Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 3 – Vulnerability descriptions for the programming language C

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

Document type: International technical report
Document subtype: if applicable
Document stage: publication stage
Document language: E

Deleted: TR

Contents

Page

Deleted: 15
Deleted: 15
Deleted: 15
Deleted: 16
Deleted: 17
Deleted: 18
Deleted: 19
Deleted: 21
Deleted: 22
Deleted: 24
Deleted: 24
Deleted: 25
Deleted: 26
Deleted: 26
Deleted: 27
Deleted: 28
Deleted: 30
Deleted: 30
Deleted: 31
Deleted: 31
Deleted: 31
Deleted: 32
Deleted: 32
Deleted: 33
Deleted: 33
Deleted: 34
Deleted: 35
Deleted: 36
Deleted: 37
Deleted: 38
Deleted: 39

Deleted: TR

Deleted: 39
Deleted: 40
Deleted: 41
Deleted: 42
Deleted: 43
Deleted: 43
Deleted: 44
Deleted: 44
Deleted: 45
Deleted: 45
Deleted: 45
Deleted: 45
Deleted: 45
Deleted: 46
Deleted: 46
Deleted: 46
Deleted: 46
Deleted: 47
Deleted: 47
Deleted: 48
Deleted: 48
Deleted: 49
Deleted: 49
Deleted: 49
Deleted: 50
Deleted: 50
Deleted: 51
Deleted: 51
Deleted: 52
Deleted: 52
Deleted: 52
Deleted: 53
Deleted: 53
Deleted: 53
Deleted: 54
Deleted: 55
Deleted: 56

Foreword

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

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

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

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

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

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

This document replaces ISO/IEC 24772-3:2019. The main changes between this document and Annex D of TR 24772:2013 are:

- A new clause 4 *Using this document* has been added to explain how the document relates to ISO/IEC 24772-1 and how it is expected to be used by organizations and language users.
- The material in clause 5 in ISO/IEC TR 24772-3 clause 4 *Language concepts* is moved to 5.1.
- Recommendations to avoid vulnerabilities are ranked and the top 10 are placed in a table in subclause 5.2, together with the vulnerabilities in clauses 6 that contain each recommendation.
- The following vulnerabilities were renamed to track the changes made in Part 1:
  - [HFC] *Pointer casting and pointer type changes* was renamed to *Pointer type conversion*;
  - [JCW] *Operator precedence/Order of evaluation* was renamed to *Operator precedence and associativity*;
  - [XYL] *Memory leak* is renamed to *Memory leaks and heap fragmentation*
  - [LAV] *Initialization of variables* is renamed to *Missing initialization of variables*;
  - [CLL] *Switch statements and static analysis* was renamed to *Switch statements and lack of static analysis*;
  - [EOJ] *Demarcation of control flow* is renamed to *Non-demarcation of control flow*;
  - [TEX] *Loop control variables* is renamed to *Loop control variable abuse;*
- New vulnerabilities are added, to match the additions of Part 1:

Deleted: may
Deleted: may
Deleted: I
Deleted: TR 24772
Deleted: With the cancellation of TR 24772:2013, t
Deleted:
Deleted: TR
Deleted: 2 Annex D
Deleted: ¶
R
Deleted: ¶
The following vulnerabilities that were documented in clause 8 of TR 24772:2013 are now addressed in this document in clause 6.¶
[CGA] *Concurrency – Activation*¶
[CGT] *Concurrency – Directed termination*¶
[CGX] *Concurrent data access*¶
[CGS] *Concurrency – Premature termination*¶
[CGM] *Protocol lock errors is now Lock protocol errors*¶
[CGY] *Inadequately secure communication of shared resource.*¶
Deleted: <#>Clauses 6.2 *Terminology* is integrated into clause 3, and all subclauses in clause 6 are renumbered.¶
The following vulnerabilities were removed:¶
[XZI] *Sign extension error* was integrated into [XTR] *Type system*.¶
[REU] *Termination strategy*, 6.39, is placed in clause 7 in Part 1, and hence is not documented for C herein.¶
Deleted: ;¶
Formatted: Font: Italic
Formatted: Font: Italic
Formatted: Font: Italic
Formatted: Font: Italic
Deleted: [XYP] *Hard coded password* is renamed *Hard coded credentials*;...
Formatted: Font: Italic
Formatted: Font: Not Italic
Formatted: Font: Italic

- [UJO] Modifying constants
- Avoidance mechanisms for each vulnerability given in subclause 6.X.2 is reworded to be more explicit and directive, and to be clear that these are avoidance mechanisms and not “guidance”.

Addition material has been added for some vulnerabilities to reflect addition knowledge gained since the publication of ISO/IEC TR 24772-1:2019 and IS/IEC TR 24772-3.

Deleted: TR

Deleted: <#>[YAN] *Deep vs shallow copying;*
[BLP] *Violations of the Liskov substitution principle or the contract model;*
[PPH] *Redispatching;*
[BKK] *Polymorphic Variables;*
[SHL] *Reliance on external format strings;*

Deleted: Guidance

Deleted: material

Deleted: .

Deleted: 3

Introduction

This document provides guidance for the programming language C [2,6], so that application developers considering or using C will be better able to avoid the programming constructs that lead to vulnerabilities and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate such constructs in their software, or the developers of such tools.

This document is intended to be used with ISO/IEC 24772–12024, which discusses programming language vulnerabilities in a language independent fashion.

