of using inherited *get* and *set* member functions, thus missing additional functionality provided by these member functions.

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

As methods are inherited from multiple chains of ancestors, the determination of which methods implementations exist and are being called, becomes increasingly more difficult for the programmer. Understanding which methods and data components apply to a given (sub)class becomes exceedingly difficult if these methods or components are inherited homographs (i.e. data components with identical names or methods with identical signatures). Different languages have different rules to resolve the resulting ambiguities. Misunderstandings lead to inadvertent coding errors. The complexity increases even more when multiple inheritance is used to model "has-a" relationships (see 6.42 *Violations of the Liskov substitution principle [BLP]*): methods never intended to be applicable to instances of a subclass are inherited nevertheless. For example, an instance of class `aircraftCarrier` may be "`turn`"ed merely because it obtained its propulsion screw by a "has-a"-inheritance with "turn" being an obviously meaningful method for the class of `propulsionScrew`. Meanwhile the user has a quite different expectation of what it means to turn an aircraft carrier. The complications increase if the carrier inherits twice from the class `propulsionScrew` because it has two propulsion screws.

Finally, if ambiguities in method or component namings are resolved by preference rules, changes in the execution of methods can be introduced by adding yet another unrelated but homographic method or data declaration anywhere is the hierarchies of ancestor classes during maintenance of the code. Malicious implementations can thus be added with each release of an object-oriented library and affect the behaviour of previously verified code. (see 6.42 *Violations of the Liskov substitution principle [BLP]*)

The mechanism of failure for these additional dangers caused by multiple inheritance is the inadvertent use of the wrong data components or methods. Knowledge of such incorrect use might be exploitable, as instances of the affected (sub)class may be corrupted by inappropriate operations.

6.41.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that allow single or multiple inheritances.

6.41.5 Avoiding the vulnerability or mitigating its effects

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

- Avoid the use of multiple inheritance whenever possible.
- Avoid accessing data components when getting and setting functions are available for them.
- Provide complete documentation of all encapsulated data, and how each method affects that data for each object in the hierarchy.
- Inherit only from trusted sources, and, whenever possible, check the version of the parent classes during compilation and/or initialization.
- Prohibit the use of visible inheritance for "has-a" relationships.
- Use components of the respective class for "has-a"-relationships.
- Delegate initialization, copy, or destruction of the parent's data components by calling the corresponding operation of the parent type, and in particular when the parent has data components not visible to methods of the subclass.

6.41.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Providing a compiler option to report the class in which a resolved method resides.
- Providing for runtime environments a trace of all runtime method resolutions.

6.42 Violations of the Liskov substitution principle or the contract model [BLP]

6.42.1 Description of application vulnerability

Object orientation typically allows polymorphic variables containing values of subclasses of the declared class of the variable. Methods of the declared class of a receiving object can be invoked and the caller has the right to expect that the semantics of the interface called upon are observed regardless of the precise nature of the value of the receiving object. Similarly, the existence of accessed components of the declared class needs to be ensured. Instances of subclasses thus need to be both technically and logically specialized instances of the parent class. This is the basis of the Liskov substitution principle.

The Liskov substitution principle states that an instance of a subclass is always an instance of the superclass as well if one ignores the added specializations. It implies that inheritance is used only if there is a logical "is-a"-relationship between the subclass and the superclass. Moreover, preconditions of methods can at most be weakened and never strengthened as they are redefined for a subclass. Inversely, postconditions can at most be strengthened and never be weakened by such a redefinition. The caller of an interface needs to guarantee only the preconditions of the interface and is allowed to rely on its postconditions. The rules stated make sure of this property which is also known as the Contract Model.

Violations of the Liskov substitution principle or the Contract Model can result in system malfunctions as additional preconditions of redefinitions or promised postconditions of interfaces are not met.

An alternative inheritance semantics is that of "has-a"-relationships, usually appearing in programs in languages with multiple inheritance, where the paradigm is sometimes referred to as a "mix-in". It is in stark conflict with the Liskow Principle, since a polymorphic variable `motor` of class `engine` should not be able to hold a `car`, merely because the subclass `car` was created by a mix-in of the class `engine` to the class `vehicle`.

The principles stated above apply to implicit as well as explicit preconditions and postconditions. Explicit conditions permit formal reasoning tools to be applied.

6.42.2 Related coding guidelines

JSF AV Rules [24]: 89, 91, 92, 93

Ada Quality and Style Guide [1]: 9.3.2

6.42.3 Mechanism of failure

When a client calls the method of a class which dispatches to the implementation of a subclass with a strengthened precondition, the client has no mechanism to know about the added preconditions to be satisfied. Hence the call may fail on a violated precondition. Similarly, if the called implementation has a weaker postcondition, the postcondition asserted to the client might not be satisfied. As a consequence, the client may fail. Failing to meet preconditions or to guarantee postconditions is bound to cause exceptions or system failures. The specific scenarios are extensive and range from faults that happen to be handled by the system to complete loss of security and safety.

Using visible inheritance to implement a "has-a"-relationships deteriorates class design and thereby may be the cause of consequential errors. There is no immediate failure mode, however.

6.42.4 Applicable language characteristics

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

- Languages that have polymorphic variables, particularly object-oriented languages;
- Languages that provide inheritance among classes.

6.42.5 Avoiding the vulnerability or mitigating its effects

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

- Obey all preconditions and postconditions of each method, whether they are specified in the language or not;
- Prohibit the strengthening of preconditions (specified or not) by redefinitions of methods;
- Prohibit the weakening of postconditions (specified or not) by redefinitions of methods;
- Prohibit the use of visible inheritance for "has-a" relationships and use components of the respective class for "has-a"-relationships instead;
- Use static analysis tools that identify misuse of inheritance in the contract model.

6.42.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider providing language mechanisms to formally specify preconditions and postconditions, including class-wide preconditions and postconditions.

6.43 Redispatching [PPH]

6.43.1 Description of application vulnerability

When very similar functionality is provided by methods or interfaces with varying parameter structures, a frequently found implementation strategy is to designate one of them as the *work horse* and have all others call on it to perform the (common) work. A prime example are constructor or initialization methods where different sets of initial values for certain components are provided and the remaining components are set to default values.

When the semantics of inner calls of dispatching methods ask for dispatching in turn, the call is said to be *redispatching*. In this case, the following scenario can evolve: In `class C`, the implementation of `method A` dispatches to `method B`, the work horse. In a derived `class CD`, the implementation of `B` needs to be changed. The programmer finds the signature of the inherited `method A` matching his needs and calls `A` as part of the redefinition of `B`. The outcome of a previously correct dispatching call on `B` in `C` for a polymorphic variable of `class C` holding a reference to an object of `class CD` now causes infinite recursion between the redefined `method B` and the inherited `method A` of `class CD`.

This vulnerability is not restricted to the example above, but can happen whenever the design calls for multiple services converging to a single implementation.

6.43.2 Related coding guidelines

Ada Quality and Style Guide [1]: 9.4.1

6.43.3 Mechanism of failure

The mechanism is the intrinsic call semantics of the language. If the call semantics demand dispatching for nested method calls, the failure scenario is guaranteed. While the example above is tractable, the infinite recursion can involve multiple objects along a reference chain and, thus, it becomes quickly undecidable whether such a situation exists or not. Even for simple cases, avoidance requires knowledge about the implementation of all called methods inherited from superclasses and needs to apply this knowledge transitively. Such a requirement is diametrically opposed to fundamental software engineering axioms.

It has been shown that released libraries have contained many instances of infinite recursions.

Malicious exploit of the vulnerability adds a subclass that contains this infinite recursion conditionally on some trigger value. The recursion can be sufficiently obscured so that no analysis tool or reviewer can detect it with any certainty. The system can then be caused to fault with a stack overflow anytime this trigger is used. The vulnerability can thus be used for Denial-of-Service attacks.

6.43.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that demand or allow dispatching for calls within dispatching operations.

6.43.5 Avoiding the vulnerability or mitigating its effects

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

- Enforce a principle that, even across class hierarchies, converging services use a consistent implementation;
- Agree on and document a redispatch hierarchy within groups of methods, such as initializers or constructors, and use it consistently throughout the class hierarchy;
- Avoid dispatching calls in methods where possible. See upcast consequences in 6.44 Polymorphic Variables [BKK].

6.43.6 Implications for language design and evolution

None.

6.44 Polymorphic variables [BKK]

6.44.1 Description of application vulnerability

Object-oriented languages allow polymorphic variables, in which values of different classes can be stored at different times. In most of these languages, variables are declared to be of some class, while the actual value may be of a more specialized subclass. Polymorphic variables go hand in hand with method selection at run time, when the method defined for the actual subclass of the receiving object or controlling argument is invoked. This approach is safe, as method implementation and actual type of the object match by construction. If, however, the language permits casting of the polymorphic reference to process the object as if it were of the class casted to, several vulnerabilities arise. We distinguish the following casts:

- *upcasts*, where the cast is to a superclass;
- *downcasts*, where the cast is to a subclass and a check is made that the object is indeed of the target class of the cast (or a subclass thereof) ;
- *unsafe casts*, where there is no assurance that the object is of the casted class.

Distinct vulnerabilities arise for each of these cast types.

- Upcasts are needed so that redefined methods can call upon the corresponding method of the parent class to achieve the respective portion of the needed functionality and then complete it for the extensions added by the subclass. Without calling the parent's implementation of a method in the redefined method, the private components of the parent class are inaccessible to the redefined method. Hence there is a risk that they are no longer consistent with the overall state of the object. Inversely, if the issue is avoided by inheriting rather than redefining the method for a subclass, there is the risk that the subclass-specific parts are inconsistent with the overall state of the object or even uninitialized.

- Downcasts carry the risk that the object is not of the correct class. If checked by the language, as language-defined downcasts typically are, an exception will occur in this case.

- Unsafe casts allow arbitrary breaches of safety and security similar to the breaches described in 6.11 Pointer type conversions [HFC].

Some languages also have implicit upcasts and downcasts as part of the language semantics. The same issues apply for implicit casts as for explicit casts.

6.44.2 Related coding guidelines

JSF AV Rules [24]:

67 Make all data members private

78 Virtual method and virtual destructor

94 redefinition of an inherited non-virtual function

178 Limited downcast

179 Pointer casts

185 Use C++ upcasts in place of C casts

6.44.3 Mechanism of failure

Objects left in an inconsistent state by means of an upcast and a subsequent legitimate method call of the parent class can be exploited to cause system malfunctions.

Exceptions raised by failing downcasts allow Denial-of-Service attacks. Typical scenarios include the addition of objects of some unexpected subclasses in generic containers.

Unsafe casts to classes with the needed components allow reading and modifying arbitrary memory areas. See 6.11 Pointer type conversions [HFC] for more details.

6.44.4 Applicable language characteristics

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

- Languages that have polymorphic variables, particularly object-oriented languages.

- Languages that permit upcasts, downcasts, or unsafe casts.

6.44.5 Avoiding the vulnerability or mitigating its effects

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

- Forbid the use of unsafe casts.
- When upcasting:
  - Ensure functional consistency of the subclass-specific data to the changes affected via the upcasted reference;
  - Use type invariants if provided to detect semantic violations caused by upcasts;
- Try to avoid downcasts., and where a downcast is necessary:
  - Make sure that any resulting error situation are handled;
  - Precede downcasts by an appropriate membership test as needed to avoid possible errors or exceptions.

6.44.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider forbidding unsafe casts.

6.45 Extra intrinsics [LRM]

6.45.1 Description of application vulnerability

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

6.45.2 Related coding guidelines

[None]

6.45.3 Mechanism of failure

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

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

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

6.45.4 Applicable language characteristics

This vulnerability description is intended to be applicable to any language where translators may extend the set of intrinsic procedures and where intrinsic procedures are selected ahead of application defined (or external library defined) procedures of the same signature.

6.45.5 Avoiding the vulnerability or mitigating its effects

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

- Use whatever language features are available to mark a procedure as language defined or application defined;
- Avoid using procedure signatures matching those defined by the translator as extending the standard set.

6.45.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Providing mechanisms to document whether translators can extend the set of intrinsic procedures or not;
- Providing mechanisms to document the precedence for resolving collisions;
- Providing mechanisms to mark a subprogram signature as being the intrinsic or an application provided procedure;
- Implementing a diagnostic to be issued when an application procedure matches the signature of an intrinsic procedure.

6.46 Argument passing to library functions [TRJ]

6.46.1 Description of application vulnerability

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

6.46.2 Related coding guidelines

CWE [7]: 114. Process Control

JSF AV Rules [24] 16, 18, 19, 20, 21, 22, 23, 24, and 25

MISRA C [29]: 1.3, 4.11, 21.2-21.8, and 21.10

MISRA C++ [30]: 17-0-1, 17-0-5, 18-0-2, 18-0-3, 18-0-4, 18-2-1, 18-7-1 and 27-0-1

CERT C guidelines [31]: INT03-C and STR07-C

6.46.3 Mechanism of failure

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

6.46.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that provide or use libraries that do not validate the parameters accepted by functions, methods and objects.

6.46.5 Avoiding the vulnerability or mitigating its effects

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

- Use libraries that validate any values passed to the library functions before the value is used;
- Develop wrappers around library functions that check the parameters before calling the function;
- Demonstrate statically that the parameters are never invalid using static analysis tools capable of detecting data validation routines;
- Use only libraries that are known to have been developed with consistent and validated interface requirements.

NOTE Several approaches can be taken. Some work best if used in conjunction with other approaches.

6.46.6 Implications for language design and evolution

In future language design and evolution activities, consider the following items:

- Ensuring that all library functions defined operate as intended over the specified range of input values and react in a defined manner to values that are outside the specified range.
- Defining libraries that provide the capability to validate parameters during compilation, during execution or by static analysis.
- Implementing language-defined libraries that provide the preconditions and postconditions for each call so that function arguments can be validated during compilation, execution or via other static analysis tools.

6.47 Inter-language calling [DJS]

6.47.1 Description of application vulnerability

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

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

6.47.2 Related coding guidelines

[None]

6.47.3 Mechanism of failure

When calling a function that has been developed using a language different from the calling language, the call convention and the return convention used must be considered. If these conventions are not handled correctly, there is a good chance the calling stack will be corrupted, see *6.34 Subprogram signature mismatch [OTR]*. The call convention covers how the language invokes the call; see *6.32 Passing parameters and return values [CSJ],* and how the parameters are handled.

Many languages restrict the length of identifiers, the type of characters that can be used as the first character, and the case of the characters used. In addition, modules developed in different languages or using different compilers, may map names differently, causing mistakes to be made during program build. All of these restrictions should be considered when invoking a routine written in a language other than the calling language. Otherwise, the identifiers might bind in a manner different than intended.

Character and aggregate data types require special treatment in a multi-language development environment. The data layout of all languages that are to be used must be taken into consideration; this includes padding and alignment. If these data types are not handled correctly, the data could be corrupted, the memory could be corrupted, or both may become corrupt. This can happen by writing/reading past either end of the data structure, see *6.8 Buffer boundary violation (buffer overflow) [HCB]*. For example, a Pascal `STRING` data type

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

may correspond to a C structure (to capture the length information)

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

and **not** to the C structure

```
char str [10]
```

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

Most numeric data types have counterparts across languages, but the layouts may differ and only those types that match the in the different languages be used. For example, in some implementations of C++ a

```
signed char
```

would match a Fortran

```
  integer(1)
```

and would match a Pascal

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

These correspondences can be implementation-defined, necessitating verification.

Sophisticated error-handling mechanisms, such as exception handling, often do not work across language boundaries. Consequently, very simple error reporting mechanisms are needed across such boundaries, restricting the sophisticated mechanisms for use only within the bounds of a single language.

6.47.4 Applicable language characteristics

The vulnerability is applicable to all high-level programming languages and low-level programming languages since all are susceptible to this vulnerability when used in a multi-language development environment.

6.47.5 Avoiding the vulnerability or mitigating its effects

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

— Use the inter-language methods and syntax specified by the applicable language standard(s);

NOTE For example, Fortran[16] and Ada[17] specify how to call C[18] functions.

- Understand the calling conventions of all languages and language processors used;
- For items comprising the inter-language interface:
  - Understand the data layout of all data types used;
  - Understand the return conventions of all languages used;
  - Prefer that the language in which error check occurs is the one that handles the error;
  - Avoid assuming that the language makes (or does not make) a distinction between upper case and lower case letters in identifiers;
  - Avoid using a special character as the first character in identifiers;
  - Avoid using long identifier names.

6.47.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider developing standard provisions for inter-language calling to languages most often used with the programming language under consideration.

6.48 Dynamically-linked code and self-modifying code [NYY]

6.48.1 Description of application vulnerability

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

On some platforms, and in some languages, instructions can modify other instructions in the code space. Historically self-modifying code was needed for software to overcome limitations of the hardware, such as running on a platform with very limited memory. It is now often used (or misused) to hide functionality of software and make it more difficult to reverse engineer or for specialty applications such as graphics where the algorithm is tuned at runtime to give better performance or just-in-time (JIT) compilation to replace interpreted code with compiled code. Apart from automatically-generated benign code, self-modifying code can be difficult to write correctly and even more difficult to test and maintain correctly leading to unanticipated errors.

6.48.2 Related coding guidelines

JSF AV [24] Rule: 2

6.48.3 Mechanism of failure

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

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

6.48.4 Applicable language characteristics

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

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

6.48.5 Avoiding the vulnerability or mitigating its effects

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

- Verify that the dynamically linked or shared code being used is the same as that which was tested.
- Retest the application before use when it is possible that the dynamically linked or shared code has changed.
- Do not write self-modifying code except in extremely rare instances. Most software applications should never have a requirement for self-modifying code.
- In those extremely rare instances where its use is justified, limit the amount of self-modifying code and heavily document them.

6.48.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider providing a mechanism so that a program can implicitly or explicitly check that the digital signature of a library matches the one in the compile/test environment.

6.49 Library signature [NSQ]

6.49.1 Description of application vulnerability

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

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

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

6.49.2 Related coding guidelines

MISRA C [29]: 1.1

MISRA C++ [30]: 1-0-2

6.49.3 Mechanism of failure

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

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

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

6.49.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that do not specify how to describe signatures for subprograms written in other languages.

6.49.5 Avoiding the vulnerability or mitigating its effects

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

- Use tools to create the signatures;

- Avoid using translator options or language features to reference library subprograms that do not have proper signatures.

6.49.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Providing correct linkage even in the absence of correctly specified procedure signatures; (This may be very difficult where the original source code is unavailable.)

- Providing specified means to describe the signatures of subprograms.

6.50 Unanticipated exceptions from library routines [HJW]

6.50.1 Description of application vulnerability

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

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

6.50.2 Cross reference

JSF AV [24] Rule: 208

MISRA C [29]: 4.11

MISRA C++ [30]: 15-3-1, 15-3-2, 17-0-4

Ada Quality and Style Guide [1]: 5.8 and 7.5

6.50.3 Related coding guidelines

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

Considerations of 6.36 *Ignored Error Status and Unhandled Exceptions [OYB]*, are also relevant here.

6.50.4 Applicable language characteristics

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

— Languages that can link previously developed library code (where the developer and compiler do not have access to the library source).

— Languages that permit exceptions to be thrown but do not require handlers for them.

6.50.5 Avoiding the vulnerability or mitigating its effects

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

— Wrap all library calls within a 'catch-all' exception handler (if the language supports such a construct), so that any unanticipated exceptions can be caught and handled appropriately. This wrapping may be done for each library function call or for the entire behaviour of the program, for example, having the exception handler in main for C++. However, the latter is not a complete solution, as static objects are constructed before main is entered and are destroyed after it has been exited. Consequently, MISRA C++ [30] bars class constructors and destructors from throwing exceptions (unless handled locally).

— Alternatively, use only library routines for which all possible exceptions are specified.

6.50.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

— Providing a mechanism for catching all possible exceptions (for example, a 'catch-all' handler).

— Fully defining the behaviour of the program when encountering an unhandled exception, see 6.51 Pre-processor directives [NMP].

6.51 Pre-processor directives [NMP]

6.51.1 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.

6.51.2 Related coding guidelines

Holzmann [14] rule 8

JSF AV [24] Rules: 26, 27, 28, 29, 30, 31, and 32

MISRA C [29]: 1.3, 4.9, 20.5, and 20.6

MISRA C++ [30]: 16-0-3, 16-0-4, and 16-0-5

CERT C guidelines [31]: PRE01-C, PRE02-C, PRE10-C, and PRE31-C

6.51.3 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.

6.51.4 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.

6.51.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects by not using pre-processor directives where it is possible to achieve the desired functionality without the pre-processor directives.

6.51.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Reducing or eliminating dependence on lexical-level pre-processors for essential functionality (such as conditional compilation).
- Providing capabilities to inline functions and procedure calls, to reduce the need for pre-processor macros.

6.52 Suppression of language-defined run-time checking [MXB]

6.52.1 Description of application vulnerability

Some languages provide runtime checking to detect errors that can lead to vulnerabilities, and thus prevent them. Canonical examples are bounds or length checks on array operations or null-value checks upon dereferencing pointers or references. In most cases, the reaction to a failed check is the raising of a language-defined exception.

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

6.52.2 Related coding guidelines

[None]

6.52.3 Mechanism of Failure

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

6.52.4 Applicable language characteristics

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

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

6.52.5 Avoiding the vulnerability

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

- Avoid suppressing checks at all, or restrict the suppression of checks to regions of the code that have been proven to be performance-critical;
- If the default behaviour of the compiler or the language is to suppress checks, then explicitly enable those checks;

- Where checks are suppressed, statically verify that each suppressed check cannot fail, and if language-defined checks must be suppressed, use explicit checks at appropriate places in the code to ensure that errors are detected before any processing that relies on the correct values;
- Clearly identify code sections where checks are suppressed.

6.52.6 Implications for language design and evolution

[None]

6.53 Provision of inherently unsafe operations [SKL]

6.53.1 Description of application vulnerability

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

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

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

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

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

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

The vulnerability is eminently exploitable to violate program security.

6.53.2 Related coding guidelines

[None]

6.53.3 Mechanism of Failure

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

6.53.4 Applicable language characteristics

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

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

6.53.5 Avoiding the vulnerability

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

- Restrict the suppression of compile-time checks to where the suppression is functionally essential;
- Use inherently unsafe operations only when they are functionally essential and document each usage at the site of that usage;
- Clearly identify program code that suppresses checks or uses unsafe operations to permit the focusing of review effort to examine whether the function could be performed in a safer manner;
- Use static analysis tools that detect and report the use of unsafe features.

6.53.6 Implications for language design and evolution

[None]

6.54 Obscure language features [BRS]

6.54.1 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 software may not fully understand the intent.

The consequences of the above problems are more severe if the software is to be used in trusted applications, such as safety-critical or mission-critical ones.

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

6.54.2 Related coding guidelines

JSF AV Rules [24]: 84, 86, 88, and 97

MISRA C [29]: 1.1, 10.4, 13.4, 13.6, 18.5, 21.4, 21.5, 21.6, 21.7 and 21.8

MISRA C++ [30]: 0-2-1, 2-3-1, and 12-1-1

CERT C guidelines[31]: FIO03-C, MSC05-C, MSC30-C, and MSC31-C.

ISO/IEC TR 15942:2000 [20]: 5.4.2, 5.6.2 and 5.9.3

6.54.3 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 thereby overlook problems;
- Maintainers of the code cannot fully understand the intent or the usage and could introduce problems during maintenance.

6.54.4 Applicable language characteristics

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

6.54.5 Avoiding the vulnerability or mitigating its effects

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

— Avoid the use of language features that are obscure or difficult to use, especially in combination with other difficult language features;

— Adopt coding standards that discourage use of such features or show how to use them correctly;

— (Organizations) When developing software with critically important requirements, adopt a mechanism to monitor which language features are correlated with failures during the development process and during deployment;

— (Organizations) 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;

— Use tool-based static analysis to find incorrect usage of obscure language features and to determine that features forbidden by coding standards are not used.

NOTE 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.

6.54.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

— Removing or deprecating obscure, difficult to understand, or difficult to use features;

— Providing language directives that optionally disable obscure language features;

— Providing precise descriptions of complex features in the language standard;

— Be attentive to ease of use of features.

6.55 Unspecified behaviour [BQF]

6.55.1 Description of application vulnerability

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 phrase 'unspecified behaviour' is sometimes applied to such behaviours, (language specific guidelines need to analyze and document the terms used by their respective language).

The external behaviour of a program whose source code contains one or more instances of constructs having unspecified behaviour cannot be deterministically predicted. A typical example in many languages is the order of evaluation of expressions and statements in the presence of side effects.

6.55.2 Related coding guidelines

JSF AV Rules [24]: 17, 18, 19, 20, 21, 22, 23, 24, 25

MISRA C [29]: 1.1, 1.3, 19.1, and 20.2

MISRA C++ [30]: 5-0-1, 5-2-6, 7-2-1, and 16-3-1

CERT C guidelines[31]: MSC15-C

6.55.3 Mechanism of failure

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

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

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

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

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

```
A = B;
```

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

6.55.4 Applicable language characteristics

This vulnerability is intended to be applicable to 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.

6.55.5 Avoiding the vulnerability or mitigating its effects

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

- Use language constructs that have specified behaviour;
- Use static analysis tools that identify conditions that can result in unspecified 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.;
- For situation where the order of evaluation or the number of evaluations is unspecified, use only operations with no side-effects, to avoid the vulnerability;

- When developing coding guidelines for a specific language, identify all constructs that have unspecified behaviour and, for each construct where the set of possible behaviours can vary, mandate that all alternatives are considered.

6.55.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Minimizing the amount of unspecified behaviours;
- Minimizing the number of possible behaviours for any given unspecified choice;
- Documenting what might be the difference in external effect associated with different choices.

6.56 Undefined behaviour [EWF]

6.56.1 Description of application vulnerability

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. In this case, no specific behaviour is required and the translator or runtime system is at liberty to do anything it pleases.

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

6.56.2 Related coding guidelines

JSF AV Rules [24]: 17, 18, 19, 20, 21, 22, 23, 24, 25

MISRA C [29]: 1.1, 1.3, 5.4, 18.2, 18.3, and 20.2

MISRA C++ [30]: 2-13-1, 5-2-2, 16-2-4, and 16-2-5

CERT C Secure Coding guidelines [31]: MSC15-C.

6.56.3 Mechanism of failure

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. Hence, the behaviour of the program can be surprising to the programmer and the user and can result in destructive malfunctions.

6.56.4 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.

6.56.5 Avoiding the vulnerability or mitigating its effects

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

- Ensure that undefined language constructs are not used;

- Ensure that a use of a construct having undefined behaviour does not operate within the domain in which the behaviour is undefined;

  NOTE 1 When it is not possible to completely verify the domain of operation during translation a runtime check might need to be performed.

- (Organizations) When developing coding guidelines for a specific language, document all constructs that have undefined behaviour;

  NOTE 2 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).

- Use static analysis tools that identify conditions that can result in undefined behaviour;

- When developing coding guidelines for a specific language all constructs that have undefined behaviour, document for each construct the situations where the set of possible behaviours can vary.

6.56.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Minimizing undefined behaviours to the extent possible and practical;

- Enumerating all cases of undefined behaviour;

- Providing mechanisms that permit the disabling or diagnosing of constructs that may produce undefined behaviour.

6.57 Implementation-defined behaviour [FAB]

6.57.1 Description of application vulnerability

Language specifications do not always fully define the behaviour of a construct, and thus leave compiler implementations to decide how the construct will operate. 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.

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.

6.57.2 Related coding guidelines

JSF AV Rules [24]: 17, 18, 19, 20, 21, 22, 23, 24, 25

MISRA C [29]: 1.1, 1.3, 5.4, 18.2, 18.3, and 20.2

MISRA C++ [30]: 5-2-9, 5-3-3, 7-3-2, and 9-5-1

CERT Secure C coding guidelines [31]: MSC15-C

ISO/IEC TR 15942:2000 [20]: 5.9

Ada Quality and Style Guide [1]: 7.1.5 and 7.1.6

6.57.3 Mechanism of failure

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 `pragma` 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.

6.57.4 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 are not required to provide a mechanism for controlling implementation-defined behaviour.

6.57.5 Avoiding the vulnerability or mitigating its effects

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

— 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;

— Use a language implementation whose implementation-defined behaviours are within an acceptable subset of all implementation-defined behaviours;

NOTE The subset is acceptable if 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 the use of constructs that have implementation-defined behaviour, disallow all uses in which the variations of possible behaviours can produce undesirable results;

— Verify code behaviour using at least two different compilers with two different technologies.

6.57.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Providing a list of implementation-defined behaviours for portability guidelines for a specific language;
- Enumerating all cases of implementation-defined behaviour;
- Providing language directives that optionally disable language features that have implementation-defined behaviours.

6.58 Deprecated language features [MEM]

6.58.1 Description of application vulnerability

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 should not 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.

6.58.2 Related coding guidelines

JSF AV Rules [24]: 8 and 11

MISRA C [29]: 1.1 and 4.2

MISRA C++ [30]: 1-0-1, 2-3-1, 2-5-1, 2-7-1, 5-2-4, and 18-0-2

Ada Quality and Style Guide [1]: 7.1.1

6.58.3 Mechanism of failure

Ideally all code conforms to the current standard for the respective language. In reality, 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 from the language architects 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 such features can lead to problems when others are assigned the task of debugging or modifying the code containing those features.

6.58.4 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 de facto standards;
- All languages that evolve over time and as such could potentially have deprecated features at some point.

6.58.5 Avoiding the vulnerability or mitigating its effects

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

- Adhere to the latest published standard for which a suitable compiler and development environment is available;
- Use multiple compilers and other static analysis tools to help identify and eliminate deprecated features;
- Avoid the use of deprecated features of the language;
- Stay abreast of language discussions in language user groups and standards groups.

6.58.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Removing obscure language features for which there are commonly used alternatives;
- Removing 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 or project-specific coding standards;
- Providing language mechanisms that optionally disable deprecated language features.

6.59 Concurrency – Activation [CGA]

6.59.1 Description of application vulnerability

A vulnerability can occur if an attempt has been made to activate a thread, but a programming error or the lack of some resource prevents the activation from completing. The activating thread may not have sufficient visibility or awareness into the execution of the activated thread to determine if the activation has been successful. The unrecognized activation failure can cause a protocol failure in the activating thread or in other threads that rely upon some action by a not yet activated thread. This may cause the other thread(s) to wait forever for some event from the not yet activated thread, or may cause an unhandled event or exception in the other threads.

6.59.2 Related coding guidelines

CWE [7]: 364. Signal Handler Race Condition

See also Hoare C.A.R [12], Holzmann G., [15], Larsen, Peterson, and Wang [26], and the Ravenscar Tasking Profile, specified in D.13 of ISO/IEC 8652:2012 Information Technology – Programming Languages – Ada [17]

6.59.3 Mechanism of Failure

The context of the problem is that thread activation occurs for all threads except the main thread by program steps of another thread. The activation of each thread requires that dedicated resources be created for that thread, such as a thread stack, thread attributes, and communication ports.

If all activation in a program is static activation, static analysis can determine exactly how many threads will be created and how much resource, in terms of memory, processors, CPU cycles, priority ranges and inter-thread communication structures, will be needed by the executing program before the program begins. If the activation of any thread in the program is dynamic activation, then runtime queries are required to determine if all threads successfully started.

If insufficient resources remain when the activation attempt is made, the activation will fail. Similarly, if there is a program error in the activated thread or if the activated thread detects an error that causes it to terminate before beginning its main work, then it may appear to have failed during activation. When *static task activation* occurs, resources have been preallocated, so activation failure because of a lack of resources will not occur. However, errors may occur for reasons other than resource allocation and the results of an activation failure will be similar. If the activation is dynamic activation, the resources are allocation from the dynamic computational resources such as dynamic memory (heap).

If the activating thread waits for each activated thread, then the activating thread will likely be notified of activation failures (if the particular construct or capability supports activation failure notification) and can be programmed to take alternate action. If notification occurs but alternate action is not programmed, then the program will execute erroneously. If the activating thread is loosely coupled with one or more not yet activated threads, and the activating thread does not receive notification of a failure to activate, then it may wait indefinitely for the not yet activated thread to do its work or may make wrong calculations because of incomplete data.

Activation of a single thread is a special case of activations of collections of threads simultaneously. This paradigm (activation of collections of threads) can be used in languages that parallelise calculations and create anonymous threads to execute each slice of data. In such situations, the activating thread is unlikely to individually monitor each activated thread, so a failure of some to activate without explicit notification to the activating thread can result in erroneous calculations.

If the rest of the application is unaware that an activation has failed, an incorrect execution of the application algorithm may occur, such as deadlock of threads waiting for the activated thread, or possibly causing errors or incorrect calculations.

6.59.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages that permit concurrency within the language, or to languages that use support libraries and operating systems (such as POSIX or Windows) that provide concurrency control mechanisms. In essence, all traditional languages on fully functional operating systems (such as POSIX-compliant OS or Windows) can access the OS-provided mechanisms.

6.59.5 Avoiding the vulnerability or mitigating its effects

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

- Always check error return codes on operating system commands, library provided or language thread activation mechanisms before processing any other parameters or attempting to access any activated threads;
- Use static analysis tools to verify that return codes are checked;
- Handle errors and exceptions that occur on activation;
- Create explicit synchronization protocols, to ensure that all activations have occurred before beginning the parallel algorithm, if not provided by the language or by the threading subsystem;
- Use programming language provided features or thread-library provided features that couple the activated thread with the activating thread to detect activation errors so that errors can be reported and recovery made;
- Use static thread activation in preference to dynamic thread activation so that static analysis can guarantee correct activation of threads.

6.59.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Including automatic synchronization of thread initiation as part of the concurrency model.
- Providing a mechanism permitting query of activation success.

6.60 Concurrency – Directed termination [CGT]

6.60.1 Description of application vulnerability

This discussion is associated with the effects of unsuccessful or late termination of a thread. For a discussion of premature termination, see 6.63 Lock Protocol Errors [CGM].

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

When a thread is working cooperatively with other threads and is directed to terminate, there are a number of error situations that may occur that can lead to compromise of the system. The termination directing thread may request that one or more thread abort(s) occur, but the terminating thread(s) may not be in a state such that the termination can occur, may ignore the direction, or may take longer to abort or terminate than the application can tolerate. In any case, on most systems, the thread will not terminate until it is next scheduled for execution.

Unexpectedly delayed termination or the consumption of resources by the termination itself may cause a failure to meet deadlines, which, in turn, may lead to other failures.

6.60.2 Related coding guidelines

CWE[7]: 364. Signal Handler Race Condition

See also Hoare C.A.R [12], Holzmann G., [15], Larsen, Peterson, and Wang [26], and the Ravenscar Tasking Profile, specified in D.13 of ISO/IEC 8652:2012 Information Technology – Programming Languages – Ada [17] and the Guide to using the Ravenscar tasking profile [21].

6.60.3 Mechanism of failure

The abort of a thread may not happen if a thread is in an abort-deferred region and does not leave that region (for whatever reason) after the abort directive is given. Similarly, if abort is implemented as an event sent to a thread and it is permitted to ignore such events, then the abort will not be obeyed.

The termination of a thread may not happen if the thread ignores the directive to terminate, or if the finalization of the thread to be terminated does not complete.

If the termination directing thread continues on the false assumption that termination has completed, then arbitrary failure can occur, up to and including unbounded behaviours, see 6.56 Undefined behaviour [EWF].

6.60.4 Applicable language characteristics

This vulnerability is intended to be applicable to all languages that permit concurrency within the language, or support libraries and operating systems (such as POSIX-compliant or Windows operating systems) that provide hooks for concurrency control. In essence, all traditional languages on fully functional operating systems (such as POSIX-compliant OS or Windows) can access the OS-provided mechanisms.

6.60.5 Avoiding the vulnerability or mitigating its effect

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

- Use mechanisms of the language or system to determine that aborted threads or threads directed to terminate have successfully terminated;

NOTE These mechanisms may include direct communication, runtime-level checks, explicit dependency relationships, or progress counters in shared communication code to verify progress.

- Provide mechanisms to detect and/or recover from failed termination;
- Use static analysis techniques, such as CSP or model-checking to show that thread termination is safely handled;
- Where appropriate, use scheduling models where threads never terminate;
- Where possible do not use forced termination.

6.60.6 Implications for language design and evolution

In future language design and evolution activities, programming language designers should consider providing a mechanism (either a language mechanism or a service call) to signal either another thread or an entity that can be queried by other threads when a thread terminates.

6.61 Concurrent data access [CGX]

6.61.1 Description of application vulnerability

Concurrency presents a significant challenge to program correctly and has a large number of possible ways for failures to occur, quite a few known attack vectors, and many possible but undiscovered attack vectors. In particular, data visible from more than one thread and not protected by a sequential access lock can be corrupted by out-of-order accesses. This, in turn, can lead to incorrect computation, premature program termination, livelock, or system corruption.

6.61.2 Related coding guidelines

CWE [7]:

214. Information Exposure Through Process Environment

362. Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

366. Race Condition Within a Thread

368. Context Switching Race Conditions

413. Improper Resource Locking

764. Multiple Locks of a Critical Resource

765. Multiple Unlocks of a Critical Resource

820. Missing Synchronization

821. Incorrect Synchronization

See also Burns A. and Wellings A. [5], and Hoare C.A.R. [12]

6.61.3 Mechanism of failure

Shared data can be monitored or updated directly by more than one thread, possibly circumventing any access lock protocol in operation. Some concurrent programs do not use access lock mechanisms but rely upon other mechanisms such as timing or other program state to determine if shared data can be read or updated by a thread. Regardless, direct visibility to shared data permits direct access to such data concurrently. Arbitrary behaviour of any kind can result if such actions are not performed atomically.

6.61.4 Applicable language characteristics

The vulnerability is intended to be applicable to all languages that provide concurrent execution and data sharing, whether as part of the language or by use of underlying operation system facilities, including facilities such as event handlers and interrupt handlers.

6.61.5 Avoiding the vulnerability or mitigating its effect

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

- Place all data in memory accessible to only one thread at a time;
- Use languages and those language features that provide a robust synchronization mechanism to protect against data corruption;
- Use operating system primitives, such as the POSIX locking primitives, for synchronization, to develop a protocol following the principles of the Ada `protected` or Java `synchronized` paradigms.
- Where order of access is important for correctness, implement blocking and releasing paradigms, or provide a test in the same protected region to check for correct order and generate errors if the test fails;
- Where facilities for atomic access exist, use such mechanisms to prevent simultaneous access. See also 6.63 Lock protocol errors [CGM].

6.61.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- For languages that do not presently consider concurrency, consider creating primitives that let applications specify regions of sequential access to data;

  NOTE Mechanisms such as protected regions, Hoare monitors or synchronous message passing between threads result in significantly fewer resource access mistakes in a program.

- Providing the possibility of selecting alternative concurrency models that support static analysis, such as one of the models that are known to have safe properties. For examples, see Dionisio [8], and Einarsson [9].