Deleted: TR

It should be noted that this document is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such report can only describe those that have been found, characterized, and determined to have sufficient probability and consequence. The guidance in this document has been drawn from existing safety and security coding rules [7..15].

Programming Languages — Guidance to avoiding vulnerabilities in programming languages — Part 3: Vulnerability descriptions for the programming language C

1. Scope

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

This document describes the way that the vulnerabilities listed in the language-independent ISO/IEC 24772–1:2024 are manifested or avoided in the C language and catalogues avoidance mechanisms that can be used to produce safer C programs.

Deleted: TR

Deleted: .

2. Normative references

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

ISO/IEC 24772-1 - *Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages*
ISO/IEC 9899:2011 — *Information Technology — Programming languages — Programming Languages—C*
ISO/IEC 9899:2011/Cor. 1:2012 — *Information Technology — Programming languages — Programming languages — C*

3. Terms and definitions

Deleted: , symbols and conventions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, in ISO/IEC 24772–1, in ISO/IEC 9899:20xx and the following apply. Other terms are defined where they appear in *italic* type.

Deleted: 3.1 Terms and definitions¶

Deleted: TR

Deleted: 11

The following terms are in alphabetical order, with general topics referencing the relevant specific terms.

3.2
access
read or modify the value of an object

Note: Modify includes the case where the new value being stored is the same as the previous value. Expressions that are not evaluated do not access objects.

3.3
alignment
requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

3.4
argument
expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

Note 1: Also called actual argument.
Note 2: An argument replaces a *formal parameter* as the call is realized.

3.5
behaviour
external appearance or action

Note: See: implementation-defined behaviour, locale-specific behaviour, undefined behaviour, unspecified behaviour

3.6
bit
unit of data storage in the execution environment large enough to hold an object that can have one of two values

Deleted: may

Note: It need not be possible to express the address of each individual bit of an object.

3.7
byte
addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

Note: It is possible to express the address of each individual byte of an object uniquely. A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit.

3.8
character
abstract member of a set of elements used for the organization, control, or representation of data

Note: See: single-byte character, multibyte character , wide character

3.9
correctly rounded result

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

3.10
diagnostic message
message belonging to an implementation-defined subset of the implementation's message output

Note: The C Standard requires diagnostic messages for all constraint violations.

3.11
formal parameter
object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

3.12
implementation
particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

3.13
implementation-defined behaviour
behaviour where multiple options are permitted by the standard and where each implementation documents how the choice is made

Note: An example of implementation-defined behaviour is the propagation of the high-order bit when a signed integer is shifted right. Implementation-defined behaviours are listed in the C language standard [2] Annex J.3.

3.14
implementation-defined value
value not specified in the standard where each implementation documents how the choice for the value is selected

3.15
implementation limit
restriction imposed upon the program by the implementation

3.16
indeterminate value
unspecified value or a trap representation

3.17
locale-specific behaviour

behaviour that depends on local conventions of nationality, culture, and language that each implementation documents

Note: An example of locale-specific behaviour is whether the `islower()` function returns true for characters other than the 26 lower case Latin letters

3.18

memory location

object of scalar[1] type, or a maximal sequence of adjacent bit-fields all having nonzero width

3.19

multibyte character

sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment, where the extended character set is a superset of the basic character set

3.20

object

region of data storage in the execution environment, the contents of which can represent values

3.21

parameter

actual argument, argument, or formal parameter

3.22

recommended practice

specification that is strongly recommended as being in keeping with the intent of the C Standard, but that can be impractical for some implementations

Deleted: may

3.23

runtime-constraint

requirement on a program when calling a library function

3.24

sequence point

point in the language syntax where the compiler guarantees that all calculations and assignments required by the code preceding the sequence point are completed, before those following it are started

Note: The comma operator is a sequence point. Hence in `A, B;` all calculations and assignments required by sub-expression `A` are completed before any required by `B` are started.

---

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

3.25
single-byte character
bit representation that fits in a byte

3.26
trap representation
object representation that need not represent a value of the object type

3.27
undefined behaviour
use of a non-portable or erroneous program construct or of erroneous data, for which the C standard imposes no requirements

> Note: Undefined behaviour ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). An example of undefined behaviour is the behaviour on integer overflow. Undefined behaviours are listed in the C language standard, ISO/IEC 9899 Annex J.2.

3.28
unspecified behaviour
use of an unspecified value, or other behaviour where the C Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

> Note: For example, unspecified behaviour is the order in which the arguments of a function are evaluated. Unspecified behaviours are listed in the C language standard, ISO/IEC 9899 Annex J.1.

3.29
unspecified value
valid value of the relevant type where the C Standard imposes no requirements on which value is chosen in any instance

> Note: An unspecified value cannot be a trap representation.

3.30
value
meaning of the contents of an object when interpreted as having a specific type

> Note: See implementation-defined value, indeterminate value, unspecified value, trap representation

3.31
wide character
bit representation capable of representing any character in the current locale::

Note: The C Standard uses the name `wchar_t` for objects of this type

4. Using this document

ISO/IEC 24772-1:2024 4.2 documents the process of creating software that is safe, secure and trusted within the context of the system in which it is fielded. As this document shows, vulnerabilities exist in the C programming environment, and organizations are responsible for understanding and addressing the programming language issues that arise in the context of the real-world environment in which the program will be fielded.