6.62 Concurrency – Premature termination [CGS]

6.62.1 Description of application vulnerability

When a thread is working cooperatively with other threads and terminates prematurely for whatever reason but unknown to other threads, then the portion of the interaction protocol between the terminated thread and other threads is damaged. This may result in:

- indefinite blocking of the other threads as they wait for the terminated thread if the interaction protocol was synchronous;
- other threads receiving wrong or incomplete results if the interaction was asynchronous;

- deadlock if all other threads were depending upon the terminated thread for some aspect of their computation before continuing.

6.62.2 Related coding guidelines

CWE [7]: 364. Signal Handler Race Condition

See also Hoare C.A.R., [12] Larsen, Peterson, and Wang, [26], *"The Ravenscar Tasking Profile*, specified in clause D.13 of ISO/IEC 8652 D.1[17], and Guide to using the Ravenscar tasking profile [21]..

6.62.3 Mechanism of failure

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

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

If a thread terminates prematurely, threads that depend upon services from the terminated thread (in the sense of waiting exclusively for a specific action before continuing) may wait forever since held locks may be left in a locked state resulting in waiting threads never being released or messages or events expected from the terminated thread will never be received.

If a thread depends on the terminating thread and receives notification of termination, but the dependent thread ignores the termination notification, then a protocol failure will occur in the dependent thread. For asynchronous termination events, an unexpected event may cause immediate transfer of control from the execution of the dependent thread to another (possible unknown) location, resulting in corrupted objects or resources; may cause termination in the master thread, which can also cause the failure to propagate to child threads.

These conditions can result in:

- premature shutdown of the system;
- corruption or arbitrary execution of code;
- livelock;
- deadlock;

depending upon how other threads handle the termination errors.

If the thread termination is the result of an abort and the abort is immediate, there is nothing that can be done within the aborted thread to prepare data for return to the master thread, except possibly the management thread (or operating system) notifying other threads that the event occurred. If the aborted thread was holding resources or performing active updates when aborted, then any direct access by other threads to such locks, resources or memory may result in corruption of those threads or of the complete system, up to and including arbitrary code execution.

Static analysis techniques, specifically model checking, can be used to statically verify several concurrency properties, including correct data access and termination protocols.

6.62.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages that permit concurrency within the language, or support libraries and operating systems (such as POSIX-compliant or Windows operating systems) that provide hooks for concurrency control.

6.62.5 Avoiding the vulnerability or mitigating its effect

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

- Use concurrency mechanisms that are known to be robust;
- If possible, do not force immediate termination externally;
- At appropriate times use mechanisms of the language or system to determine that necessary threads are still operating;

  NOTE Such mechanisms can be direct communication, runtime-level checks, explicit dependency relationships, or progress counters in shared communication code to verify progress.

- Handle events and exceptions from termination;
- Provide manager threads to monitor progress and to organize and recover from improper terminations or abortions of threads.;
- Use static analysis techniques, such as model checking, to show that thread termination is safely handled.

6.62.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Providing a mechanism to preclude the abort of a thread from another thread during critical pieces of code. Some languages (for example, Ada or Real-Time Java) provide a notion of an abort-deferred region;
- Providing a mechanism to signal another thread (or an entity that can be queried by other threads) when a thread terminates;
- Providing a mechanism that, within critical pieces of code, defers the delivery of asynchronous exceptions or asynchronous transfers of control.

6.63 Lock protocol errors [CGM]

6.63.1 Description of application vulnerability

Concurrent programs use protocols to control:

- The way that threads interact with each other,
- How to schedule the relative rates of progress,
- How threads participate in the generation and consumption of data,

- The allocation of threads to the various roles,
- The preservation of data integrity,
- The detection and correction of incorrect operations.

When protocols are not correct, or when a vulnerability lets an exploit destroy a protocol, then the concurrent portions fail to work co-operatively and the system behaves incorrectly.

This vulnerability is related to *6.61 Concurrent data access [CGX]*, which discusses situations where the protocol to control access to resources is explicitly visible to the participating partners and makes use of visible shared resources. In comparison, this vulnerability discusses scenarios where such resources are protected by protocols and considers ways that the protocol itself may be misused.

6.63.2 Related coding guidelines

CWE [7]:

413. Improper Resource Locking

414. Missing Lock Check

609. Double Checked Locking

667. Improper Locking

821. Incorrect Synchronization

833. Deadlock

See also Hoare [12], Larsen et al [26], the Ravenscar Tasking Profile, specified in clause D.13 of ISO/IEC 8652 Information technology – Programming Languages – Ada [17], and the Guide to using the Ravenscar tasking profile [21].

6.63.3 Mechanism of failure

Threads use locks and protocols to schedule their work, control access to resources, exchange data, and to effect communication with each other. Protocol errors occur when the expected rules for co-operation are not followed, or when the order of lock acquisitions and release causes the threads to quit working together. These errors can be as a result of:

- deliberate termination of one or more threads participating in the protocol,
- disruption of messages or interactions in the protocol,
- errors or exceptions raised in threads participating in the protocol,
- errors in the programming of one or more threads participating in the protocol.

In such situations, there are a number of possible consequences:

- *deadlock*, where some sets (possibly all) of threads eventually stop computing as they wait for results from another thread, and no further progress in the system is made;
- *livelock*, where one or more threads commandeer all of the computing resource and effectively lock out the other portions, no further progress in the system is made;

- data may be corrupted or lack currency (timeliness);
- one or more threads detect an error associated with the protocol and terminate prematurely, leaving the protocol in an unrecoverable state.

The potential damage from attacks on protocols depends upon the nature of the system using the protocol and the protocol itself. Self-contained systems using private protocols can be disrupted, but it is highly unlikely that predetermined executions (including arbitrary code execution) can be obtained. On the other extreme, threads communicating openly between systems using well-documented protocols can be disrupted in any arbitrary fashion with effects such as the destruction of system resources (such as a database), the generation of wrong but plausible data, or arbitrary code execution. In fact, many documented client-server based attacks consist of some abuse of a protocol such as SQL transactions.

6.63.4 Applicable language characteristics

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

- Languages that support concurrency directly;
- Languages that permit calls to operating system primitives to obtain concurrent behaviours;
- Languages that permit IO or other interaction with external devices or services;
- Languages that support interrupt handling directly or indirectly (via the operating system).

6.63.5 Avoiding the vulnerability or mitigating its effect

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

- Consider the use of synchronous protocols, such as defined by CSP, Petri Nets or by the Ada rendezvous protocol since these can be statically shown to be free from protocol errors such as deadlock and livelock;
- Consider the use of simple asynchronous protocols that exclusively use concurrent threads and protected regions, such as defined by the Ravenscar Tasking Profile[17][21], which can also be shown statically to have correct behaviour using model checking technologies, as shown by Asplund and Lundqvist [28];
- When static verification is not possible, consider the use of detection and recovery techniques using simple mechanisms and protocols that can be verified independently from the main concurrency environment. Watchdog timers coupled with checkpoints constitute one such approach;
- Use high-level synchronization paradigms, for example monitors, rendezvous, or critical regions;
- Design the architecture of the application to ensure that some threads or tasks never block, and can be available for detection of concurrency error conditions and for recovery initiation;
- Use model checkers to model the concurrent behaviour of the complete application and check for states where progress fails;
- Place all locks and releases in the same subprograms, and ensure that the order of locking and releasing of multiple locks is correct;
- On a single processor, make use of a scheduling regime based on ceiling protocols with delays prohibited while priority is elevated, this is guaranteed to be deadlock free (if the tasks and resources are assigned the correct priorities);

- For multicore systems, consider assigning all interacting tasks to the same CPU then treat each such group as a separate process;
- Minimize the use of dynamic priorities and dynamic ceiling priorities (so that the static values can be verified).

6.63.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Raising the level of abstraction for concurrency services;
- Designing concurrency services that help to avoid typical failures such as deadlock;
- Providing services or mechanisms to detect and recover from protocol lock failures;

6.64 Reliance on external format strings [SHL]

6.64.1 Description of application vulnerability

Many languages use format string to control how output is generated or input acquired. If the contents of the format string can be influenced by external data, there is an opportunity for an attacker to gain access to what was intended to be private data, to execute arbitrary code, or to cause resource exhaustion or buffer overrun. Even without an attacker, mistakes in format strings may cause serious program errors.

6.64.2 Related coding guidelines

CWE [7]: 134. Uncontrolled Format String

6.64.3 Mechanism of failure

Format strings are parameters of input or output functions. They consist of fixed text and control sequences that are associated with other parameters of the function, and which control how the parameters are displayed or loaded.

There are several mechanisms relating to format strings that can lead to safety and security problems.

1. For an output function, the format string controls what is written to an output channel (file or printer) or a character buffer. In the latter case, particularly there is the possibility of buffer overrun, when the format string causes data to be written beyond the end of the buffer. In most languages that provide I/O control using format strings, it is possible for control sequences in the format string to control the size of the value written (e.g. the control sequence `%6d` in C based languages means write an integer value in a 6 character field, padding with spaces if necessary). If the size of the target field is accidentally or maliciously increased (say to `%6000d`) at runtime then buffer overrun or resource exhaustion can occur.

2. As the format string controls what is written to an output channel, if an attacker can influence the format string, then they can control what is written to a buffer, which could include executable code. If the attacker can then cause corruption of the program stack, it may be possible to execute this code.

3. As the format string is interpreted at run-time and expects to find a parameter for each control sequence, if the format string has more control sequences than supplied parameters, it is likely that additional values will be read off the stack. This can lead to values being output that can leak sensitive information.

4. Format strings are able to modify data values passed for output, with the result that values generated by the application can be arbitrarily changed, with serious consequences for applications that rely upon the output. Again, using C-based languages as an example, the `%n` control sequence means write the number

of characters output so far by this function to the value pointed to by the associated parameter. If the function is intended to output the value of an object whose address is supplied by a pointer, and the control sequence `%n` is added to apply to the object, then the object is not output but is modified to the number of bytes output so far.

5. The programmer rarely intends for a format string to be user controlled. However, this weakness frequently occurs in code that reads log messages from a file. Such messages may safely be output using a format string that is interpreted as ‘output a string’, but it is not unknown for the programmer to omit the format string and use the message to be output as the format string, expecting it to consist solely of literal text. If the message has been corrupted, so that it includes control sequences, any of the issues mentioned above may occur.

6.64.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages that support format strings for input/output functions.

6.64.5 Avoiding the vulnerability or mitigating its effects

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

- Ensure that all format string functions are passed as static string which cannot be controlled by the user and that the proper number of arguments is always sent to that function;
- Always supply an expected format string, even if it is the apparently redundant ‘write a string’;
- Never let a non-static text string be output as the format string;
- Ensure all control sequences used to format I/O match the associated parameter.

6.64.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider mechanisms to ensure that all format strings are verified to be correct in regard to the associated arguments or parameters.

6.65 Modifying Constants [UJO]

6.65.1 Description of application vulnerability

Many programming languages allow the user to specify some declared entity to be `constant`. The `constant` qualification assists in static verification and optimization of the code, and hence is very useful.

However, some of these languages allow alteration of the value of this entity in some cases after all. The semantics then range from legitimate and deterministic behaviour to implementation-defined or undefined behaviour. Often, the alterations are performed by means of indirection.

6.65.2 Related coding guidelines

CERT C guidelines [31]: DCL52-CPP , ES.50, EXP 40-C, EXP55-CPP, EXP05-C

MISRA C [29]: 11.8

MISRA C++ [30]: 5.2.5, 7-1-1, 9-3-3

6.65.3 Mechanism of failure

In code reviews and manual code inspections, users tend to rely on the belief that an entity declared to be constant does not change its value during the execution of the program (regardless of the exact semantics of the language). The initializing value is taken to be its value throughout the execution. For example, the upper bound of a ring buffer array might be declared as a constant. If, however, the value can be changed during the execution, the belief in immutability can be falsified. In the example, after changing the upper-bound constant, insufficiently large buffer allocations or out-of-bounds buffer accesses, seemingly checked against the *constant* upper bound, may occur.

Even the well-meant alteration of constants is very risky if the language permits optimizations based on the known initial value of the constant entity. The optimization *constant propagation* may replace uses of the constant by its initializing value. The alteration of the value at run-time then has no effect on this use of the constant, while it changes other uses of the constant where constant propagation did not take place. Moreover, different compilers or even the same compiler under different switch setting can optimize different uses of the constant differently, leading to non-deterministic executions that often result in dangerous malfunctions.

The vulnerability can be exploited if the modification of constants is known to the attacker and the code that modifies the constant can be triggered by the attacker.

The vulnerability may be difficult to detect if levels of indirection are involved in the modification of the constant.

6.65.4 Applicable language characteristics

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

Languages that allow the specification of an entity to be *constant* and, at the same time, legitimize or tolerate changes of its value.

6.65.5 Avoiding the vulnerability or mitigating its effects

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

- Qualify entities that are not changed within their scope as constants;
- Do not change the value of entities declared to be constant;
- Do not create references or pointers to entities declared to be constant since this includes passing constants as actual parameters by reference, unless immutability of the formal parameter is ensured;
- Use static analysis tools that detect the alteration of constant entities.

6.65.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider:

- Avoiding language constructs that allow the modification of constant entities;
- Ensuring that the property to be immutable cannot be changed by language operations such as assignment or conversion.

7 Application vulnerabilities

7.1 General

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

- a summary of the vulnerability;

- typical mechanisms of failure;
- techniques that programmers can use to avoid the vulnerability.

These vulnerabilities are application-related rather than language-related. They are written in a language-independent manner, and there are no corresponding sections in the language-specific Parts.

7.2 Unrestricted file upload [CBF]

7.2.1 Description of application vulnerability

A first step often used in an attack is to get an executable developed by the attacker loaded on the system under attack. Then the attack only needs to execute this code. Many times, this first step is accomplished by unrestricted file upload. In many of these attacks, the malicious code can obtain the same privilege of access as the application, or even administrator privilege.

7.2.2 Related coding guidelines

CWE [7]: 434. Unrestricted Upload of File with Dangerous Type

7.2.3 Mechanism of failure

There are several failures associated with an uploaded file:

- Executing arbitrary code;
- Phishing page added to a website;
- Defacing a website;
- Creating a vulnerability for other attacks;
- Browsing the file system;
- Creating a denial of service;
- Uploading a malicious executable to a server, which could be executed with administrator privilege.

7.2.4 Avoiding the vulnerability or mitigating its effects

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

- Allow only certain file extensions;
- Disallow certain file extensions;
- Use a utility to check the type of the file;
- Check the content-type in the header information of all files that are uploaded;

  Note 1 The purpose of the content-type field is to describe the data contained in the body completely enough that the receiving agent can pick an appropriate agent or mechanism to present the data to the user, or otherwise deal with the data in an appropriate manner.

- Use a dedicated location, which does not have execution privileges, to store and validate uploaded files, and then serve these files dynamically;

- Require a unique file extension (named by the application developer), so only the intended type of the file is used for further processing. Each upload facility of an application could handle a unique file type;
- Remove all Unicode characters and all ASCII control characters [4] from the filename and its extension;
- Set a limit for the filename length; including the file extension. In an *NTFS* (New Technology File System) partition, usually a limit of 255 characters, without path information will suffice;
- Set upper and lower limits on file size. Setting these limits can help in denial of service attacks.

NOTE 2 All of the above have some shortcomings, for example, a GIF (.gif) file's free-form comment field might not be amenable to a sanity check of the file's contents. An attacker can hide code in a file segment that will still be executed by the application or server. In many cases it will take a combination of the techniques from the above list to avoid this vulnerability.

7.3 Download of code without integrity check [DLB]

7.3.1 Description of application vulnerability

Some applications download source code or executables from a remote, and implicitly trusted, location (such as the application author) and use the source code or invoke the executables without sufficiently verifying the integrity of the downloaded files.

7.3.2 Related coding guidelines

CWE [7]: 494. Download of Code Without Integrity Check

7.3.3 Mechanism of failure

An attacker can execute malicious code by compromising the host server used to download code or executables, performing DNS spoofing, or modifying the code in transit.

7.3.4 Avoiding the vulnerability or mitigating its effects

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

- Perform proper forward and reverse DNS lookups to detect DNS spoofing. Encrypt the code with a reliable encryption scheme before transmission;

  Note 1 This is only a partial solution since it will not prevent target code from being modified on the hosting site or in transit.

- Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid;

  Note 2 Specifically, it can be helpful to use tools or frameworks to perform integrity checking on the transmitted code.

- If providing code that is to be downloaded, such as for automatic updates of software, then use cryptographic signatures for the code and document that download clients are to verify the signatures.

7.4 Executing or loading untrusted code [XYS]

7.4.1 Description of application vulnerability

Executing commands or loading libraries from an untrusted source or in an untrusted environment can cause an application to execute malicious commands (and payloads) on behalf of an attacker.

7.4.2 Related coding guidelines

CWE[7]:

114. Process Control

306. Missing Authentication for Critical Function

CERT C guidelines [31]: PRE09-C, ENV02-C, and ENV03-C

7.4.3 Mechanism of failure

Process control vulnerabilities take two forms:

- An attacker can change the command that the program executes so that the attacker explicitly controls what the command is;
- An attacker can change the environment in which the command executes so that the attacker implicitly controls what the command means.

Considering only the first scenario, the possibility that an attacker may be able to control the command that is executed, process control vulnerabilities occur when:

- Data enters the application from a source that is not trusted;
- The data is used as or as part of a string representing a command that is executed by the application;
- By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

7.4.4 Avoiding the vulnerability or mitigating its effects

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

- Ensure that libraries that are loaded are well understood and come from a trusted source with a digital signature, since the application can execute code contained in native libraries, which often contain calls that are susceptible to other security problems, such as buffer overflows or command injection;
- Validate all native libraries;
- Determine if the application requires the use of the native library since it can be very difficult to determine what these libraries actually do, and the potential for malicious code is high;
- Validate all input to native calls for content and length to help prevent buffer overflow attacks;
- If the native library does not come from a trusted source, review the source code of the library and build the library from the reviewed source before using it.

  NOTE Rebuilding from source code may require escrow on the source code for proprietary software.

7.5 Inclusion of functionality from untrusted control sphere [DHU]

7.5.1 Description of application vulnerability

The software imports, requires, or includes executable functionality (such as a library) from a source that is unknown to the user, unexpected or otherwise. Any call or use of the included functionally can result in unexpected behaviour, up to and including arbitrary execution.

7.5.2 Related coding guidelines

CWE [7]]:

98. Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')

829. Inclusion of Functionality from Untrusted Control Sphere

7.5.3 Mechanism of failure

When including third-party functionality, such as a web widget, library, or other source of functionality, the software must effectively trust that functionality. Without sufficient protection mechanisms, the functionality could be malicious in nature (either by coming from an untrusted source, being spoofed, or being modified in transit from a trusted source). The functionality might also contain its own weaknesses or grant access to additional functionality and state information that was intended to be kept private to the base system, such as system state information, sensitive application data, or the DOM of a web application.

This might lead to many different consequences depending on the included functionality, but some examples include injection of malware, information exposure by granting excessive privileges or permissions to the untrusted functionality, DOM-based XSS vulnerabilities, stealing user's cookies, or open redirect to malware.

7.5.4 Avoiding the vulnerability or mitigating its effects

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

- Use a vetted library or framework that does not allow this weakness to occur or provide constructs that make this weakness easier to avoid;
- When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs;
- For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602 [7] as described in 7.14 Authentication logic error [XZO], 7.7 Cross-site scripting [XYT], and 7.9 Injection [RST].

  NOTE For example, ID 1 could map to "inbox.txt" and ID 2 could map to "profile.txt". Features such as the ESAPI AccessReferenceMap provide this capability.

7.6 Use of unchecked data from an uncontrolled or tainted source [EFS]

7.6.1 Description of application vulnerability

This vulnerability covers a general class of behaviours, the identification of which is referred to as ‘taint analysis’.

Whenever a program gets data from an external source, there is a possibility that that data may have been tampered with by an attacker attempting to induce the program into performing some damaging action, or may have been corrupted accidently leading to the same result. Such data is called ‘tainted’.

The general principle is that before tainted data is used, it is checked to ensure that it is within acceptable bounds or has an appropriate structure, or otherwise can be accepted as untainted, and so safe to use.

7.6.2 Related coding guidelines

(none)

7.6.3 Mechanism of failure

The principal mechanisms of failure are:

- Use of the data in an arithmetic expression, causing the one of the problems described in clause 6;
- Use of the data in a call to a function that executes a system command;
- Use of the data in a call to a function that establishes a communications connection.

7.6.4 Avoiding the vulnerability or mitigating its effects

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

Different mechanisms of failure require different mitigations, which also may depend on how the tainted data is to be used. Developers can:

- Test potentially tainted data used in an arithmetic expression to ensure that it does not cause arithmetic overflow, divide by zero or buffer overflow;
- Check integer data used to allocate memory or other resources to ensure that the size of the integer data won't cause resource exhaustion;
- Check strings passed to system functions to ensure that they are well formed and have an expected structure.

NOTE 1 This vulnerability is described as 'data from an uncontrolled source', as a distinction may need to be drawn between data from outside the program, but which is still trustworthy, and data that comes from a source that could credibly be modified by an attacker, or otherwise corrupted.

NOTE 2 Data read from a file may be regarded as trustworthy (untainted) if the file is read-only and inside a firewall, but potentially tainted if it is from a more generally accessible location. See 7.22 Missing required cryptographic step [XZS].

7.7 Cross-site scripting [XYT]

7.7.1 Description of application vulnerability

Cross-site scripting (XSS) occurs when dynamically generated web pages display input, such as login information that is not properly validated, allowing an attacker to embed malicious scripts into the generated page and then execute the script on the machine of any user that views the site. If successful, cross-site scripting vulnerabilities can be exploited to manipulate or steal cookies, create requests that can be mistaken for those of a valid user, compromise confidential information, or execute malicious code on the end user systems for a variety of nefarious purposes.

7.7.2 Related coding guidelines

CWE [7]:

79. Failure to Preserve Web Page Structure ('Cross-site Scripting')

80. Failure to Sanitize Script-Related HTML Tags in a Web Page (Basic XSS)

81. Failure to Sanitize Directives in an Error Message Web Page

82. Failure to Sanitize Script in Attributes of IMG Tags in a Web Page

83. Failure to Sanitize Script in Attributes in a Web Page

84. Failure to Resolve Encoded URI Schemes in a Web Page

85. Doubled Character XSS Manipulations

86. Invalid Characters in Identifiers

87. Alternate XSS Syntax

7.7.3 Mechanism of failure

Cross-site scripting (XSS) vulnerabilities occur when an attacker uses a web application to send malicious code, generally JavaScript, to a different end user. When a web application uses input from a user in the output it generates without filtering it, an attacker can insert an attack in that input and the web application sends the attack to other users. The end user trusts the web application, and the attacks exploit that trust to do things that would not normally be allowed. Attackers frequently use a variety of methods to encode the malicious portion of the tag, such as using Unicode, so the request looks less suspicious to the user.

XSS attacks can generally be categorized into two categories: stored and reflected. Stored attacks are those where the injected code is permanently stored on the target servers in a database, message forum, visitor log, and so forth. Reflected attacks are those where the injected code takes another route to the victim, such as in an email message, or on some other server. When a user is tricked into clicking a link or submitting a form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. The browser then executes the code because it came from a 'trusted' server. For a reflected XSS attack to work, the victim must submit the attack to the server. This is still a very dangerous attack given the number of possible ways to trick a victim into submitting such a malicious request, including clicking a link on a malicious Web site, in an email, or in an inter-office posting.

XSS flaws are very common in web applications, as they require a great deal of developer discipline to avoid them in most applications. It is relatively easy for an attacker to find XSS vulnerabilities. Some of these vulnerabilities can be found using scanners, and some exist in older web application servers. The consequence of an XSS attack is the same regardless of whether it is stored or reflected.

The difference is in how the payload arrives at the server. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. The most severe XSS attacks involve disclosure of the user's session cookie, which allows an attacker to hijack the user's session and take over their account. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirecting the user to some other page or site, and modifying presentation of content.

Cross-site scripting (XSS) vulnerabilities occur when:

— Data enters a Web application through an untrusted source, most frequently a web request. The data is included in dynamic content that is sent to a web user without being validated for malicious code.

— The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Cross-site scripting attacks can occur wherever an untrusted user has the ability to publish content to a trusted web site. Typically, a malicious user will craft a client-side script, which — when parsed by a web browser — performs some activity (such as sending all site cookies to a given e–mail address). If the input is unchecked, this script will be loaded and run by each user visiting the web site. Since the site requesting to run the script has access to the cookies in question, the malicious script does also. There are several other possible attacks, such as running "Active X" controls (under Microsoft Internet Explorer) from sites that a user perceives as trustworthy; cookie theft is however by far the most common. All of these attacks are easily prevented by ensuring that no script tags — or for good measure, HTML tags at all — are allowed in data to be posted publicly.

Specific instances of XSS include the following.

- 'Basic' XSS involves a complete lack of cleansing of any special characters, including the most fundamental XSS elements such as "<", ">", and "&".

- A web developer displays input on an error page (such as a customized 403 Forbidden page). If an attacker can influence a victim to view/request a web page that causes an error, then the attack may be successful.

- A Web application that trusts input in the form of HTML IMG tags is potentially vulnerable to XSS attacks. Attackers can embed XSS exploits into the values for IMG attributes (such as SRC) that is streamed and then executed in a victim's browser. When the page is loaded into a user's browser, the exploit will automatically execute.

- The software does not filter "JavaScript:" or other *URI*'s (Uniform Resource Identifier) from dangerous attributes within tags, such as `onmouseover, onload, onerror`, or `style`.

- The web application fails to filter input for executable script disguised with URI encodings.

- The web application fails to filter input for executable script disguised using doubling of the involved characters.

- The software does not strip out invalid characters in the middle of tag names, schemes, and other identifiers, which are still rendered by some web browsers that ignore the characters.

- The software fails to filter alternate script syntax provided by the attacker.

Cross-site scripting attacks may occur anywhere that possibly malicious users are allowed to post unregulated material to a trusted web site for the consumption of other valid users. The most common example can be found in bulletin-board web sites that provide web-based mailing list-style functionality. The most common attack performed with cross-site scripting involves the disclosure of information stored in user cookies. In some circumstances it may be possible to run arbitrary code on a victim's computer when cross-site scripting is combined with other flaws.

7.7.4 Avoiding the vulnerability or mitigating its effects

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

- Carefully check each input parameter against a rigorous positive specification (inclusion-list) defining the specific characters and format allowed;

- Sanitize all input, not just parameters that the user is supposed to specify, but all data in the request, including hidden fields, cookies, headers, the *URL* (Uniform Resource Locator) itself, and so forth;

NOTE A common mistake that leads to continuing XSS vulnerabilities is to validate only fields that are expected to be redisplayed by the site.

- Validate all parts of the *HTTP* (Hypertext Transfer Protocol) request, including fields that were not expected to have changed in the client or fields that were anticipated for future growth;
- Where the base system is a SQL database, follow the recommendations of 7.9 Injection [RST].

7.8 URL redirection to untrusted site ('open redirect') [PYQ]

7.8.1 Description of application vulnerability

A web application accepts a user-controlled input that specifies a link to an external site, and then uses that link in a redirect without checking that the URL points to a trusted location. This simplifies phishing attacks.

7.8.2 Related coding guidelines

CWE [7] 601. URL Redirection to Untrusted Site ('Open Redirect')

7.8.3 Mechanism of failure

An http parameter may contain a URL value and could cause the web application to redirect the request to the specified URL. By modifying the URL value to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance.

7.8.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects through input validation in the following ways. They can:

- Assume all input is malicious and take appropriate action, including:
  - Use an acknowledged input validation strategy such as an inclusion list of acceptable inputs that strictly conform to specifications;
  - Either reject any input that does not strictly conform to specifications or transform it into something that does;
  - Avoid relying exclusively on searching for malicious or malformed inputs (for example, do not rely on an exclusion list);
  - Use exclusion lists for detecting potential attacks or determining which inputs are so malformed that they are rejected outright;
- Consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, `boat` may be syntactically valid because it only contains alphanumeric characters, but it is not valid if a `colour` such as `red` or `blue` was expected. Use an inclusion list of approved URLs or domains to be used for redirection.

7.9 Injection [RST]

7.9.1 Description of application vulnerability

Injection problems span a wide range of instantiations. The basic form of this weakness involves the software allowing injection of additional data in input data to alter the control flow of the process. Command injection problems are a subset of injection problems, in which the process can be tricked into calling external processes of an attacker's choice through the injection of command syntax into the input data. Multiple leading/internal/trailing special elements injected into an application through input can be used to compromise a system. As data is parsed, improperly handled multiple leading special elements may cause the process to take unexpected actions that result in an attack. Software may allow the injection of special elements that are non-typical but equivalent to typical special elements with control implications. This frequently occurs when the product has protected itself against special element injection. Software may allow inputs to be fed directly into an output file that is later processed as code, such as a library file or template. Line or section delimiters injected into an application can be used to compromise a system.

Many injection attacks involve the disclosure of important information — in terms of both data sensitivity and usefulness in further exploitation. In some cases, injectable code controls authentication; this may lead to a remote vulnerability. Injection attacks are characterized by the ability to significantly change the flow of a given process, and in some cases, to the execution of arbitrary code. Data injection attacks lead to loss of data integrity in nearly all cases as the control-plane data injected is always incidental to data recall or writing. Often the actions performed by injected control code are not logged.

SQL injection attacks are a common instantiation of injection attack, in which SQL commands are injected into input to effect the execution of predefined SQL commands. Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities. If poorly implemented SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password. If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of the SQL injection vulnerability. Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL injection attack.

Injection problems encompass a wide variety of issues — all mitigated in very different ways. The most important issue to note is that all injection problems share one common trait — they allow for the injection of control data into the user-controlled data. This means that the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism. While buffer overflows and many other flaws involve the use of some further issue to gain execution, injection problems need only for the data to be parsed. Many injection attacks involve the disclosure of important information in terms of both data sensitivity and usefulness in further exploitation. In some cases, injectable code controls authentication, this may lead to a remote vulnerability.

7.9.2 Related coding guidelines

CWE [7]:

74. Failure to Sanitize Data into a Different Plane ('Injection')

76. Failure to Resolve Equivalent Special Elements into a Different Plane

78. Failure to Sanitize Data into an OS Command (aka 'OS Command Injection')

89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

90. Failure to Sanitize Data into LDAP Queries (aka 'LDAP Injection')

91. XML Injection (aka Blind XPath Injection)

92. Custom Special Character Injection

95. Insufficient Control of Directives in Dynamically Code Evaluated Code (aka 'Eval Injection')

97. Failure to Sanitize Server-Side Includes (SSI) Within a Web Page

98. Insufficient Control of Filename for Include/Require Statement in PHP Program (aka 'PHP File Inclusion')

99. Insufficient Control of Resource Identifiers (aka 'Resource Injection')

144. Failure to Sanitize Line Delimiters

145. Failure to Sanitize Section Delimiters

161. Failure to Sanitize Multiple Leading Special Elements

163. Failure to Sanitize Multiple Trailing Special Elements

165. Failure to Sanitize Multiple Internal Special Elements

166. Failure to Handle Missing Special Element

167. Failure to Handle Additional Special Element

168. Failure to Resolve Inconsistent Special Elements

564. SQL Injection: Hibernate

CERT C guidelines [31]: FIO30-C

7.9.3 Mechanism of failure

A software system that accepts and executes input in the form of operating system commands (such as `system(), exec(), open()`) could allow an attacker with lesser privileges than the target software to execute commands with the elevated privileges of the executing process. Command injection is a common problem with wrapper programs. Often, parts of the command to be run are controllable by the end user. If a malicious user injects a character (such as a semi-colon) that delimits the end of one command and the beginning of another, he may then be able to insert an entirely new and unrelated command to do whatever he pleases.

Dynamically generating operating system commands that include user input as parameters can lead to command injection attacks. An attacker can insert operating system commands or modifiers in the user input that can cause the request to behave in an unsafe manner. Such vulnerabilities can be very dangerous and lead to data and system compromise. If no validation of the parameter to the exec command exists, an attacker can execute any command on the system the application has the privilege to access.

There are two forms of command injection vulnerabilities. An attacker can change the command that the program executes (the attacker explicitly controls what the command is). Alternatively, an attacker can change the environment in which the command executes (the attacker implicitly controls what the command means). The first scenario where an attacker explicitly controls the command that is executed can occur when:

— Data enters the application from an untrusted source;

- The data is part of a string that is executed as a command by the application;
- By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

Eval injection occurs when the software allows inputs to be fed directly into a function (such as `eval`) that dynamically evaluates and executes the input as code, usually in the same interpreted language that the product uses. Eval injection is prevalent in handler/dispatch procedures that might want to invoke a large number of functions, or set a large number of variables.

A PHP file inclusion occurs when a PHP product uses `require` or `include` statements, or equivalent statements, that use attacker-controlled data to identify code or *HTML* (HyperText Markup Language) to be directly processed by the PHP interpreter before inclusion in the script.

A resource injection issue occurs when the following two conditions are met:

- An attacker can specify the identifier used to access a system resource, for example specifying part of the name of a file to be opened or a port number to be used;
- By specifying the resource, the attacker gains a capability that would not otherwise be permitted. For example, the program may give the attacker the ability to overwrite the specified file, run with a configuration controlled by the attacker, or transmit sensitive information to a third-party server.

  NOTE Resource injection that involves resources stored on the file system goes by the name path manipulation and is reported in separate category. See 7.11 Path Traversal [EWR] description for further details of this vulnerability. Allowing user input to control resource identifiers may enable an attacker to access or modify otherwise protected system resources.

Line or section delimiters injected into an application can be used to compromise a system. As data are parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions that result in an attack. One example of a section delimiter is the boundary string in a multipart *MIME* (Multipurpose Internet Mail Extensions) message. In many cases, doubled line delimiters can serve as a section delimiter.

7.9.4 Avoiding the vulnerability or mitigating its effects

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

- Assume all input is malicious and use an appropriate combination of exclusion lists and inclusion lists to ensure only valid, expected and appropriate input is processed by the system;
- Narrowly define the set of safe characters based on the expected values of the parameter in the request;
- Anticipate that delimiters and special elements would be injected/removed/manipulated in the input vectors of their software system and program appropriate mechanisms to handle them;
- Implement SQL strings using prepared statements that bind variables;
- Use vigorous inclusion-list style checking on any user input that may be used in a SQL command;

  NOTE Rather than escape meta-characters, it is safest to disallow them entirely since the later use of data that have been entered in the database may neglect to escape meta-characters before use.

- Follow the principle of least privilege when creating user accounts to a SQL database, since if the requirements of the system indicate that users are permitted to read and modify their own data, then limit their privileges so they cannot read/write others' data;

- Assign permissions to the software system that prevents the user from accessing/opening privileged files;
- Restructure code so that there is not a need to use the `eval()` utility.

7.10 Unquoted search path or element [XZQ]

7.10.1 Description of application vulnerability

Strings injected into a software system that are not quoted can permit an attacker to execute arbitrary commands.

7.10.2 Related coding guidelines

CWE [7]: 428. Unquoted Search Path or Element

CERT C guidelines [31]: ENV04-C

7.10.3 Mechanism of failure

The mechanism of failure stems from missing quoting of strings injected into a software system. By allowing white-spaces in identifiers, an attacker could potentially execute arbitrary commands. This vulnerability covers "`C:\Program  Files`" and space-in-search-path issues. Theoretically, this could apply to other operating systems besides Windows, especially those that make it easy for spaces to be in filenames or folders names.

7.10.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects by examining strings that are to be interpreted to ensure that they do not contain constructs designed to exploit the system, such as separators.

7.11 Path traversal [EWR]

7.11.1 Description of application vulnerability

The software constructs a path that contains relative traversal sequence such as ".." or an absolute path sequence such as "/path/here." Attackers run the software in a particular directory so that the hard link or symbolic link used by the software accesses a file that the attacker has under their control. In doing this, the attacker may be able to escalate their privilege level to that of the running process.

7.11.2 Related coding guidelines

CWE [7]:

22. Path Traversal

24. Path Traversal: - '../filedir'

25. Path Traversal: '/../filedir'

26. Path Traversal: '/dir/../filename'

27. Path Traversal: 'dir/../../filename'

28. Path Traversal: '..\filename'

29. Path Traversal: '\..\filename'

30. Path Traversal: ‘\dir\..\filename’

31. Path Traversal: ‘dir\..\filename’

32. Path Traversal: ‘’…' (Triple Dot)

33. Path Traversa‘: '.’..' (Multiple Dot)

34. Path Traversa‘: '...’//'

35. Path Traversa‘: '.../..’//'

37. Path Traversal: ‘/absolute/pathname/here’

38. Path Traversal: ‘ \absolute\pathname\here’

39. Path Traversa‘: 'C:dirn’me'

40. Path Traversa‘: '\\UNC\share\na’e\' (Windows UNC Share)

61. UNIX Symbolic Link (Symlink) Following

62. UNIX Hard Link

64. Windows Shortcut Following (.LNK)

65. Windows Hard Link

CERT C guidelines [31]: FIO02-C

7.11.3 Mechanism of failure

There are two primary ways that an attacker can orchestrate an attack using path traversal. In the first, the attacker alters the path being used by the software to point to a location that the attacker has control over. Alternatively, the attacker has no control over the path, but can alter the directory structure so that the path points to a location that the attacker does have control over.

For instance, a software system that accepts input in the form of:

```
`  '..\filen’me',;
`  '\..\filen’me';
`  '/directory/../filen’me';
`  'directory/../../filen’me';
`  '..\filen’me';
`  '\..\filen’me';
`  '\directory\..\filen’me';
`  'directory\..\..\filen’me';
`  '’ ..';
`  '.’..' (multiple do ts);
`  '...’//';
`  '.../..’//'
```

without appropriate validation can allow an attacker to traverse the file system to access an arbitrary file. Note that `’..'` is ignored if the current working directory is the root directory. Some of these input forms can be used to cause problems for systems that strip out `'..'` from input in an attempt to remove relative path traversal.

There are several common ways that an attacker can point a file access to a file the attacker has under their control. A software system that accepts input such as
`'/absolute/pathname/h'rе'` ‘r `'\absolute\pathname\h're'`
without appropriate validation can also allow an attacker to traverse the file system to unintended locations or access arbitrary files. An attacker can inject a drive letter or Windows volume letter (`'C:dirn'me'`)
into a software system to potentially redirect access to an unintended location or arbitrary file. A software system that accepts input in the form of a backslash absolute path without appropriate validation can allow an attacker to traverse the file system to unintended locations or access arbitrary files. An attacker can inject a Windows UNC (Universal Naming Convention or Uniform Naming Convention) share
(`'\\UNC\share\n'me'`)
into a software system to potentially redirect access to an unintended location or arbitrary file. A software system that allows UNIX symbolic links (symlink) as part of paths whether in internal code or through user input can allow an attacker to spoof the symbolic link and traverse the file system to unintended locations or access arbitrary files. The symbolic link can permit an attacker to read/write/corrupt a file that they originally did not have permissions to access. Failure for a system to check for hard links can result in vulnerability to different types of attacks. For example, an attacker can escalate their privileges if he/she can replace a file used by a privileged program with a hard link to a sensitive file, for example, `etc/passwd`. When the process opens the file, the attacker can assume the privileges of that process.