Organizations following this document meet the expectations of 4.2 of ISO/IEC 24772-1, repeated here for the convenience of the reader, in the context of the C programming language environment and tools:

- 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 5.2, 6 and 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 addition, organizations can determine avoidance and mitigation mechanisms using clause 6 of this document as well as other technical documentation, such as the MITRE Corporation, Common Weakness Enumeration (CWE) [8], Sun Microsystems, Inc. [18], and Einarsson [2]. Other views of avoiding programming mistakes and design flaws are addressed by Martelli [13] and Sebesta[17].

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

Programmers and software designers follow this document by following the architectural and coding guidelines of their organization, and by choosing appropriate mitigation techniques when a vulnerability is not avoidable.

Formatted: Font: 12 pt
Formatted: Font: 12 pt
Formatted: Font: 12 pt
Formatted: Font: 11 pt
Formatted: Font: 11 pt
Formatted: Font: 11 pt
Formatted: Font: 11 pt
Formatted: Font: 11 pt
Formatted: Font: 12 pt
Formatted: Font: Cambria, 12 pt
Formatted: Normal
Formatted: Font: 12 pt

5.General language concepts and primary avoidance mechanisms

Deleted: Language concepts

5.1 General C language concepts

Formatted: Heading 2

The C programming language was developed in the early 1970's at Bell Labs, in support of the development of the Unix operating system. Its first published specification was in 1978 in the book "The C programming language" [1]. The first ISO standard for C was published in 1990 and updated in 1999 and 2011.

C is an imperative language that supports structured programming and has a static type system. It has often been described as a 'high-level assembler', in that the semantic gap between a program and the executable code is small (as in a traditional assembler), but having the advantages of a high-level language: machine independence and structured programming control constructs.

The small semantic gap between program and executable code means that the resulting executables are compact and fast, making C a popular language for developing operating systems and embedded applications. There is a desire to maintain this advantage of the language. Consequently, as the language has developed there is a strategy of avoiding the addition of overheads that do not directly contribute to the behaviour of the application and to maintain backwards compatibility, as embedded systems in particular can be in development and maintenance for a very long time. This document proposes restrictions that should be imposed on development in an environment where run-time failure is unacceptable.

Some key features of the language are:

- Due to C being a 'high-level assembler' and having been around for longer than most other high-level languages, it has become a common exchange format between other languages. In particular, many languages implement the C function calling model (at least as a selectable option), so that third party libraries can be used in many language environments.
- C has a particularly close relationship with C++. Initially C++ was a strict superset of C, with only one exception of a feature in C not being in C++. Whilst over the years there has been some divergence, the relationship is still close.
- An unusual feature of C is the preprocessor. This allows textual manipulation of the code before the compiler considers the program. It is used to allow changes to the code to match specific implementation environments, implement in-line functions and implement code 'short-cuts' by allowing component statements to be constructed that would not be syntactically legal using a function definition.
- Since ISO/IEC 9899:2011 [2], the language has had a native threading model. Previously, parallelism was only achieved using third-party libraries not included in the standard.
- Unlike some other languages, in C the terms 'pass by reference', 'pass by pointer', 'pass by address' have the same meaning.

5. Top avoidance mechanisms

Deleted: Avoiding programming language vulnerabilities

Deleted: in C

In addition to the generic programming rules from ISO/IEC 24772-1:2024 5.4, additional rules from this section apply specifically to the C programming language. The recommendations of this section are restatements of

Deleted: TR

Deleted: 2019 clause

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 recommendations, as well as explanations of the problems that led to the recommendations being made.

Table 1: Top avoidance mechanisms in C

Formatted: Heading 3, Centered, Indent: Left: 0 cm, Widow/Orphan control, Don't suppress line numbers, Allow hanging punctuation

Index		Reference
1	Use a macro to ensure that the size of memory allocated with malloc matches the intended type of the object.	6.11 Pointer type conversions [HFC]
2	Use bounds checking interfaces from Annex K of ISO/IEC 9899:2011 [2] in favour of non-bounds checking interfaces, such as `strcpy_s` instead of `strcpy`.	6.8 Buffer boundary violation (buffer overflow) [HCB]
3	Use commonly available functions such as the POSIX [4] functions `htonl()`, `htons()`, `ntohl()` and `ntohs()` to convert from host byte order to network byte order and vice versa.	6.3 Bit representations [STR]
4	Perform range checking before copying memory (using mechanisms such as `memcpy` and `memmove`), unless it can be shown that a range error cannot occur. Bounds checking is not performed automatically, but in the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.	6.10 Unchecked array copying [XYW]
5	Check that a pointer is not null before dereferencing, unless it can be shown statically that the pointer cannot be null.	6.13 Null pointer dereference [XYH]

6	After a call to `free`, set the pointer to `null` to prevent multiple deallocation or use of a dangling reference via this pointer, as illustrated in the following code:
	`free (ptr);`
	`ptr = NULL;`
	.
		6.14 Dangling reference to heap [XYK]

7	Do not read uninitialized memory, including memory allocated by functions such as malloc.	6.22 Missing initialization of variables [LAV]

8	Check that the result of an operation on an unsigned integer value will not cause wrapping, unless it can be shown that wrapping cannot occur, or document and verify the intended behaviour.
	Any of the following operators have the potential to wrap:
	`a + b   a – b   a * b   a++   ++a   a--   --a`
	`a += b   a -= b   a *= b   a << b   a<<=b   -a`
		6.15 Arithmetic wrap-around error [FIF]