A software system that allows Windows shortcuts (.LNK) as part of paths whether in internal code or through user input can allow an attacker to spoof the symbolic link and traverse the file system to unintended locations or access arbitrary files. The shortcut (file with the `.lnk` extension) can permit an attacker to read/write a file that they originally did not have permissions to access.

Failure for a system to check for hard links can result in vulnerability to different types of attacks. For example, an attacker can escalate their privileges if he/she can replace a file used by a privileged program with a hard link to a sensitive file (such as `etc/passwd`). When the process opens the file, the attacker can assume the privileges of that process or possibly prevent a program from accurately processing data in a software system.

A sanitizing mechanism can remove characters such as ‘.' and ‘;' which may be required for some exploits. An attacker can try to fool the sanitizing mechanism into "cleaning" data into a dangerous form. Suppose the attacker injects a ’.' inside a filename (say, `sensi.tiveFile`) and the sanitizing mechanism removes the character resulting in the valid filename, `sensitiveFile`. If the input data is now assumed to be safe, then the file may be compromised.

When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories.

Securely creating temporary files in a shared directory is error-prone and dependent on the version of the runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system.

Mitigate by converting relative paths into absolute paths and then verifying that the resulting absolute path makes sense with respect to the configuration and rights or permissions. This may include checking inclusion-lists and exclusion lists, authorized super user status, access control lists, or other fully trusted status.

7.11.4 Avoiding the vulnerability or mitigating its effects

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

— Assume all input is malicious. Attackers can insert paths into input vectors and traverse the file system;

- Use an appropriate combination of exclusion lists and inclusion lists to ensure only valid and expected input is processed by the system;
- Use sanitizers to scrub input for sensitive programs. Ensure that sanitizers work properly;

  NOTE 1 For example, a sanitizer can remove "." or ".." at a string beginning, but not in the middle of a valid file system address.

- Compare multiple attributes of the file to improve the likelihood that the file is the expected one;

  NOTE 2 Files can often be identified by other attributes in addition to the file name, for example, by comparing file ownership or creation time. Information regarding a file that has been created and closed can be stored and then used later to validate the identity of the file when it is reopened.

- Follow the principle of least privilege when assigning access rights to files;
- Deny access to a file can prevent an attacker from replacing that file with a link to a sensitive file;
- Ensure good compartmentalization in the system to provide protected areas that can be trusted;
- Restrict the use of shared directories; prefer files pulled from configuration management systems;
- Do not permit temporary files to be created in shared directories.

7.12 Resource names [HTS]

7.12.1 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 prohibits "`/?:&\*"<>|#%`"; but UNIX, Linux, and OS X operating systems allow any character except for the reserved character '`/`' to be used in a filename.

Some operating systems are case sensitive while others are not. On non-case sensitive operating systems, depending on the software being used, the same filename could be displayed, as `filename`, `Filename` or `FILENAME` and all would refer to the same file.

Some operating systems, particularly older ones, only rely on the significance of the first `n` characters of the file name. `n` can be unexpectedly small, such as the first 8 characters in the case of Win16 architectures which would cause `filename1`, `filename2` and `filename3` to all map to the same file.

Variations in the filename, named resource or external identifier being referenced can be the basis for various kinds of problems. Such mistakes or ambiguity can be unintentional, or intentional, and in either case they can be potentially exploited, if surreptitious behaviour is a goal.

7.12.2 Related coding guidelines

JSF AV Rules [24]: 46, 51, 53, 54, 55, and 56

MISRA C [29]: 1.1

CERT C guidelines [31]: MSC09-C and MSC10-C

7.12.3 Mechanism of Failure

The wrong named resource, such as a file, may be used within a program in a form that provides access to a resource that was not intended to be accessed. Attackers could exploit this situation to intentionally misdirect access of a named resource to another named resource.

7.12.4 Avoiding the vulnerability or mitigating its effects

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

- 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, so that the means that information on such characteristics as the directory separator string and methods of accessing parent directories need to be parameterized and not exist as fixed strings within a program;
- Avoid creating resource names that are longer than the guaranteed unique length of all potential target platforms;
- Avoid creating resources, which are differentiated only by the case in their names;
- Avoid all non-ASCII Unicode characters and all ASCII control characters in filenames and the extensions, as documented in the ASCII Codes Table [4].

7.13 Resource exhaustion [XZP]

7.13.1 Description of application vulnerability

The application is susceptible to generating and/or accepting an excessive number of requests that could potentially exhaust limited resources, such as memory, file system storage, database connection pool entries, or CPU. This could ultimately lead to a denial of service that could prevent any other applications from accessing these resources.

7.13.2 Related coding guidelines

CWE [7]:

400. Resource Exhaustion

7.13.3 Mechanism of failure

There are two primary failures associated with resource exhaustion. The most common result of resource exhaustion is denial of service. In some cases, an attacker or a defect may cause a system to fail in an unsafe or insecure fashion by causing an application to exhaust the available resources.

Resource exhaustion issues are generally understood but are far more difficult to prevent. Taking advantage of various entry points, an attacker could craft a wide variety of requests that would cause the site to consume resources. Database queries that take a long time to process are good *DoS* (Denial of Service) targets. An attacker would only have to write a few lines of Perl code to generate enough traffic to exceed the site's ability to keep up. This would effectively prevent authorized users from using the site at all.

Resources can be exhausted simply by ensuring that the target machine must do much more work and consume more resources to service a request than the attacker must do to initiate a request. Prevention of these attacks requires that the target system either recognizes the attack and denies that user further access for a given amount of time or uniformly throttles all requests to make it more difficult to consume resources more quickly than they can again be freed. The first of these solutions is an issue in itself, since it may allow attackers to prevent the use of the system by a particular valid user. If the attacker impersonates the valid user, he may be able to prevent the user from accessing the server in question. The second solution is simply difficult to effectively institute and even when properly done, it does not provide a full solution. It simply makes the attack require more resources on the part of the attacker.

The final concern that must be discussed about issues of resource exhaustion is that of systems which *fail open*. This means that in the event of resource consumption, the system fails in such a way that the state of the system — and possibly the security functionality of the system — are compromised. A prime example of this can be found in old switches that were vulnerable to "macof" attacks (so named for a tool). These attacks flooded a switch with random Internet Protocol (IP) and Media Access Control (MAC) address combinations, therefore exhausting the switch's cache, which held the information of which port corresponded to which MAC addresses. Once this cache was exhausted, the switch would fail in an insecure way and would begin to act simply as a hub, broadcasting all traffic on all ports and allowing for basic sniffing attacks.

7.13.4 Avoiding the vulnerability or mitigating its effects

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

- Implement throttling mechanisms into the system architecture that are capable of detecting potential denial of service attacks and throttling access when detected;

  NOTE The best protection is to limit the quantity of resources that an application can cause to be expended.

- Implement a strong authentication and access control model to deter such attacks and ensure that the authentication application is protected against denial of service attacks as much as possible;

- Limit the critical resource (such as database) access, perhaps by caching often-used result sets, to reduce the resources expended;

- Track the rate of requests received from users and blocking requests that exceed a defined rate threshold to further limit the potential for a denial-of-service attack;

- Ensure that applications have specific limits of scale placed on them, and that all failures in resource allocation cause the application to fail safely.

7.14 Authentication logic error [XZO]

7.14.1 Description of application vulnerability

The software does not properly ensure that the user has proven their identity.

7.14.2 Related coding guidelines

CWE [7]:

287. Improper Authentication

288. Authentication Bypass by Alternate Path/Channel

289. Authentication Bypass by Alternate Name

290. Authentication Bypass by Spoofing

294. Authentication Bypass by Capture-replay

301. Reflection Attack in an Authentication Protocol

302. Authentication Bypass by Assumed-Immutable Data

303. Improper Implementation of Authentication Algorithm

305. Authentication Bypass by Primary Weakness

602. Client-side Enforcement of Server-side Security

7.14.3 Mechanism of failure

There are many ways that an attacker can potentially bypass the validation of a user. Some of the ways are means of impersonating a legitimate user while others are means of bypassing the authentication mechanisms that are in place. In either case, an unauthorized user gains access to the software system.

Authentication bypass by alternate path or channel occurs when a product requires authentication, but the product has an alternate path or channel that does not require authentication. This is often seen in web applications that assume that access to a particular *CGI* (Common Gateway Interface) program can only be obtained through a "front" screen, but this problem is not just in web applications.

Authentication bypass by alternate name occurs when the software performs authentication based on the name of the resource being accessed, but there are multiple names for the resource, and not all names are checked.

Authentication bypass by capture-replay occurs when it is possible for a malicious user to sniff network traffic and bypass authentication by replaying it to the server in question to the same effect as the original message (or with minor changes). Messages sent with a capture-relay attack allow access to resources that are not otherwise accessible without proper authentication. Capture-replay attacks are common and can be difficult to defeat without cryptography. They are a subset of network injection attacks that rely on listening in on previously sent valid commands, then changing them slightly if necessary and resending the same commands to the server. Since any attacker who can listen to traffic can see sequence numbers, it is necessary to sign messages with some kind of cryptography to ensure that sequence numbers are not simply doctored along with content.

Reflection attacks capitalize on mutual authentication schemes to trick the target into revealing the secret shared between it and another valid user. In a basic mutual-authentication scheme, a secret is known to both a valid user and the server; this allows them to authenticate. In order that they may verify this shared secret without sending it plainly over the wire, they utilize a Diffie-Hellman-style scheme in which they each pick a value, then request the hash of that value as keyed by the shared secret. In a reflection attack, the attacker claims to be a valid user and requests the hash of a random value from the server. When the server returns this value and requests its own value to be hashed, the attacker opens another connection to the server. This time, the hash requested by the attacker is the value that the server requested in the first connection. When the server returns this hashed value, it is used in the first connection, authenticating the attacker successfully as the impersonated valid user.

Authentication bypass by assumed-immutable data occurs when the authentication scheme or implementation uses key data elements that are assumed to be immutable, but can be controlled or modified by the attacker, for example, if a web application relies on a cookie

```
Authenticated = 1.
```

Authentication logic error occurs when the authentication techniques do not follow the algorithms that define them exactly and so authentication can be jeopardized. For instance, a malformed or improper implementation of an algorithm can weaken the authorization technique.

An authentication bypass by primary weakness occurs when the authentication algorithm is sound, but the implemented mechanism can be bypassed as the result of a separate weakness that is primary to the authentication error.

A server that relies on client-side authentication or protection logic can be fooled by unexpected changes to the client protection code, resulting in maliciously modified server behaviour.

7.14.4 Avoiding the vulnerability or mitigating its effects

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

- Funnel all access through a single choke point to simplify how users can access a resource;
- For every access, perform a check to determine if the user has permissions to access the resource;
- Avoid making decisions based on names of resources (for example, files) if those resources can have alternate names;
- Canonicalize the name to match that of the file system's representation of the name;

  NOTE This can sometimes be achieved with an available API (for example, in Win32 the `GetFullPathName` function).

- Ensure that messages can be parsed only once, e.g. by including a sequence number or time stamp in a checksum;
- Use different keys for the initiator and responder or of a different type of challenge for the initiator and responder.

7.15 Improper restriction of excessive authentication attempts [WPL]

7.15.1 Description of application vulnerability

The software does not implement sufficient measures to prevent multiple failed authentication attempts within in a short time frame, making it more susceptible to brute force attacks.

7.15.2 Related coding guidelines

CWE [7]: 307. Improper Restriction of Excessive Authentication Attempts

7.15.3 Mechanism of failure

In a recent incident, an attacker targeted a member of a popular social networking sites support team and was able to successfully guess the member's password using a brute force attack by guessing a large number of common words. Once the attacker gained access as the member of the support staff, he used the administrator panel to gain access to a number of accounts that belonged to celebrities and politicians. Ultimately, fake messages were sent that appeared to come from the compromised accounts.

7.15.4 Avoiding the vulnerability or mitigating its effects

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

- Disconnect the user after a small number of failed attempts;

- Implement a timeout on authentication;
- Lock out a targeted account;
- Require a computational task on the user's part;
- Use a vetted library or framework that does not allow this vulnerability to occur or provides constructs that make this weakness easier to avoid;
- Consider using reputable libraries with authentication capabilities.

7.16 Hard-coded credentials [XYP]

7.16.1 Description of application vulnerability

Hard coded credentials (such as password, username/password or id number) will compromise system security in a way that cannot be easily remedied. It is never a good idea to hardcode any credential since not only does hard coding the credential allow all of the project's developers to view it, it also makes fixing the unintended disclosure of any credential extremely difficult. Once the code is in production, the credential cannot be changed without patching the software. If the account protected by the credential is compromised, the owners of the system will be forced to choose between security and availability.

7.16.2 Related coding guidelines

CWE[7]:

259. Hard-Coded Password

798. Use of Hard-coded Credentials

7.16.3 Mechanism of failure

The use of a hard-coded credentials has many negative implications – the most significant of these being a failure of authentication measures under certain circumstances. On many systems, a default administration account exists which is set to a simple default password that is hard-coded into the program or device. This hard-coded password is the same for each device or system of this type and often is not changed or disabled by end users. If a malicious user comes across a device of this kind, it is a simple matter of looking up the default credential (which is likely freely available and public on the Internet or the malicious user can view firmware as text to find text strings that resemble credentials) and logging in with complete access. In systems that authenticate with a back-end service, hard-coded credentials within closed source or drop-in solution systems require that the back-end service use a credential that can be easily discovered. Client-side systems with hard-coded credentials present even more of a threat since the extraction of a credential from a binary is exceedingly simple. If hard-coded credentials are used, it is almost certain that unauthorized users will gain access through the account in question.

7.16.4 Avoiding the vulnerability or mitigating its effects

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

- Use a *first login* mode that requires the user to enter a unique strong password or other credential rather than hard code a default credential (such as username and password) for first time logins;
- For front-end to back-end connections, use one or more of the following solutions:

1. Use generated credentials that are changed automatically and must be entered at given time intervals by a system administrator with the caveat that these credentials will be held in memory and only be valid for the time intervals.

2. Use credentials that are limited at the back end to only performing actions for the front end, as opposed to having full access;

3. Tag messages with a checksum that includes time sensitive values so as to prevent replay style attacks.

7.17 Insufficiently protected credentials [XYM]

7.17.1 Description of application vulnerability

This weakness occurs when the application transmits, or stores authentication credentials and uses an insecure method that is susceptible to unauthorized interception and/or retrieval.

7.17.2 Related coding guidelines

CWE [7]:

256. Plaintext Storage of a Password

257. Storing Passwords in a Recoverable Format

7.17.3 Mechanism of failure

Storing a credential, such as a password, in plaintext may result in a system compromise. Credential management issues occur when a credential is stored in plaintext in an application's properties or configuration file. A programmer can attempt to remedy the credential management problem by obscuring the credential with an encoding function, such as Base64 encoding, but this effort does not adequately protect the credential. Storing a plaintext credential in a configuration file allows anyone who can read the file access to the credential-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good credential management guidelines require that a credential never be stored in plaintext.

The storage of credentials in a recoverable format makes them subject to credential reuse attacks by malicious users. If a system administrator can recover the credential directly or use a brute force search on the information available to him, he can use the credential on other accounts.

The use of recoverable credentials significantly increases the chance that credentials will be used maliciously. Recoverable encrypted credentials provide no significant benefit over plain-text credentials since they are subject not only to reuse by malicious attackers but also by malicious insiders.

7.17.4 Avoiding the vulnerability or mitigating its effects

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

- Avoid storing credentials in easily accessible locations;
- Never store a credential in plain text;
- Ensure that strong, non-reversible encryption is used to protect stored credentials;
- Store cryptographic hashes of credentials as an alternative to storing in plaintext.

7.18 Missing or inconsistent access control [XZN]

7.18.1 Description of application vulnerability

The software does not perform access control checks in a consistent manner across all potential execution paths.

7.18.2 Related coding guidelines

CWE [7]:

285. Missing or Inconsistent Access Control

352. Cross-Site Request Forgery (CSRF**)**

807. Reliance on Untrusted Inputs in a Security Decision

862. Missing Authorization

CERT C guidelines [31]: FIO06-C

7.18.3 Mechanism of failure

For web applications, attackers can issue a request directly to a page (URL) that they may not be authorized to access. If the access control policy is not consistently enforced on every page restricted to authorized users, then an attacker could gain access to and possibly corrupt these resources.

7.18.4 Avoiding the vulnerability or mitigating its effects

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

- For web applications, ensure that the access control mechanism is enforced correctly at the server side on every page, so that users cannot access any information simply by requesting direct access to that page, if they do not have authorization;

- Ensure that all pages containing sensitive information are not cached, and that all such pages restrict access to requests that are accompanied by an active and authenticated session token associated with a user who has the required permissions to access that page.

7.19 Incorrect authorization [BJE]

7.19.1 Description of application vulnerability

The software performs a flawed authorization check when an actor attempts to access a resource or perform an action. This allows attackers to bypass intended access restrictions.

7.19.2 Related coding guidelines

CWE [7]:

863. Incorrect Authorization

7.19.3 Mechanism of failure

Authorization is the process of determining whether that user can access a given resource, based on the user's privileges and any permissions or other access-control specifications that apply to the resource.

When access control checks are incorrectly applied, users are able to access data or perform actions that they are not authorized to perform. This can lead to a wide range of problems, including information exposures, denial of service, and arbitrary code execution.

7.19.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects by ensuring that access control checks related to their business needs are performed.

NOTE These checks may be different and more detailed than those applied to more generic resources such as files, connections, processes, memory, and database records. For example, a database may restrict access for medical records to a specific database user, but each record might only be intended to be accessible to the patient and the patient's doctor.

7.20 Adherence to least privilege [XYN]

7.20.1 Description of application vulnerability

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

7.20.2 Related coding guidelines

CWE [7]: 250. Design Principle Violation: Failure to Use Least Privilege

CERT C guidelines [31]: POS02-C

7.20.3 Mechanism of failure

This vulnerability type refers to cases in which an application grants greater access rights than necessary. Depending on the level of access granted, this may allow a user to access confidential information. For example, programs that run with root privileges have caused innumerable UNIX security disasters.

It is imperative that developers carefully review privileged programs for all kinds of security problems, but it is equally important that privileged programs drop back to an unprivileged state as quickly as possible to limit the amount of damage that an overlooked vulnerability might be able to cause.

Privilege management functions can behave in some less-than-obvious ways, and they have different quirks on different platforms. These inconsistencies are particularly pronounced if a transition is in progress from one non-root user to another.

Signal handlers and spawned processes run at the privilege of the owning process, so if a process is running as root when a signal fires or a sub-process is executed, the signal handler or sub-process will operate with root privileges. An attacker may be able to leverage these elevated privileges to do further damage.

To grant the minimum access level necessary, first identify the different permissions that an application or user of that application will need to perform their actions, such as file read and write permissions, network socket permissions, and so forth. Then explicitly allow those actions while denying all else.

7.20.4 Avoiding the vulnerability or mitigating its effects

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

— Carefully manage the setting, management and handling of privileges;

— Explicitly manage trust zones in the software;

— Follow the principle of least privilege when assigning access rights to entities in a software system.

7.21 Privilege sandbox issues [XYO]

7.21.1 Description of application vulnerability

A variety of vulnerabilities occur with improper handling, assignment, or management of privileges. These are especially present in sandbox environments, although it could be argued that any privilege problem occurs within the context of some sort of sandbox.

7.21.2 Related coding guidelines

CWE [7]:

266. Incorrect Privilege Assignment

267. Privilege Defined With Unsafe Actions

268. Privilege Chaining

269. Privilege Management Error

270. Privilege Context Switching Error

272. Least Privilege Violation

273. Failure to Check Whether Privileges were Dropped Successfully

274. Failure to Handle Insufficient Privileges

276. Insecure Default Permissions

732. Incorrect Permission Assignment for Critical Resource

CERT C guidelines [31]: POS36-C

7.21.3 Mechanism of failure

The failure to drop system privileges when it is reasonable to do so is not an application vulnerability by itself. It does, however, serve to significantly increase the severity of other vulnerabilities. The principle of least privilege establishes that access is allowed only when it is absolutely necessary to the function of a given system, and only for the minimal necessary amount of time. Any further allowance of privilege widens the window of time during which a successful exploitation of the system will provide an attacker with that same privilege.

Many situations could lead to a mechanism of failure:

- A product could incorrectly assign a privilege to a particular entity;
- A particular privilege, role, capability, or right could be used to perform unsafe actions that were not intended, even when it is assigned to the correct entity;

  NOTE There are two separate sub-categories here: privilege incorrectly allows entities to perform certain actions; the object is incorrectly accessible to entities with a given privilege.

- Two distinct privileges, roles, capabilities, or rights could be combined in a way that allows an entity to perform unsafe actions that would not be allowed without that combination;

- The software may not properly manage privileges while it is switching between different contexts that cross privilege boundaries;
- A product may not properly track, modify, record, or reset privileges;
- In some contexts, a system executing with elevated permissions will hand off a process/file or other object to another process/user. If the privileges of an entity are not reduced, then elevated privileges are spread throughout a system and possibly to an attacker;
- The software may not properly handle the situation in which it has insufficient privileges to perform an operation;
- A program, upon installation, may set insecure permissions for an object.

7.21.4 Avoiding the vulnerability or mitigating its effects

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

- Follow the principle of least privilege when assigning access rights to entities in a software system, including carefully managing the setting, management and handling of privileges;
- Upon changing security privileges, verify that the change was successful;
- Follow the principle of separation of privilege. Require multiple conditions to be met before permitting access to a system resource;
- Explicitly manage trust zones in the software, including if at all possible, limiting the allowance of system privilege to small, simple sections of code that may be called atomically;
- Ensure that the operating system drops the elevated privilege and returns to the privilege level of the invoking user as soon as possible after calling a privileged function such as `chroot()`.

7.22 Missing required cryptographic step [XZS]

7.22.1 Description of application vulnerability

If cryptographic implementations do not follow the algorithms that define them exactly, the encryption will be faulty.

7.22.2 Related coding guidelines

CWE [7]:

325. Missing Required Cryptographic Step

327. Use of a Broken or Risky Cryptographic Algorithm

7.22.3 Mechanism of failure

Not following the algorithms that define cryptographic implementations exactly can lead to weak encryption. This could be the result of many factors such as a programmer missing a required cryptographic step or using weak randomization algorithms.

7.22.4 Avoiding the vulnerability or mitigating its effects

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

- Use system functions and libraries rather than writing the function;
- If a self-written algorithm is mandatory, implement cryptographic algorithms precisely.

7.23 Improperly verified signature [XZR]

7.23.1 Description of application vulnerability

The software does not verify, or improperly verifies, the cryptographic signature for data. By not adequately performing the verification step, the data being received is not trustworthy and can be corrupted or made intentionally incorrect by an adversary.

7.23.2 Related coding guidelines

CWE [7]: 347. Improperly Verified Signature

7.23.3 Mechanism of failure

Data is signed using techniques that assure the integrity of the data. There are two ways that the integrity can be intentionally compromised. The exchange of the cryptologic keys may have been compromised so that an attacker could provide encrypted data that has been altered. Alternatively, the cryptologic verification could be flawed so that the encryption of the data is flawed which again allows an attacker to alter the data.

7.23.4 Avoiding the vulnerability or mitigating its effects

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

- Use data signatures to the extent possible to help ensure trust in data;
- Use built-in verifications for data.

7.24 Use of a one-way hash without a salt [MVX]

7.24.1 Description of application vulnerability

The software uses a one-way cryptographic hash against an input that is expected to be unreversible, such as a credential, but the software does not also use a *salt* as part of the input.

7.24.2 Related coding guidelines

CWE [7]:

325. Missing Required Cryptographic Step

327. Use of a Broken or Risky Cryptographic Algorithm

759. Use of a One-Way Hash without a Salt

7.24.3 Mechanism of failure

This makes it easier for attackers to pre-compute the hash value using dictionary attack techniques such as rainbow tables.

7.24.4 Avoiding the vulnerability or mitigating its effects

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

- For a salt:

  - Generate a random salt each time a new credential is processed;
  - Add the salt to the plaintext credential before hashing it;
  - When the hash is stored, also store the salt;
  - Do not use the same salt for every credential processed;
- Use one-way hashing techniques that allow the configuration of a large number of rounds, such as bcrypt;

  NOTE This may increase the expense when processing incoming authentication requests, but if the hashed credentials are ever stolen, it significantly increases the effort for conducting a brute force attack, including rainbow tables. With the ability to configure the number of rounds, one can increase the number of rounds whenever CPU speeds or attack techniques become more efficient.

- Use industry-approved techniques correctly;
- Never skip resource-intensive steps (see CWE-325[7]) as these steps are often essential for preventing common attacks.

7.25 Inadequately secure communication of shared resources [CGY]

7.25.1 Description of application vulnerability

A resource that is directly visible from more than one process (at the same approximate time) and is not protected by access locks can be hijacked or used to corrupt, control, or change the behaviour of other processes in the system. Many vulnerabilities that are associated with concurrent access to files, shared memory or shared network resources fall under this vulnerability, including resources accessed via stateless protocols such as HTTP and remote file protocols.

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

7.25.2 Related coding guidelines

CWE [7]:

15. External Control of System or Configuration Setting

311. Missing Encryption of Sensitive Data

642. External Control of Critical State Data

367: Time of check, time of use

See also Burns and Wellings [5].

7.25.3 Mechanism of failure

Any time that a shared resource is open to general inspection, the resource can be monitored by a foreign process to determine usage patterns, timing patterns, and access patterns to determine ways that a planned attack can succeed. Such monitoring could be, but is not limited to:

- Reading resource values to obtain information of value to the applications;
- Monitoring access time and access thread to determine when a resource can be accessed undetected by other threads (for example, Time-of-Check-Time-Of-Use attacks rely upon a determinable amount of time between the check on a resource and the use of the resource when the resource could be modified to bypass the check);
- Monitoring a resource and modification patterns to help determine the protocols in use;
- Monitoring access times and patterns to determine quiet times in the access to a resource that could be used to find successful attack vectors.

This monitoring can then be used to construct a successful attack, usually in a later attack.

Such monitoring is usually possible by a process executing with system privilege, but even small slips in access controls and permissions let such resources be seen from other (non system level) processes. Even the existence of the resource, its size, or its access dates/times and history (such as “last accessed time”) can give valuable information to an observer.

Any time that a resource is open to general update, the attacker can plan an attack by performing experiments to:

- Discover how changes affect patterns of usage, timing, and access;
- Discover how application threads detect and respond to forged values.

Any time that a shared resource is open to shared update by a thread, the resource can be changed in ways to further an attack once it is initiated. For example, in a well-known attack, a process monitors a certain change to a known file and then immediately replaces a virus free file with an infected file to bypass virus checking software.

With careful planning, similar scenarios can result in the foreign process determining a weakness of the attacked process leading to an exploit consisting of anything up to and including arbitrary code execution.

7.25.4 Avoiding the vulnerability or mitigating its effect

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

- Place all shared resources in memory regions accessible to only one process at a time;
- Protect resources that must be visible with encryption or with checksums to detect unauthorized modifications;
- Obtain an unforgeable access path such as the file handle obtained on first access;
- Protect access to shared resources using an unforgeable access path, permissions, access control, or obfuscation;
- Have and enforce clear rules with respect to permissions to change shared resources;
- Detect attempts to alter shared resources and take immediate action.

7.26 Memory locking [XZX]

7.26.1 Description of application vulnerability

Sensitive data stored in memory that was not locked or that has been improperly locked may be written to swap files on disk by the virtual memory manager.

7.26.2 Related coding guidelines

CWE [7]: 591. Sensitive Data Storage in Improperly Locked Memory

CERT C guidelines [31]: MEM06-C

7.26.3 Mechanism of failure

Sensitive data that is not kept cryptographically secure may become visible to an attacker by any of several mechanisms. Some operating systems may write memory to swap or page files that may be visible to an attacker. Some operating systems may provide mechanisms to examine the physical memory of the system or the virtual memory of another application.

Application debuggers may be able to stop the target application and examine or alter memory.

Systems that provide a *hibernate* facility (such as laptops) will write all of physical memory to a file that may be visible to an attacker on resume.

In almost all cases, these attacks require elevated or appropriate privilege.

7.26.4 Avoiding the vulnerability or mitigating its effects

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

- Remove debugging tools from production systems;
- Log and audit all privileged operations;
- Identify data that needs to be protected and use appropriate cryptographic and other data obfuscation techniques to avoid keeping plaintext versions of this data in memory or on disk;

  NOTE For example several implementations of the POSIX `mlock()` and the Microsoft Windows `VirtualLock()` functions will prevent the named memory region from being written to a swap or page file, but such usage, however is not portable.

- If the operating system allows, clear the swap file on shutdown.

7.27 Sensitive information not cleared before use [XZK]

7.27.1 Description of application vulnerability

The software does not fully clear previously used information in a data structure, file, or other resource, before making that resource available to another party that did not have access to the original information.

7.27.2 Related coding guidelines

CWE [7]: 226. Sensitive Information Uncleared Before Release

CERT C guidelines [31]: MEM03-C

7.27.3 Mechanism of failure

This typically involves memory in which the new data occupies less memory than the old data, which leaves portions of the old data still available (*memory disclosure*). However, equivalent errors can occur in other situations where the length of data is variable but the associated data structure is not. This can overlap with cryptographic errors and cross-boundary cleansing information leaks.

Dynamic memory managers are not required to clear freed memory and generally do not because of the additional runtime overhead. Furthermore, dynamic memory managers are free to reallocate this same memory. As a result, it is possible to accidentally leak sensitive information if it is not cleared before calling a function that frees dynamic memory. Programmers cannot rely on memory being cleared during allocation.

7.27.4 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects by using library functions and or programming language features (such as destructors or finalization procedures) that provide automatic clearing of freed buffers or the functionality to clear buffers.

7.28 Time consumption measurement [CCM]

7.28.1 Description of application vulnerability

All applications consume resources as they execute, in particular, Time. Each thread, event, interrupt and OS service consume CPU time that may be separately measurable by the system.

A common paradigm in managing applications is to monitor such resource usage by thread and take action to cease the calculation for the thread that is consuming excessive time, using techniques such as abort, raise exception, lower priority or suspending the thread. If the calculation cannot be completed in time or within the resource constraints imposed upon it, then the application may fail.

The consumption of CPU resources (execution time) can be affected by changes in the CPU itself: for example, CPU’s may slow down to manage heat, resulting in more execution time to achieve a result. Similarly, cache misses due to the way a program is organized and executed, due to multiprocessor effects, can increase the execution time needed to complete a calculation.

The measurement of resource timing and consumption can be used to break sensitive algorithms. For example, some devices draw power from systems that they piggyback onto (such as chip cards and proximity-based passive systems).

7.28.2 Related coding guidelines

See Burns and Wellings [5]

7.28.3 Mechanism of failure

Many applications measure resource consumption to detect failures of portions of portions of the algorithm and to make decisions about alternative actions. For example, excessive consumption of CPU may indicate that a thread is executing erroneously; and that other needed threads may not be able to execute due to excessive resource consumption.

Other factors, such a CPU speed changes and cache misses can cause a thread to consume significantly more CPU resources than expected to perform the same calculations.

A thread consuming more CPU resources than planned can result in missed deadlines for itself, or can take CPU resources needed by other threads, causing incorrect processing or missed deadlines for other threads. Missed deadlines are catastrophic for hard real-time systems and cover the range of causing wrong results through to complete failure of the application.

For systems that live in the low powered consumption domain but require modern encryption, the device providing the power can use knowledge about power consumed to narrow the possible hashing algorithms or encryption algorithms used which may let the attacker defeat encrypt-ion or digital signing security systems.

7.28.4 Avoiding the vulnerability or mitigating its effect

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

- Monitor time consumption by execution unit (process, task, thread, etc.) and react to overconsumption in ways that make sense for the system being developed;
- Execute with cache disabled to provide consistent timing and behaviour to avoid situations where cache misses provide a significant potential hindrance;
- Perform static response time analysis to guard against overconsumption;
- For ultra-low powered devices (and for encryption-based systems in general), base the protection on more than encryption, such as obfuscation and indirection inside of the encryption protection.

7.29 Discrepancy information leak [XZL]

7.29.1 Description of application vulnerability

A discrepancy information leak is an information leak in which the product behaves differently, or sends different responses, in a way that reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

7.29.2 Related coding guidelines

CWE [7]:

203. Discrepancy Information Leaks

204. Response Discrepancy Information Leak

206. Internal Behavioral Inconsistency Information Leak

207. External Behavioral Inconsistency Information Leak

208. Timing Discrepancy Information Leak

7.29.3 Mechanism of failure

A response discrepancy information leak occurs when the product sends different messages in direct response to an attacker’s request, in a way that allows the attacker to learn about the inner state of the product. The leaks can be inadvertent (bug) or intentional (design).

A behavioural discrepancy information leak occurs when the product’s actions indicate important differences based on (1) the internal state of the product or (2) differences from other products in the same class. Attacks such as OS fingerprinting rely heavily on both behavioural and response discrepancies.

An internal behavioural inconsistency information leak is the situation where two separate operations in a product cause the product to behave differently in a way that is observable to an attacker and reveals security-relevant information about the internal state of the product, such as whether a particular operation was successful or not.

An external behavioural inconsistency information leak is the situation where the software behaves differently than other products like it, in a way that is observable to an attacker and reveals security-relevant information about which product is being used, or its operating state.

A timing discrepancy information leak occurs when two separate operations in a product require different amounts of time to complete, in a way that is observable to an attacker and reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

7.29.4 Avoiding the vulnerability or mitigating its effects

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

- Compartmentalize the system to have *safe* areas where trust boundaries can be unambiguously drawn;
- Prevent sensitive data from migrating outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area.

7.30 Unspecified functionality [BVQ]

7.30.1 Description of application vulnerability

While unspecified functionality may be no more than an amusing 'Easter Egg', like the flight simulator in a spreadsheet, it raises serious questions about the level of control of the development process.

In a security-critical environment particularly, the developer of a security-critical software application could include a 'trap-door' to allow illegitimate access to the system on which it is eventually executed, irrespective of whether the application has obvious security requirements.

7.30.2 Related coding guidelines

JSF AV [24] Rule: 127

MISRA C [29]: 1.2, 2.1, 3.1, and 4.4

7.30.3 Mechanism of failure

Unspecified functionality is not a software vulnerability per se, but more a development issue. In some cases, unspecified functionality may be added by a developer without the knowledge of the development organization. In other cases, typically Easter Eggs, the functionality is unspecified as far as the user is concerned (nobody buys a spreadsheet expecting to find it includes a flight simulator), but is specified by the development organization. In effect they only reveal a subset of the program's behaviour to the users.

In the first case, one would expect a well-managed development environment to discover the additional functionality during validation and verification. In the second case, the user is relying on the supplier not to release harmful code.

In effect, a program's requirements are 'the program behaves in the following manner and does nothing else'. The 'and do nothing else' statement is often not explicitly stated, and can be difficult to demonstrate. See also 6.65 Modifying constants [UJO]

7.30.4 Avoiding the vulnerability or mitigating its effects

End users can avoid the vulnerability or mitigate its ill effects in the following ways:

- Ensure that programs and development tools that are to be used in critical applications come from a developer or organization that uses a recognized and audited development process for the development of those programs and tools;

- Ensure that the development process generates documentation showing traceability from source code to requirements, in effect answering ‘why is this unit of code in this program?’.

  Note Where unspecified functionality is there for a legitimate reason (such as diagnostics required for developer maintenance or enhancement), the documentation is expected to record this. It is not unreasonable for customers of bespoke critical code to ask to see such traceability as part of their acceptance of the application.

7.31 Fault tolerance and failure strategies [REU]

7.31.1 Description of application vulnerability

In spite of the best intentions, system components may fail, either from internally poorly written software or external forces such as power outages/variations, radiation or inadmissible user input. Safety-critical software systems and application-critical software systems are often designed with fault tolerance to detect and deal with such failures. Faults are the points in execution where a failure manifests by processing going wrong. If unnoticed or unhandled, they turn into failures at the boundaries of enclosing control units or components. Failures of services are faults to their clients and, if not handled, lead to a failure of the client and consequently to faults and failures in its clients, possibly until the entire system fails. The origins of faults are often vulnerabilities discussed in the other clauses of this document.

A lack of dealing with a failure is obviously an exploitable vulnerability, but even if the failure is dealt with, the resulting fault tolerance is itself a potential source of vulnerabilities, particularly when inappropriate or incomplete strategies are implemented.

Fault-handling code is difficult to design and program, since it needs to execute in an already damaged environment. Handler code is also difficult to test, since it is executed only when primary failures have occurred. These failures, e.g., radiation damage, may be impossible to recreate with sufficient coverage in a testing environment.

Moreover, it is not easy to determine the right kind of fault tolerance for a given fault. For security, termination of the malfunctioning system may be the best action; for safety, termination may be more catastrophic than other fault tolerance mechanisms. Recovery in a local context may be impossible, e.g. querying a faulty location sensor, while a (transitively) calling routine may have sufficient content for a recovery action, e.g. obtaining location information from another source.

Arising vulnerabilities are, for example:

- The fault is not recognized and the system malfunctions or terminates as a consequence;
- The fault is recognized but the damage already done is incompletely repaired, with the same consequences as in the first bullet;
- A value fault is recognized too late, allowing the incorrect value to be used in the computations of other, thus corrupted, values (which, if not repaired, can cause vulnerabilities such as buffer overflows);
- The fault tolerance processing takes too long to meet timing demands;
- Recovery is prevented by the cause of a permanent fault, e.g. a programming error, leading to an infinite series of recovery attempts;
- The fault tolerance mechanism causes itself new faults

Triggering known fault detection mechanisms can be used to initiate or aggravate Denial-of-Service attacks. Knowledge of a lack of fault detection, particularly of value faults, can be used to initiate system intrusions through mechanisms explained elsewhere in this document. Whatever the failure or termination process, it is expected that the termination of an application does not result in damage to system elements that rely upon it. Thus, it needs to perform *last wishes* to minimize the effects of the failure on enclosing components (e.g. release software locks) and the real world (e. g. close valves).

7.31.2 Related coding guidelines

JSF AV [24] Rule: 24

MISRA C [29]: 4.1

MISRA C++ [30]: 0-3-2, 15-5-2, 15-5-3, and 18-0-3

CERT C guidelines [31]: ERR04-C, ERR06-C and ENV32-C

Ada Quality and Style Guide [1]: 5.8 and 7.5

7.31.3 Mechanism of failure

Reasons for failures are plentiful and varied, stemming from both hardware and software. The mechanisms of failure from fault tolerance or the lack thereof can be described only in very general terms.