9	Check that the result of an operation on a signed integer value will not cause overflow, unless it can be shown that overflow cannot occur. Any of the following operators have the potential to overflow, which is undefined behaviour in C: `a + b   a – b   a * b   a/b   a%b   a++   ++a   a--   --a`	6.15 Arithmetic wrap-around error [FIF]

Deleted: 6.11 Pointer type conversions [HFC]

Deleted: 6.8 Buffer boundary violation (buffer overflow) [HCB]

Deleted: 6.3 Bit representations [STR]

Deleted: 6.10 Unchecked array copying [XYW]

Deleted: 6.13 Null pointer dereference [XYH]

Deleted: 6.14 Dangling reference to heap [XYK]

Deleted: 6.22 Initialization of variables [LAV]

Deleted: 6.15 Arithmetic wrap-around error [FIF]

Deleted: 6.15 Arithmetic wrap-around error [FIF]

	`a += b   a -= b   a *= b   a /= b    a %= b` `a << b   a <<= b   -a`	
10	Ensure that a type conversion results in a value that can be represented in the resulting type.	6.6 Conversion errors [FLC]

6. Specific Guidance for C Vulnerabilities

6.1 General

This clause contains specific advice for C about the possible presence of vulnerabilities as described in ISO/IEC 24772-1:2024, and provides specific avoidance mechanisms for C programs. This section mirrors ISO/IEC 24772-1:2024 clause 6 in that the vulnerability "Type System [IHN]" is found in 6.2 of ISO/IEC 24772–1, and C specific guidance is found in clause 6.2 and subclauses of this document.

6.2 Type system [IHN]

6.2.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.2 is applicable to C.

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

C allows implicit conversions as in the following example:

```
short a = 1023;
int b;
b = a;
```

If an implicit conversion can result in truncation of the value, such as in a conversion from a 32-bit int to a 16-bit short int:

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

many compilers will issue a warning message.

Deleted: 6.6 Conversion errors [FLC]

Deleted: TR

Deleted: guidance

Deleted: on how to avoid them in

Deleted: code

Deleted: TR

Deleted: 2019

Deleted: TR

Formatted: Font: Cambria, 12 pt, Not Bold

Formatted: Font: Cambria, 12 pt, Not Bold

Formatted: Font: (Default) Courier New, 10.5 pt

Formatted: python Char, Font: +Headings (Cambria), 10.5 pt, Bold

Formatted: Font: (Default) Courier New, 10.5 pt

Formatted: Font: (Default) Courier New, 10.5 pt

Formatted: Font: (Default) Courier New, 10.5 pt

Deleted: may

Formatted: Font: Not Bold

C has a set of rules to determine how conversion between data types will occur. For instance, every integer type has an integer conversion rank that determines how conversions are performed. The ranking is based on the concept that each integer type contains at least as many bits as the types ranked below it. So even though there are rules in place and the rules are rather straightforward, the variety and complexity of the rules can cause unexpected results and potential vulnerabilities.

6.2.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2019 6.2.5.
- Be aware of the rules for typing and conversions to avoid vulnerabilities.
- Do not cast to an inappropriate type.
- Enable compiler warnings regarding implicit conversions or use static analysis tools that provide such warnings.

Deleted: Guidance to language users
Formatted: Normal, Space After: 0 pt
Deleted: Follow the guidance
Deleted: TR 24772
Deleted: clause

6.3 Bit representations [STR]

6.3.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.3 is applicable to C. C supports a variety of sizes for integer types such as `short int`, `int`, `long int` and `long long int`. Each integer type is either signed or unsigned. C also supports a variety of bitwise operators that facilitate bit manipulations, such as left and right shifts and bitwise `&` and `|`. Some bit manipulations can cause unexpected results through miscalculated shifts or platform dependent variations.

Deleted: ¶
Formatted: Normal
Deleted: may
Deleted: be

For instance, right shifting a signed integer is implementation defined in C, while shifting by an amount greater than or equal to the size of the data type is undefined behaviour. For instance, on a host where an int is of size 32 bits,

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

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

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

Deleted: may be in l

6.3.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Deleted: ¶
6.3.2 Guidance to language users

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2019 6.3.5.
- Only use bitwise operators on unsigned integer values as the results of some bitwise operations on signed integers are implementation defined or undefined.
- Where available, use functions such as the POSIX [4] standard functions `htonl()`, `htons()`, `ntohl()` and `ntohs()` to convert from host byte order to network byte order and vice versa. This would be needed to interface between an i80x86 architecture, where the Least Significant Byte is first, and devices with network byte order, as used on the Internet, where the Most Significant Byte is first. Use bitwise operations only as a last resort.
- In cases where there is a possibility that a shift is greater than the size of the variable, perform a check as the following example shows, or a modulo reduction before the shift:

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

6.4 Floating-point arithmetic [PLF]

6.4.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.4 is applicable to C as C permits the floating-point data types float, double and long double. Due to the approximate nature of floating-point representations, the use of floating-point data types in situations where equality is to be tested or where rounding accumulates over multiple iterations can lead to unexpected results and potential vulnerabilities.

As with most data types, C is flexible in how float, double and long double can be used. For instance, C allows the use of floating-point types to be used as loop counters and in equality statements, even though in most cases these will not have the expected behaviour. For example

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

can or might not terminate after 10,000,000 iterations. The representations used for `x` and the accumulated effect of many iterations can cause `x` to never be identical to 1.0 causing the loop to continue to iterate forever.

Similarly, the Boolean test

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

can or might not evaluate to true. Given that `x` and `y` are constant values, it is expected that consistent results will be achieved on the same platform. However, it is questionable whether the logic performs as expected when a float that is twice that of another is tested for equality when divided by 2 as above.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: clause
Deleted: may
Deleted: may
Deleted: may
Deleted: may
Deleted: may

6.4.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2019 6.4.5:

Deleted: Guidance to language users

Deleted: Follow the guidance

Deleted:

Deleted: TR 24772

Deleted: clause

6.5 Enumerator issues [CCB]

6.5.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.5 is applicable to C

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

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

The values of the members of `abc` would be A=0, B=1, C=2, and so on. C allows explicit values to be assigned to the enumeration type members, so that the member is assigned the indicated value and the next member will take the next value (unless also explicitly assigned a value).

So the declaration:

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

is equivalent to:

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

Note that this has gaps in the sequence of values and repeated values.

There are a number of issues that can arise with enumeration types:

- C treats enumeration members identically to integers. So an enumeration member can be used in an integer expression (using its associated value) and an integer can be assigned to an enumeration type object, even if there is no member associated with that value. This becomes an issue if an enumeration type object is used to control a switch statement if a switch statement is controlled by a value of type `abd`, where `abd` is defined as:

  ```
  enum abd {First, Second, Third, Fourth, Fifth, Sixth, Seventh, Eighth};
  ```

  and the `switch` statement has eight case clauses, for `case First:` to `case Eighth:` then there are two scenarios where the switch might not behave as expected:

  Deleted: may

  - the user expects all possible values to be covered. However, if the control expression is a variable assigned `Eighth+1`, then the code will 'fall through', without executing any of the case statements

    Deleted: may

  - the user can address the above issue by providing a default clause. However, in the safety domain, it is common practice to provide a default clause even if the code (apparently) can only ever have enumeration member values for the control expression. This protects against unexpected corruption of the control variable, say by a buffet overrun. However, if the compiler also thinks the control value can only ever be one of the enumeration members, it is permitted to optimize away the default clause, meaning that the expected protection possibly does not exist.

    Deleted: may

- If the code has initially been written using the default assignment of values (0..Number of members – 1), and an array is declared with bounds `[Last_member + 1]`, this array has one element for each enumeration type member. If maintenance of the code then occurs that modifies the assignment of values, two issues can arise:

- a member is created that has a value greater than `Last_member`'s, so there will be undefined behaviour if this member is used to index the array, or
- the values covered by the modified enumeration type members do not form a continuous sequence from 0 to Number of members –1, with either gaps in the sequence or repeated values. If the members are used to initialize and access the array, then some members of the array will remain uninitialized if there are gaps. If some final processing is performed on the array, using an integer count from 0 to Number of members –1, again there is likely to be undefined behaviour. If there are repeated values, the result is unlikely to be that which was expected.

Deleted: may
Deleted: be
Deleted: ,
Deleted: may

6.5.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space Before: 0 pt, After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.5.5.
- Create enumeration type declarations following one of the following three formats:
  - no explicit values:
    ```
    e.g.  enum abc {A,B,C,D,E,F,G,H} var_abc;.
    ```
  - a single explicit value for the first member:
    ```
    e.g. enum abc {A=5,B,C,D,E,F,G,H} var_abc; /*rest follow numerically*/
    ```
  - all values explicit:
    ```
    e.g. enum abc {
                          A=0,
                          B=1,
                          C=6,
                          D=7,
                          E=8,
                          F=7,
                          G=8,
                          H=9} var_abc;
    ```
- Avoid using loops that iterate over an enum that has representation specified for the enums, unless it can be guaranteed that there are no gaps or repetition of representation values within the enum definition.
- Use an enumerated type to select from a limited set of choices to make possible the use of tools to detect omissions of possible values such as in switch statements.
- If a 'precautionary' default statement is added to switch statement controlled by an enumeration type, make the controlling object volatile, so the compiler cannot optimize it away (arguably, a compliant compiler shouldn't optimize it away, but a number of them have been found that do).

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

6.6 Conversion errors [FLC]

6.6.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.46is applicable to C

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

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

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

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

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

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

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

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

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

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

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

```
signed long int sl = LONG_MAX;
signed char sc = (signed char)sl;
```

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

A recent innovation from ISO/IEC TR 24731-1 [3] that has been added to the C standard 9899:2011 [2] is the definition of the `rsize_t` type. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. Also, some implementations do not support objects as large as the maximum value that can be represented by type `size_t`. For these reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces can define `RSIZE_MAX` as `SIZE_MAX`, which means that there is no object size that is considered a runtime-constraint violation.