- Lack of fault-tolerance code leaves the system in an ill-defined state in case of a fault, resulting in crashes (*fail stop*), looping or waiting forever (*fail silent*), or operating with incorrect data causing incorrect results.
- Inconsistent approaches to detecting and handling a fault or a lack of overall design for the fault tolerance code can potentially be a vulnerability, as faults might escape the necessary attention.
- Fault tolerance code, in particular fault checking code, may interfere with the timeliness of the components to meet their deadlines.
- An inappropriate fault tolerance mechanism or strategy may lead to failures during fault detection and to other secondary failures. For example, trying to recover from a systematic software error by a retry mechanism leads to an infinite loop as the same error will reoccur. Yet, retry strategies may be best for a transient fault situation.
- Considerable latency and processor use can arise from finalization and garbage collection caused by the termination of a service. Thus, termination must be designed carefully to avoid causing timing failures of other services. The termination of services can be maliciously used to prevent on-time performance of other active services.

7.31.4 Avoiding the vulnerability or mitigating its effects

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

- Decide on a consistent strategy for fault handling;
- Use a multi-tiered approach of fault prevention, fault detection and fault reaction;
- Unambiguously describe the failure modes of each possibly failing service;
- Check early for any faults, particularly value faults. Perform numerous checks on values (value range, plausibility within history, reversal checks, checksums, structural checks, etc.) to establish the validity of computed results or input received;
- Validate incoming data and computed results at strategic points to discover value failures;

- Check pre-conditions and postconditions not validated otherwise (See *6.42 Violations of the Liskov substitution principle or the contract model [BLP]*);

- Detect timing failures by watch-dog timers or similar mechanisms;

- Use environment-provided or language-provided means to stop services that substantially exceed deadlines;

- Always prepare for the possibility that a service does not return with a requested result in due time;

- Keep fault handling simple. If in doubt, decide for a lesser level of fault tolerance;

- In the case of continued execution, make sure that any corrupted variables of the program state have been corrected to an actual and correct or at least safe value;

- In the case of a *retry* strategy, ensure that progress is made by limiting the number of retries;

- Use system-defined components that assist in uniformity of fault handling when available;

- Prior to abnormal termination of a component, perform *last wishes* to minimize the effects of the failure on enclosing components (e.g. release software locks held locally) and the real world (e.g. close valves opened by the component);

- Specify a fault-handling policy whereby a service, in the absence of full fault tolerance or graceful degradation, will halt safely and securely respectively.

7.32 Distinguished values in data types [KLK]

7.32.1 Description of application vulnerability

Sometimes, in a type representation, certain values are distinguished as not being members of the type, but rather as providing auxiliary information. Examples include special characters used as string terminators, distinguished values used to indicate out of type entries in *SQL* (Structured query language) database fields, and sentinels used to indicate the bounds of queues or other data structures. When the usage pattern of code containing distinguished values is changed, it may happen that the distinguished value happens to coincide with a legitimate in-type value. In such a case, the value is no longer distinguishable from an in-type value and the software will no longer produce the intended results.

7.32.2 Related coding guidelines

CWE [7]:

20. Improper input validation

137. Representation errors

JSF AV [24] Rule: 151

7.32.3 Mechanism of failure

A *distinguished value* or a *magic number* in the representation of a data type might be used to represent out-of-type information. Some examples include the following:

- The use of a special code, such as “`00`”, to indicate the termination of a coded character string;

- The use of a special value, such as “999...9”, as the indication that the actual value is either not known or is invalid.

If the use of the software is later generalized, the once-special value can become indistinguishable from valid data. The problem may occur simply if the pattern of usage of the software is changed from that anticipated by the software’s designers. It may also occur if the software is reused in other circumstances.

An example of a change in the pattern of usage is that an organization “logs” visitors to its buildings by recording their names and national identity numbers or social security numbers in a database. Of course, some visitors legitimately do not have or do not know their social security number, so the receptionists enter numbers to “make the computer happy.” Receptionists at one building have adopted the convention of using the code “555-55-5555” to designate children of employees. Receptionists at another building have used the same code to designate foreign nationals. When the databases are merged, the children are reclassified as foreign nationals or vice-versa depending on which set of receptionists are using the newly merged database.

An example of an unanticipated change due to reuse is as follows. Suppose a software component analyzes radar data, recording data every degree of azimuth from 0 to 359. Packets of data are sent to other components for processing, updating displays, recording, and so on. Since all degree values are non-negative, a distinguished value of -1 is used as a signal to stop processing, compute summary data, close files, and so on. Many of the components are to be reused in a new system with a new radar analysis component. However, the new component represents direction by numbers in the range -180 degrees to 179 degrees. When an azimuth value of -1 is provided, the downstream components will interpret that as the indication to stop processing. If the magic value is changed to, say, -999, the software is still at risk of failing when future enhancements (say, counting accumulated degrees on complete revolutions) bring -999 into the range of valid data.

Avoid distinguished values; instead design the software to use distinct variables to encode the desired out-of-type information. For example, the length of a character string might be encoded in a dope vector and validity of data entries might be encoded in distinct Boolean values.

This vulnerability extends to numbers placed in the code, such as 7, hex F00F. such numbers are almost universally used in multiple places. The first issue is that there is rarely a full explanation given for the value in all places where it is defined, and under maintenance maintainers are left guessing at its meaning. A second issue is that, under maintenance (or before), the value changes, but not all occurrences get updated, causing erroneous algorithms. A third issue is that such “magic values” are almost always placed in the data section of the file that contains the executable program, and simple searches with data dumping tools reveals such values (say for example a password) to a possible attacker to use in attempting to break the application

7.32.4 Avoiding the vulnerability or mitigating its effects

Programmers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use auxiliary variables (perhaps enclosed in variant records) to encode out-of-type information;
- Use enumeration types to convey category information. Do not rely upon large ranges of integers, with distinguished values having special meanings;
- Use named constants to make it safer and easier to change distinguished values.

7.33 Clock issues [CCI]

7.33.1 Description of application vulnerability

All processors and operating systems maintain multiple representations of time internal to the system. In a typical system, there are the following notions of time, and potentially identifiable clocks:

- CPU time;
- Process/task/thread execution time;
- Calendar clock time, local and/or GMT;
- Elapsed time – i.e. time since system inception in seconds, or in fixed portions thereof;
- Network time.

These times have different representations, different scaling, and different semantics. For example, a time-of-day clock must account for leap years, leap seconds and standard/daylight saving times but a CPU or processor clock is a monotonic clock that must maintain time used by a task, thread, or process in a granularity appropriate to CPU speed – possibly sub-nanosecond. A real-time clock is a monotonic clock that manages and represents time to a granularity and representation needed to correctly manage the algorithms of the system. Both are usually associated with inputs from external devices or systems and outputs to initiate events in connected systems.

Some of these clocks are manifested in programming languages, especially for safety-critical software systems and application-critical software applications. For example, most languages have time of day clock lookup, while real time languages often include monotonic clocks for various purposes. Alternatively, some languages provide library services to access and manipulate time bases, and to schedule activity based upon one of the clocks.

Using the wrong clock for certain activities can lead to erroneous results for the application or for the system that relies upon it. For example, setting a trigger at 10 pm to execute a script for 6 am by using the real-time clock with a delay of 8 hours, then when the calendar clock resets due to a change from winter time to summer time, the script will execute an hour late.

Converting from one time-base to another time-base can result in loss of precision, rounding errors, and conversion errors which can lead to substantial jitter in the application behaviour or complete failure of the application.

Roll-over of a clock can cause failure of applications that are expecting uniformly increasing time, which can lead to transient failure of the application and possibly the parent system.

7.33.2 Related coding guidelines

See Burns and Wellings [5], and Kopetz and Hermann [25] for discussions of clock issues, realtime or non-realtime.

7.33.3 Mechanism of failure

The following time issues can lead to failures:

- Differing time bases within a single computational system;
- Time conversions between different time formats within a computational system;
- Drift between the notion of time for computational elements in a single system;
- Roll-over of one or multiple time-bases within the life of an executing system.

Almost all computational systems have different time bases that proceed at slightly different rates. This may be due to having different processors, each with their own clock, from having processor clocks as well as clocks provided by dedicated hardware, or by having different numbers of bits in time representation and updating them individually. The situation is most obvious when processors are physically distinct and connected by a network, but similar issues exist within a conceptual single CPU. Even within a conceptual single computer, issues such as cache misses or time lost to interrupt processing can change the rate at which clocks advance in time and can causes differences between clocks. Situations with different representations of time and converting between them are covered under time conversion below.

Different clocks will always proceed at different rates. This is covered under clock drift below.

When multiple time bases are supported, there are mechanisms to convert from one-time format to another to support calculations done. In those conversions, conversion errors, rounding errors or cumulative errors can develop, as follows:

— if the conversion is not done from the most precise time formats to less precise time formats;

— if conversions are done from one format to another and then back for comparison;

— if iterative calculations are done using less than the most precise time base possible.

Such conversions can lead to missed deadlines or wrong calculations that depended on accurate time representation and can result in catastrophic loss of the application or the parent system. A classic example of this is the common (wrong) paradigm to use the calendar clock to derive values to be programmed into the monotonic clock.

Clock drift happens when a system is spread over multiple processors, and the time measured by different code instances differs due to either distance and communication lags, or due to accumulated differences in how local clocks record time. When code is written for an application, the developer usually assumes that there is a common time base for all portions or instances of the application that are in communication with each other. Over (relatively) long durations of time, the differences will compound until a time comparison using multiple clocks or different time bases causes an algorithm failure. Such a failure may have catastrophic effects on the parent system, unless mechanisms are put in place to detect drift and to either correct some clocks, or to correct the algorithms that compare time.

Time roll-over happens because the efficient representation of time in all computational systems relies on fixed internal representation of time, which will eventually overflow the storage capacity of the representation if the computational system runs long enough. Examples of classic time roll-overs include the year 2000 issues when many time recording formats relied on a two-digit format for the year, resulting in year 2000 being rolled back to zero; and the roll-over of 32-bit seconds counting after 4 billion microseconds (or about 4 million seconds).

Classic time roll-over can also happen if the time base is external, such as the global positioning satellite time base. Code that relies upon the time-base constantly increasing will fail when a rollover occurs, leading to failure of the computational system and possible catastrophic loss of the parent system, unless the application is programmed to account for this rollover. Most systems create a real-time time base such that the system will never roll over within the expected operational time of the system. Modifications to the system, however, such as speeding up the clock that feeds the time base or dramatically increasing the expected operational lifetime of the system can make such errors happen, with potential catastrophic loss of the system and any systems that depend upon it.

The time of day clock is adjusted internally to jump or to be set backwards when going to or leaving summer time, inserting leap seconds, switching time zones or correcting time to synchronize the clock with a time base or another clock. Using the wrong clock, especially the time-of-day clock, to schedule events can result in jitter in the system, events being scheduled early, or the event being late. The mis-scheduling of events can have real world applications up to and including catastrophic loss of the parent system.

7.33.4 Avoiding the vulnerability or mitigating its effect

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

— Always convert time from the most precise and stable time base to less precise time bases;

— Avoid conversions from calendar clocks or network clocks to real time clocks;

— Use only clocks that have known synchronization properties;

— Avoid using the time of day clock to schedule events, unless the event is demonstrably connected with real world time of day, such as setting an alarm for 7 am;

— Avoid resetting or reprogramming the real-time clock or execution timers, unless the complete application is being reset;

— Allow some variability or error margin in the reading of time and the scheduling of time based on the read;

— Ensure that any code operates correctly in a time roll-over scenario;

NOTE Computations involving time values before and after roll-over may yield unexpected results.

— Ensure that any code operates correctly in any scenario involving time jumps (such as leap seconds, time corrections, time zones and daylight savings time).

7.34 Time drift and jitter [CDJ]

7.34.1 Description of application vulnerability

Many real-time systems are characterized by collections of jobs waiting for a start-time for a time-based iteration, or an event for sporadic activities. A common mistake in programming such systems is to base the start time of the next iteration upon

- either a non-monotonic or a non-realtime clock; or
- an offset from the start time or completion time of the last iteration.

In the first case, conversion errors and possible drift of the clock can cause the next iteration to be wrongly programmed. In the second case, higher priority work may have delayed the actual start or completion of the task in an individual iteration, resulting again in time drift.

With enough drift, an iterative task will begin missing its deadlines, and will either produce the wrong results or fail completely, resulting in arbitrary failures up to and including the catastrophic loss of the enclosing system.

Many systems have moved to a virtualization approach to fielding systems. Sometimes the virtual system is only an operating system change, such as running different operating systems on the same hardware. Sometimes the virtual system is hardware and software. Sometimes hardware is dedicated, such as 2 cores from an 8-core system, while in others the virtual system under consideration only executes when needed. The discussion of virtualization includes the common notions, such as hypervisors, but also include systems as diverse as satisfying ARINC 653 [3] which uses a time-based partition approach to schedule mixed criticality systems on a single CPU.

In any case, when a system is virtual, its connection with the real world (i.e. hardware and virtualizer) clocks is indirect. Clocks for the virtualized system are updated when the virtualized system resumes, and time may “jump” or may advance much faster than normal until the clocks are synchronized with the real world. Similarly, time may run slowly or erratically in an executing virtualized system. These behaviours can result in processes being mis-synchronized or missing deadlines if time jumps or progresses too quickly for the task to get its work completed.

If an attacker is aware that an application is virtualized, or that it is depending upon a non-real-time clock, and can determine what other applications share the same resource, they may be able to generate load for the other virtualized applications so that the one in question cannot retain enough resources to function correctly.

7.34.2 Related coding guidelines

See Burns and Wellings [5], and Kopetz and Hermann [25] for discussions of time representation issues and time keeping issues.

7.34.3 Mechanism of failure

Any change in the progression of time can result in a disconnect between the spacing of the delivery of time events to the application and can make jobs within the application run past their deadlines (as viewed by the timing events).

Deadline overrun is a serious flaw in the application, and usually results in failure of portions of the application up to catastrophic failure of the application and may result in loss of the parent system.

When a system is virtualized, an attacker can use influence over other applications to consume resources needed by the critical system that could trigger such systems.

Programming mistakes, such as failure to use monotonic clocks to schedule iterations, or incorrectly programming the next iteration calculations (such as setting the next wake time based on the start of the current wake time vs a fixed offset from the previous scheduled start time) result in drift or jitter which may result in missed real world inputs or loss of synchronization with external systems.

7.34.4 Avoiding the vulnerability or mitigating its effect

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

- Always set the next (absolute) start time for the iteration from the start time of the previous programmed iteration;
- Only use the real-time clock in scheduling tasks or events;
- Create management jobs that can monitor and detect application parts that exceed time bounds, such as execution time or elapsed time;
- Ensure that the behaviour of a virtualized application cannot be compromised by changes to the environment of the virtualized system.

Annex A
(informative)

Vulnerability taxonomy and list

A.1 General

This document is a catalogue that will continue to evolve. For that reason, a scheme that is distinct from subclause numbering has been adopted to identify the vulnerability descriptions. Each description has been assigned an arbitrarily generated, unique three-letter code. These codes are preferable to subclause numbers when referencing descriptions because they will not change as additional descriptions are added to future editions of this document. However, it is recognized that readers may need assistance in locating descriptions of interest.

This annex provides a taxonomical hierarchy of vulnerabilities, which users may find to be helpful in locating descriptions of interest. A.2 is a taxonomy of the programming language vulnerabilities described in clause 6 and A.3 is a taxonomy of the application vulnerabilities described in clause 7. A.4 lists the vulnerabilities in the alphabetical order of their three-letter codes and provides a cross-reference to the relevant subclause.

A.2 Taxonomy of programming language vulnerabilities

A.2.1. Types

A.2.1.1. Representation

A.2.1.1.1. [IHN] Type system

A.2.1.1.2. [STR] Bit representations

A.2.1.2. Floating-point

A.2.1.2.1. [PLF] Floating-point arithmetic

A.2.1.3. Enumerated types

A.2.1.3.1. [CCB] Enumerator issues

A.2.1.4. Integers

A.2.1.4.1. [FLC] Conversion errors

A.2.1.5. Characters and strings

A.2.1.5.1 [CJM] String termination

A.2.1.5.2. [SHL] Reliance on external format string

A.2.1.6. Arrays

A.2.1.6.1. [HCB] Buffer boundary violation (Buffer overflow)

A.2.1.6.2. [XYZ] Unchecked array indexing

A.2.1.6.3. [XYW] Unchecked array copying

A.2.1.7. Pointers

A.2.1.7.1. [HFC] Pointer type conversions

A.2.1.7.2. [RVG] Pointer arithmetic

A.2.1.7.3. [XYH] Null pointer dereference

A.2.1.7.4. [XYK] Dangling reference to heap

A.2.2. Type-conversions/limits

A.2.2.1. [FIF] Arithmetic wrap-around error

A.2.2.1 [PIK] Using shift operations for multiplication and division

A.2.3. Declarations and definitions

A.2.3.1. [NAI] Choice of clear names

A.2.3.2. [WXQ] Dead store

A.2.3.3. [YZS] Unused variable

A.2.3.4. [YOW] Identifier name reuse

A.2.3.5. [BJL] Namespace issues

A.2.3.6. [LAV] Initialization of variables

A.2.3.7. [UJO] Modifying constants

A.2.4. Operators/Expressions

A.2.4.1. [JCW] Operator precedence and associativity

A.2.4.2. [SAM] Side-effects and order of evaluation of operators

A.2.4.3. [KOA] Likely incorrect expression

A.2.4.4. [XYQ] Dead and deactivated code

A.2.5. Control flow

A.2.5.1. Conditional statements

A.2.5.1.1. [CLL] Switch statements and lack of static analysis

A.2.5.1.2. [EOJ] Non-demarcation of control flow

A.2.5.2. Loops

A.2.5.2.1. [TEX] Loop control variables

A.2.5.2.2. [XZH] Off-by-one error

A.2.5.3. Subroutines (functions, procedures, subprograms)

A.2.5.3.1. [EWD] Unstructured programming

A.2.5.3.2. [CSJ] Passing parameters and return values

A.2.5.3.3. [DCM] Dangling references to stack frames

A.2.5.3.4. [OTR] Subprogram signature mismatch

A.2.5.3.5. [GDL] Recursion

A.2.5.3.6. [OYB] Ignored error status and unhandled exceptions

A.2.6. Memory models

A.2.6.1. [AMV] Type-breaking reinterpretation of data

A.2.6.2. [YAN] Deep vs shallow copying

A.2.6.3. [XYL] Memory leaks and heap fragmentation

A.2.7. Object-oriented programming and contract model

A.2.7.1. [SYM] Templates and generics

A.2.7.2. [RIP] Inheritance

A.2.7.3. [BLP] Violations of the Liskov substitution principle or the contract model