6.6.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space Before: 0 pt, After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.6.5.
- Check the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type. Any conversion from a type with larger range to a smaller range can result in a loss of data. In some instances, this loss is desired. Such cases should be explicitly acknowledged in comments. For example, the following code can be used to check whether a conversion from an unsigned integer to an unsigned character will result in truncation:

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

- Close attention should be given to all warning messages issued by the compiler regarding multiple casts. Making a cast in C explicit will both remove the warning and acknowledge that the change in value is intended.
- If mixed types are used in an expression, ensure that each conversion preserves the value before being used as an operand in another operation in the same expression.
- When converting between wide character and multi-byte characters and strings, always use the appropriate conversion functions (`wctomb` and `wcsrtombs` or `wcsrtombs_s` respectively). Similarly for multi-byte to wide characters and strings use `mbrtowc` and `mbsrtowcs` or `mbsrtowcs_s`

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

Deleted: may

6.7 String termination [CJM]

6.7.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.7 is applicable to C.

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

6.7.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.7.5.
- Use the safer and more secure functions for string handling that are defined in normative Annex K[2] from ISO/IEC 9899:2011 [2] or the ISO TR 24731-2 — *Part II: Dynamic allocation functions* [3]. Both of these define alternative string handling library functions to the current Standard C Library. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated. One implementation of these functions has been released as the Safe C Library.

6.8 Buffer boundary violation (buffer overflow) [HCB]

6.8.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.8 is applicable to C .

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

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

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

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

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

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

[2] See comments on the correct use of Annex K functions in 6.8.1 Buffer boundary violation 6.8 Buffer boundary violation (buffer overflow) [HCB]

Deleted: Guidance to language users
Formatted: Normal, Space Before: 0 pt, After: 0 pt
Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause
Deleted: may
Deleted: 6.8 Buffer boundary violation (buffer overflow) [HCB]


```
    }
  else {
    return x[i];
    }
  }
```

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

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

The `buffer_src` is longer than the `buffer_dest`, and the code does not check for this before the actual copy operation is invoked. A safer way to accomplish this copy would be to use `strncpy`, that can be limited to copy a maximum number of characters:

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

This would not cause a buffer bounds violation, however, because the destination buffer is smaller than the source buffer, the destination buffer will now hold "abcd". Note that the final member of `buffer_dest` is explicitly assigned the terminator value. `strncpy` does not automatically terminate strings if longer than the indicated number of characters, so this manual assignment to the last character of the destination buffer should always be made.

A further alternative is to use the equivalent function from normative annex K of ISO/IEC 9899:2011 [2] *Bounds-checking interfaces*:

```
char buffer_src[]={"abcdefg"];
char buffer_dest[5]={0};
if ( strcpy_s(buffer_dest, sizeof(buffer_dest), buffer_src) )
   { /* Error Handler */ }
```

If the source string including the terminator is smaller than the indicated destination buffer size, then the source string is copied to the destination buffer. If, as in the example, the source string is too big, the first element of the destination string is assigned 0 (i.e. the destination becomes an empty string). Note that `strcpy_s` and related functions return 0 on success and a non-zero value on errors. When calling these functions, the error value should always be checked.

6.8.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.8.5.
- Validate all input values.
- Check any array index before use if there is a possibility that the value can be outside the bounds of the array.

Deleted: Guidance to language users

Formatted: Normal, Space Before: 0 pt, After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

- Use the safer and more secure functions for string handling from the normative annex K of ISO/IEC 9899:2011 [2], *Bounds-checking interfaces*, but always check each call for a returned error condition.
- Alternatively, use length restrictive functions such as `strncpy()` instead of `strcpy()`, unless it can be shown the destination buffer is big enough, and noting the requirement to ensure to destination string is terminated. Also note that this can lead to truncation of the source string in the receiving buffer.
- Use stack guarding add-ons to detect overflows of stack buffers.
- Do not use the deprecated functions, such as gets().

Deleted: may

6.9 Unchecked array indexing [XYZ]

6.9.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.9 is applicable to C since C does not perform bounds checking on arrays, so although arrays can be accessed outside of their bounds[3], the value returned is undefined and in some cases can result in a program termination. For example, in C the following code is valid, though, for example, if `i` has the value 10 it leads to unspecified behaviour:

Deleted: may

Deleted: may

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

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

6.9.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space Before: 0 pt, After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.9.5.
- Perform range checking before accessing an array. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
- Use the safer and more secure functions for string handling from the normative annex K of ISO/IEC 9899:2011 [2], *Bounds-checking interfaces*[4]. These are alternative string handling library functions. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated.

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

---

[3] In effect, this is a special case of 6.8 Buffer boundary violation (buffer overflow) [HCB].

Deleted: 6.8 Buffer boundary violation (buffer overflow) [HCB]

[4] See comments on the correct use of Annex K functions in 6.8.1 Buffer boundary violation [HCB]

6.10 Unchecked array copying [XYW]

6.10.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.10 is applicable to C. A buffer overflow occurs when some number of bytes is copied from one buffer to another and the amount being copied is greater than is allocated for the destination buffer[5]. This leads to unspecified behaviour in C.

In the interest of ease and efficiency, C library functions such as

```
memcpy(void * restrict s1, const void * restrict s2, size_t n)
```

and

```
memmove(void *s1, const void *s2, size_t n)
```

are used to copy the contents from one area to another. `memcpy()` and `memmove()` simply copy memory and no checks are made as to whether the destination area is large enough to accommodate the n bytes of data being copied. It is assumed that the calling routine has ensured that adequate space has been provided in the destination. Problems can arise when the destination buffer is too small to receive the amount of data being copied.

A separate issue is that `memcpy` assumes that the memory blocks pointed to by `s1` and `s2` are non-overlapping. If this assumption is false, the program's behaviour is undefined. This restriction does not apply to `memmove`.