A.2.7.4 [PPH] Redispatching

A.2.7.5 [BKK] Polymorphic variables

A.2.8. Libraries

A.2.8.1 [LRM] Extra intrinsics

A.2.8.2. [TRJ] Argument passing to library functions

A.2.8.3. [DJS] Inter-language calling

A.2.8.4. [NYY] Dynamically-linked code and self-modifying code

A.2.8.5. [NSQ] Library signature

A.2.8.6. [HJW] Unanticipated exceptions from library routines

A.2.9. Macros

A.2.9.1. [NMP] Pre-processor directives

A.2.10. Compile and runtime

A.2.10.1 [MXB] Suppression of language-defined run-time checking

A.2.10.2 [SKL] Provision of inherently unsafe operations

A.2.11. Language specification issues

A.2.11.1. [BRS] Obscure language features

A.2.11.2. [BQF] Unspecified behaviour

A.2.11.3. [EWF] Undefined behaviour

A.2.11.4. [FAB] Implementation-defined behaviour

A.2.11.5. [MEM] Deprecated language features

A.2.12. Concurrency

A.2.12.1 [CGA] Concurrency – Activation

A.2.12.2 [CGT] Concurrency – Directed termination

A.2.12.3 [CGS] Concurrency – Premature termination

A.2.12.4 [CGX] Concurrent data access

A.2.12.6 [CGM] Lock protocol errors

A.3 Taxonomy of application vulnerabilities

A.3.1. Design issues

A.3.1.1. [BVQ] Unspecified functionality

A.3.1.2. [REU] Fault tolerance and failure strategies

A.3.1.3. [KLK] Distinguished values in data types

A.3.2. Environment

A.3.2.1. [XYN] Adherence to least privilege

A.3.2.2. [XYO] Privilege sandbox issues

A.3.2.3. [XYS] Executing or loading untrusted code

A.3.3. Resource management

A.3.3.1. Memory management

A.3.3.1.1. [XZX] Memory locking

A.3.3.1.2. [XZP] Resource exhaustion

A.3.3.2. Input

        A.3.3.2.1. [CBF] Unrestricted file upload

        A.3.3.2.2. [HTS] Resource names

        A.3.3.2.3. [RST] Injection

        A.3.3.2.4. [XYT] Cross-site scripting

        A.3.3.2.5. [XZQ] Unquoted search path or element

        A.3.3.2.7. [XZL] Discrepancy information leak

        A.3.3.2.8. [EFS] Use of unchecked data from an uncontrolled or tainted source

    A.3.3.3. Output

        A.3.3.3.1. [XZK] Sensitive information uncleared before use

    A.3.3.4. Files

        A.3.3.4.1. [EWR] Path traversal

    A.3.3.5 Execution issues

        A.3.3.5.1 [CCM] Time consumption measurement

        A.3.3.5.2 [CCI] Clock issues

        A.3.3.5.3 [CDJ] Time drift and jitter

A.3.4 Concurrency and parallelism

    A.3.4.1 [CGY] Inadequately secure communication of shared resources

A.3.5. Flaws in security functions

    A.3.5.1. [XZS] Missing required cryptographic step

    A.3.5.2. [MVX] Use of a one-way hash without a salt

A.3.6. Flaws in authentication

    A.3.6.1. [XZR] Improperly verified signature

    A.3.6.2. [XYM] Insufficiently protected credentials

    A.3.6.3. [XZN] Missing or inconsistent access control

    A.3.6.4. [XZO] Authentication logic error

    A.3.6.5. [XYP] Hard-coded credentials

    A.3.6.6. [DLB] Download of code without integrity check

A.3.6.7. [BJE] Incorrect authorization

A.3.6.8. [DHU] Inclusion of functionality from untrusted control sphere

A.3.6.9. [WPL] Improper restriction of excessive authentication attempts

A.3.6.10. [PYQ] URL redirection to untrusted site ('Open redirect')

A.4 Vulnerability list

Code	Vulnerability Name	Subclause
[AMV]	Type-breaking reinterpretation of data	6.37
[BJE]	Incorrect authorization	7.19
[BJL]	Namespace issues	6.21
[BKK]	Polymorphic variables	6.44
[BLP]	Violations of the Liskov substitution principle or the contract model	6.42
[BQF]	Unspecified behaviour	6.55
[BRS]	Obscure language features	6.54
[BVQ]	Unspecified functionality	7.30
[CBF]	Unrestricted file upload	7.2
[CCB]	Enumerator issues	6.5
[CCI]	Clock issues	7.33
[CCM]	Time consumption measurement	7.28
[CDJ]	Time drift and jitter	7.34
[CGA]	Concurrency – activation	6.59
[CGM]	Lock protocol errors	6.63
[CGS]	Concurrency - Premature termination	6.62
[CGT]	Concurrency - Directed termination	6.60
[CGX]	Concurrent data access	6.61
[CGY]	Inadequately secure communication of shared resources	7.25
[CJM]	String termination	6.7
[CLL]	Switch statements and lack of static analysis	6.27
[CSJ]	Passing parameters and return values	6.32
[DCM]	Dangling references to stack frames	6.33
[DHU]	Inclusion of functionality from untrusted control sphere	7.5
[DJS]	Inter-language calling	6.47
[DLB]	Download of code without integrity check	7.3
[EFS]	Use of unchecked data from an uncontrolled or tainted source	7.6
[EOJ]	Non-demarcation of control flow	6.28

[EWD]	Unstructured programming	6.31
[EWF]	Undefined behaviour	6.56
[EWR]	Path traversal	7.11
[FAB]	Implementation-defined behaviour	6.57
[FIF]	Arithmetic wrap-around error	6.15
[FLC]	Conversion errors	6.6
[GDL]	Recursion	6.35
[HCB]	Buffer boundary violation (Buffer overflow)	6.8
[HFC]	Pointer type conversions	6.11
[HJW]	Unanticipated exceptions from library routines	6.50
[HTS]	Resource names	7.12
[IHN]	Type system	6.2
[JCW]	Operator precedence and associativity	6.23
[KLK]	Distinguished values in data types	7.32
[KOA]	Likely incorrect expression	6.25
[LAV]	Missing initialization of variables	6.22
[LRM]	Extra intrinsics	6.45
[MEM]	Deprecated language features	6.58
[MVX]	Use of a one-way hash without a salt	7.24
[MXB]	Suppression of language-defined run-time checking	6.52
[NAI]	Choice of clear names	6.17
[NMP]	Pre-processor directives	6.51
[NSQ]	Library signature	6.49
[NYY]	Dynamically-linked code and self-modifying code	6.48
[OTR]	Subprogram signature mismatch	6.34
[OYB]	Ignored error status and unhandled exceptions	6.36
[PIK]	Using shift operations for multiplication and division	6.16
[PLF]	Floating-point arithmetic	6.4
[PPH]	Redispatching	6.43
[PYQ]	URL redirection to untrusted site (‘open redirect’)	7.8
[REU]	Fault tolerance and failure strategies	7.31
[RIP]	Inheritance	6.41
[RST]	Injection	7.9
[RVG]	Pointer arithmetic	6.12
[SAM]	Side-effects and order of evaluation of operands	6.24
[SHL]	Reliance on external format strings	6.64

[SKL]	Provision of inherently unsafe operations	6.53
[STR]	Bit representations	6.3
[SYM]	Templates and generics	6.40
[TEX]	Loop control variable abuse	6.29
[TRJ]	Argument passing to library functions	6.46
[UJO]	Modifying constants	6.65
[WPL]	Improper restriction of excessive authentication attempts	7.15
[WXQ]	Dead store	6.18
[XYH]	Null pointer dereference	6.13
[XYK]	Dangling reference to heap	6.14
[XYL]	Memory leaks and heap fragmentation	6.39
[XYM]	Insufficiently protected credentials	7.17
[XYN]	Adherence to least privilege	7.20
[XYO]	Privilege sandbox issues	7.21
[XYP]	Hard-coded credentials	7.16
[XYQ]	Dead and deactivated code	6.26
[XYS]	Executing or loading untrusted code	7.4
[XYT]	Cross-site scripting	7.7
[XYW]	Unchecked array copying	6.10
[XYZ]	Unchecked array indexing	6.9
[XZH]	Off-by-one error	6.30
[XZK]	Sensitive information not cleared before use	7.27
[XZL]	Discrepancy information leak	7.29
[XZN]	Missing or inconsistent access control	7.18
[XZO]	Authentication logic error	7.14
[XZP]	Resource exhaustion	7.13
[XZQ]	Unquoted search path or element	7.10
[XZR]	Improperly verified signature	7.23
[XZS]	Missing required cryptographic step	7.22
[XZX]	Memory locking	7.26
[YAN]	Deep vs shallow copying	6.38
[YOW]	Identifier name reuse	6.20
[YZS]	Unused variable	6.19

Annex B
(Informative)

Selected principles for language designers

B.1. These are principles that, if implemented in languages that do not currently implement them, will reduce or eliminate sources of common vulnerabilities. These principles are provided to the language developer community as possible input to their future standards. They were extracted from "*Implications for language design and usage"* from 6.X.6 in this document. Wording has been adjusted to provide a more general context, where applicable.

B.2 The following principles are addressed to the language-designer community at large:

1. Standardized terminology for type systems
    a. Standardization on a common, uniform terminology to describe type systems so that programmers experienced in other languages can reliably learn the type-system of a language that is new to them.
    b. Standardization on a common, uniform terminology to describe generics/templates so that programmers experienced in one language can reliably learn and refer to the type-system of another language that has the same concept, but with a different name.
2. Standardized calling of subprograms, or methods
    a. Standardization on provisions for inter-language calling.
    b. Standardization on how to indicate where parameter checks are performed.
    c. Support for static analysis, as static analysis of source code can eliminate much of the need for runtime checks.
3. Standardized fault handling
    a. Standardization on the terminology and means to perform fault handling.
    b. Standardization on a set of mechanisms for detecting and treating error conditions so that all languages to the extent possible could use them. This does not mean that all languages use the same mechanisms, but each of the mechanisms needs standardization.

B.3 The following principles may be useful to developers of individual language standards.

1. Standardized arithmetic adheres to recognized standard definitions, preferably:
    a. ISO/IEC/IEC 60559 .
    b. ISO/IEC 10967-1.
    c. ISO/IEC 10967-2.
2. Standardized conversions are type-safe:

a. By avoiding use of unchecked casts or marking them to be immediately recognizable as unsafe.

b. By providing mechanisms to prevent programming errors due to conversions.

3. Automatic bounds checks are performed:

a. By automatically checking bounds on accesses to array elements, unless the compiler or static analysis can statically determine that the check is unnecessary.

b. By providing whole array operations, such as full array assignment and safe copying of arrays, to obviate the need to access individual elements.

4 Subprograms, and in particular library-based subprograms, have contracts for callers

a. To formally specify preconditions and postconditions.

b. To specify the signatures of subprograms.

c. To provide the preconditions and postconditions for each call on functions in language-defined libraries so that function arguments can be validated during compilation, analysis by other static analysis tools, or during execution.

5 Overflow errors are detected and handled by specifying either an error, a saturated value, or a modulo result when numeric overflow occurs. Ideally, the selection among these alternatives is made by the programmer.

6 Undefined, unspecified and implementation-defined behaviours are minimized

a. By providing a list of undefined, unspecified and implementation-defined behaviours.

b. By minimizing the amount of unspecified and undefined behaviours.

c. By minimizing the number of possible behaviours for any construct with unspecified behaviour.

7 Use of deprecated features is diagnosed and they are optionally disabled, in particular when deprecated for security or safety reasons.

8 Synchronization among parallel/concurrent constructs are supported and primitives provided that let applications specify regions of sequential access to data using mechanisms such as protected regions, Hoare monitors, or synchronous message passing between code segments executing concurrently.

9 Termination of for loops is guaranteed and an iterator type for loop control is provided such that the value of the iterator is immutable within the body of the loop

Annex C
(informative)

Language-specific vulnerability template

Each language-specific Part has the following heading information and initial sections:

1 Scope

In addition to the standard ISO/IEC scope statement, add the following:

This document addresses how the vulnerabilities described in the language-independent writeup (in ISO/IEC 24772-1) are manifested in [*language*].

2 Normative references

Follow standard ISO/IEC guidelines.

3 Terms and definitions, symbols and conventions

Follow standard ISO/IEC guidelines.

4 Using this document

This document is intended to be used with ISO/IEC 24772-1 to detail how programming language vulnerabilities arise in the context of programming language [*language*].

ISO/IEC 24772-1 subclauses 4.1 and 4.2 describe how ISO/IEC 24772-1 is used and applied for the creation of software that is safe, secure and trusted within the context of the system that is fielded. The statements of ISO/IEC 24772-1 apply transitively to this document in the context of [*language*].

[If additional criteria are required for [*language*], then they may be stated here.]

5 General language concepts and primary avoidance mechanisms

5.1 Language concepts

[This sub-clause provides an overview of general terminology and concepts of [*language*] that are utilized throughout this document.]

5.2 Primary avoidance mechanisms

In addition to the generic avoidance mechanisms from ISO/IEC 24772-1 Clause 5.2, additional avoidance mechanisms from this section apply specifically to the programming language [*language*]. The recommendations of this section are restatements of recommendations from clause 6 of this document, but represent ones stated frequently, or that are considered as particularly noteworthy by the authors. clause 6 of this document contains the full set of avoidance mechanisms, as well as explanations of the related problems.

[Following this statement, provide a table that provides the most common (approximately 10) or most important avoidance mechanisms that are not provided in ISO/IEC 24772-1 clause 5.2. The format of the table is rule number (sequential), the rule itself, and references to subclause 6.x.2, where the rule is relevant.]

6 Language vulnerabilities

[Address every vulnerability description of clause 6 of the main document in this document in the same order even if there is simply a notation that it is not relevant to the language in question. Each vulnerability description has the following format:]

6.x <Vulnerability name> [<3 letter tag>]

6.<x>.1 Applicability to [*language*]

[This section states the applicability of the vulnerability from ISO/IEC 24772-1 clause 6.X, to [*language*]. The following cases are to be considered:

1. If [*language*] prevents all facets of the vulnerability as described in ISO/IEC 24772-1 clause 6.X, state that the vulnerability *does not apply* and provide a sound but brief rationale for that statement. In such a case, if there is no further vulnerability, this statement may be the only statement in clause 6.X.

2. If [*language*] provides strong mitigations to help the developer avoid the vulnerability, then state in 6.X.1 that [*language*] *mitigates* the vulnerability as described in ISO/IEC 24772-1 clause 6.X, together with rationale for the mitigation statement and an explanation of remaining actions or behaviours needed to eliminate the vulnerability.

3. Otherwise, state that the vulnerability as described in ISO/IEC 24772-1 clause 6.X *applies* to [*language*] and explain how it applies. Provide a description of additional required actions to prevent the vulnerability.

4. If the vulnerabilities described in ISO/IEC 24772-1 clause 6.X do not apply, except for a remaining corner case, one may use the shorter terminology “The vulnerabilities *do not apply except* ...”

Compile-time rejection or run-time checks as means to prevent vulnerabilities qualify for the “does not apply” case. In the case of run-time checks, add the standard reference (after adjusting it to the terminology of [*language*]: “The vulnerability associated with [u*nhandled errors*] is discussed in clause 6.36 “Ignored error status and unhandled exceptions [OYB]”).

Rules to observe:

a. Ensure that each vulnerability in Part 1 is addressed by an explicit statement as described above. (Part 1 groups closely related vulnerabilities. Deal with each one.).

b. In considering a vulnerability, abstract Part 1 sufficiently to address all facets of the relevant issue (Example: if Part 1 presents a vulnerability about pointers, one cannot deny the vulnerability merely because [language] does not have pointers, while it provides references or labels that expose analogous vulnerabilities).

c. In clause 6, assume that the reader knows [*language*]. If it is judged that tutorial text is very important, put terminology in clause 3 and relevant descriptions in clause 5.1. An exception to this rule is the short description for a feature completely dedicated to the vulnerability at hand and of little or no consequence otherwise.

d. Keep rationales short, particularly the ones for non-applicability of a vulnerability.

e. Do not describe language features not directly related to the vulnerability at hand or not related to specific advice for its avoidance.

f. Prefer informal, but comprehensible rationale to precise, but difficult-to-understand reference manual descriptions.

g. The document is not a place to advertise [*language*] and its features. If necessary, describe them briefly and in factual terms. Subjective qualifiers common in marketing literature and subjective statements will be deleted.

h. “Programmers do not do this” is not a valid argument in this document.

i. “Software Engineering principles prevent this” is not a valid argument in this document, unless prevention is strictly enforced by the language. Formulate it as a mitigation, instead.

j. “Tool X prevents this” is not a valid argument in this document, unless use of the tool is mandatory for every program in [*language*]. Formulate tool usage as a mitigation, instead.

k. Never compare with other languages.

l. Do not cite specific products.

6.<x>.2 Avoidance mechanisms for language users

[If the vulnerability is mitigated or if it applies, describes what the programmer or user can do in order to avoid or eliminate the vulnerability.

Rules to observe:

1. Formulate full sentences that can be understood in isolation by experts.

2. Use gradations to indicate the strength of the advice (on a scale from “enforce, prohibit, and mandate” via “avoid and prefer” to “consider and examine“).

3. Provide necessary technical details supporting or explaining the advice in 6.<x>.1.

4. Do not include justifications in the advice itself.]

Following the final vulnerability description, optionally provide sub-clauses as follows:

7. Language specific vulnerabilities for [*language]*

[This section is where vulnerabilities not covered by ISO/IEC 24772-1 will be placed. It is possible that there are none for any given language.]

8 Implications for standardization or future revision

[This section provides the opportunity to discuss changes anticipated for future versions of the language specification. The section may be vacant.]

Bibliography

[1] *Ada Quality and Style and Guide, Guidelines for professional programmers.* https://en.wikibooks.org/wiki/Ada_Style_Guide

[2] *ARIANE 5: Flight 501 Failure*, Report by the Inquiry Board, July 19, 1996 https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[3] ARINC 653. Avionics Application Software Standard Interface. SAE International https://www.sae.org/standards/content/arinc653p0-3/

[4] *ASCII Codes table.* http://ascii.cl

[5] Burns, Alan and Wellings, Andy. *Real-Time Systems and Programming Languages: Ada, Real-time Java and C/Real-Time POSIX (4th Edition),* Addison Wesley 2009

[6] CERT. *CERT C++ Secure Coding Standard*. https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard (2016).

[7] CWE. *The Common Weakness Enumeration (CWE) Initiative*, MITRE Corporation, (https://cwe.mitre.org/)

[8] Dionisio, John David N, *Type Checking*[1].

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

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

[11] Hatton, Les, *Safer C: developing software for high-integrity and safety-critical syste*ms. McGraw-Hill 1995

[12] Hoare C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985

[13] Hogaboom, Richard, *A Generic API Bit Manipulation in C*, Embedded Systems Programming, Vol 12, No 7, July 1999 https://www.embedded.com/1999/9907/9907feat2.htm (Link Broken) still exists on site)

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

[15] Holzmann, Garard J., *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004

[1] http://myweb.lmu.edu/dondi/share/pl/type-checking-v02.pdf

[16] ISO/IEC 1539-1, *Information technology — Programming languages — Fortran — Part 1: Base Language*

[17] ISO/IEC 8652, *Information technology — Programming languages — Ada*

[18] ISO/IEC 9899, *Information technology — Programming languages — C*

[19] ISO/IEC 14882, *Information technology — Programming languages — C++*

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

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

[22] ISO/IEC/TR 24731-1, *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces*

[23] *Jones, Derek M "Developer beliefs about binary operator precedence". Association of C and C++ Users, CVu. 18 (4): 14–21.*

[24] *JSF, Joint Strike Fighter Air Vehicle: C++ Coding Standards for the System Development and Demonstration Program*. Lockheed Martin Corporation. December 2005.

[25] Kopetz, Hermann. *Real-Time Systems: Design Principles for Distributed Embedded Applications,* Springer 2011

[26] Larsen, Petersen, Wang, *Model Checking for Real-Time Systems*, Proceedings of the 10th International Conference on Fundamentals of Computation Theory, 1995

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

[28] Lundqvist, K. and Asplund, L, *A formal model of a run-time kernel for Ravenscar*, Proceedings of the Sixth International Conference on Real-Time Computing, December 1999

[29] MISRA Limited. *MISRA C*: 2012 *Guidelines for the Use of the C Language in Critical Systems*. Warwickshire, UK: MIRA Limited, March 2013 (ISBN 978-1-906400-10-1 and 978-1-906400-11-8).

[30] MISRA, Motor Industry Software Reliability Association. *Guidelines for the Use of the C++ Language in critical systems*, June 2008

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