6.10.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.10.5.
- Perform range checking before calling a memory copying function such as `memcpy()` and `memmove()`. These functions do not perform bounds checking automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur
- For any functions defined with two or more restrict pointers, ensure that the arrays pointed to do not overlap
- Use the safer and more secure functions for string handling from the normative annex K of ISO/IEC 9899:2011 [2], *Bounds-checking interfaces*[6].

6.11 Pointer type conversions [HFC]

6.11.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.11 is applicable to C since C allows casting of the value of a pointer to and from another data type. These conversions can cause unexpected changes to pointer values.

---

5 This also is a special case of 6.8 Buffer boundary violation (buffer overflow) [HCB].

6 See comments on the correct use of Annex K functions in 6.8 Buffer boundary violation (buffer overflow) [HCB].

Deleted: Guidance to language users

Formatted: Normal, Space Before: 0 pt, After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

Deleted: 6.8 Buffer boundary violation (buffer overflow) [HCB]

Deleted: 6.8 Buffer boundary violation (buffer overflow) [HCB]

If a pointer is cast to a different type and then pointer arithmetic is applied (including array indexing) then the possibility exists that memory accessed is not the intended location. In particular casting from a pointer to a struct to a pointer to a basic type (like `int`) and then attempting to examine the members of the struct by incrementing the pointer might not give the expected results because of the possible presence of padding bytes.

Deleted: may not
Deleted: be
Deleted: may

The one safe pointer conversion is from a pointer to some object type to `void*` and then back to the original pointer type. The standard guarantees this to restore the original pointer.

One specific recommendation is that a macro is used to ensure that when `malloc` is used to allocate space for an object or array of a particular type, the result of `malloc` is cast to the appropriate pointer type. That is for an object of type T:

```
#define makeObjectOfTypeT(T)      (T*)malloc(sizeof(T))
```

or for an array of N elements:

```
#define makeArrayOfTypeT(T, N)    (T*)malloc(sizeof(T) * N)
```

6.11.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space Before: 0 pt, After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.11.5.
- Maintain the same type to avoid errors introduced through conversions.
- Use a macro to cast the value returned by `malloc` to the correct type.
- Heed compiler warnings that are issued for pointer conversion instances.

Deleted: Follow the advice guidance
Deleted: TR 24772
Deleted: 2019 clause

6.12 Pointer arithmetic [RVG]

6.12.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.12 is applicable to C.

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

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

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

where the address of `buf` is `0x1234`, after the assignment `buf_ptr` points to `buf[0]`. Adding 1 to `buf_ptr` will result in `buf_ptr == 0x1238` on a host where an `int` is 4 bytes; `buf_ptr` will then point to `buf[5]`. Not realizing that address operations will be in terms of the size of the object being pointed to can lead to address miscalculations and undefined behavior . .

Indexing an array is implemented by pointer arithmetic, so that accessing an array element `array[n]` creates a pointer equivalent to `array + n` and accessing the memory at that address.

6.12.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.12.5.
- Consider imposing a ban on pointer arithmetic (other than by use of the index operator) due to its error-prone nature.
- Verify that all pointers are assigned a valid memory address for use.

Deleted: Follow the advice guidance

Deleted: TR 24772

Deleted: 2019 clause

6.13 Null pointer dereference [XYH]

6.13.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.13 is applicable to C.

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

Deleted: may

Space for 10000 integers can be dynamically allocated in C in the following way[7]:

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

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

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

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

6.13.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.13.5.

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

[7] This allocation does not follow the advice in [HFC], for simplicity

- Create a specific check that a pointer is not null before dereferencing it. As this can be expensive in some cases (such as in a for loop that performs operations on each element of a large segment of memory), judicious checking of the value of the pointer at key strategic points in the code is recommended.

6.14 Dangling reference to heap [XYK]

6.14.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.14 is applicable to C.

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

Consider the following segment of code:

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

The use of memory in C after it has been freed is undefined behavior. Depending on the execution path taken in the program, freed memory can have been reallocated via another call of malloc() or other dynamic memory allocation. If the memory has not been reallocated, use of the memory might not be noticed. However, if the memory has been reallocated, altering of the data contained in the memory will almost certainly result in data corruption. Determining that a dangling memory reference is the cause of a problem and locating it can be difficult.

Deleted: may

Deleted: may

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

```
int foo() {
 int *ptr = malloc (100*sizeof(int));/* allocate space for 100 integers */
 if (ptr != NULL) {                /* check to see that the memory
                                      has been allocated */
  int ptr2 = &ptr[10];             /* set ptr2 to point to the 10th
                                      element of the allocated memory */
    …                              /* perform some operations on the memory */

    free (ptr);                    /* memory is no longer needed */
```

```
        ptr = NULL;                     /* set ptr to NULL to prevent ptr
                                           from being used again */
        …                               /* program continues performing
                                           other operations */
        ptr2[0] = 10;                   /* ERROR – memory is being used
                                           after it has been released via ptr2 */
        …
        }
      return (0);
    }
```

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

6.14.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in by ISO/IEC 24772-1:2024 6.14.2.
- Use a macro to call `free()` and to set the freed pointer to `NULL` to prevent multiple deallocation or use of a dangling reference via this pointer.
- Avoid creating additional pointers to dynamically allocated memory.

Deleted: Guidance to language users

Formatted: Normal, Space After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

6.15 Arithmetic wrap-around error [FIF]

6.15.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.15 is applicable to C.

Given the fixed size of integer data types, continuously adding to an *unsigned* integer eventually results in a value that cannot be represented. For C this is defined as a 'wrap around', so adding one to the maximum positive value results in zero. This happens without any detection or notification mechanism. Continuously adding to a *signed* integer until it reaches a value that cannot be represented results in undefined behaviour.

Similarly, repeatedly subtracting from an unsigned integer leads to wrap-around, or undefined behaviour for signed integers.

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

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

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

For unsigned integers, the wrap-around behaviour is well defined, and can be what the programmer intended. However, if the programmer is unaware that the value was getting too big to represent, they will not expect wrap-around behaviour. As it is impossible for the compiler or an analysis tool to determine what the programmer intended, it is preferable to warn if wrap-around can occur.

In C, bit shifting by a value greater than the size of the data type or by a negative number is undefined behaviour for both signed and unsigned integers. The following code, where a `int` is 16 bits, would be undefined when `j >= 16` or `j` is negative:

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

6.15.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.15.2.
- Check that the result of an operation on an unsigned integer value will not cause wrapping, unless it can be shown that wrapping cannot occur, or document and verify the intended behaviour. Any of the following operators have the potential to wrap:
  ```
  a + b    a - b    a * b    a++    ++a    a--    --a
  a += b   a -= b   a *= b   a << b a<<=b  -a
  ```
- Check that the result of an operation on a signed integer value will not cause an overflow, unless it can be shown that overflow cannot occur. Any of the following operators have the potential to overflow, which is undefined behaviour in C:
  ```
  a + b    a - b    a * b    a/b     a%b     a++     ++a     a--   --a
  a += b   a -= b   a *= b   a /= b  a %= b  a << b  a <<= b -a
  ```
- Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. These techniques can be omitted if it can be shown at compile time that overflow or underflow is not possible.
- The number of bits to be shifted by a shift operator should lie between 0 and (n-1), where n is the size of the data type.

6.16 Using shift operations for multiplication and division [PIK]

6.16.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.16 is applicable to C. Also see clause *6.15 Arithmetic Wrap-around Error [FIF].*

6.16.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 *6.16.5.*
- See also *6.15 Arithmetic Wrap-around Error [FIF]*.

Deleted: may
Deleted: may be
Deleted: and hence
Deleted: better
Deleted: Guidance to language users
Formatted: Normal, Space After: 0 pt
Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause
Deleted: The issues for C are well defined in ISO/IEC TR 24772-1:2019 *clause 6.16 Using shift operations for multiplication and division [PIK].* ...
Deleted: Guidance to language users
Formatted: Normal
Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

6.17 Choice of clear names [NAI]

6.17.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 617 is applicable to C .

The possible confusion of names with typographically similar characters is not specific to C, but C is as prone to it as any other language. Depending upon the local character set, avoid having names that only differ by characters that can be confused, such as 'O' and '0'

For C, the maximum significant name length is implementation defined. If a program includes names that are longer than the defined maximum, the compiler will truncate them to the maximum. Therefore, if two names in a program only differ in characters after the maximum, they will be treated as the same. For functions this is usually detected by the compiler as an attempted redeclaration, but for variables declared in different but overlapping scopes this can lead to the wrong variable being used, as in:

```
int long_name_ending in_A = …
   { int long_name_ending in_B = …
     /* Use of long_name_ending in_A here will actually use
        long_name_ending in_B  */
   }
```

This issue is related to *6.20 Identifier name reuse [YOW]*, as they are both mechanisms by which the programmer can inadvertently use an object other than the one intended.

6.17.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.17.2.
- Use names that are clear and non-confusing.
- Use consistency in choosing names.
- Keep names short and concise in order to make the code easier to understand.
- Do not declare names longer than the maximum defined by the implementation.
- Choose names that are rich in meaning.
- Do not uses names that only differ by a mixture of case or the presence or absence of an underscore character.
- Avoid differentiating through characters that are commonly confused visually such as 'O' and '0', 'l' (lower case 'L'), 'I' (capital 'I') and '1', 'S' and '5', 'Z' and '2', and 'n' and 'h'.

6.18 Dead store [WXQ]

6.18.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.18 is applicable to C.

Because C is an imperative language, programs in C can contain dead stores (locations that are written but never subsequently read, or overwritten without an intervening read). This can result from an error in the initial design

Deleted: may

Deleted: *6.20 Identifier name reuse [YOW]*

Formatted: Font: Italic, Underline

Deleted: may

Deleted: **Guidance to language users**

Formatted: Normal, Space After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

or implementation of a program, or from an incomplete or erroneous modification of an existing program. However, it can also be intended behaviour, for example when initializing a sparse array. It can be more efficient to clear the entire array to zero, then to assign the non-zero values, so the presence of dead stores should be regarded as a warning of a possible error, rather than an actual error.

A store into a volatile-qualified variable generally should not be considered a dead store because accessing such a variable can cause additional side effects, such as input/output (memory-mapped I/O) or observability by a debugger or another thread of execution.

6.18.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in by ISO/IEC 24772-1:2024 6.18.2.
- Use compilers and analysis tools to identify dead stores in the program.
- Mark all variables observable by another thread or hardware agent as volatile, also see *6.60.2 Avoidance mechanisms for language users*

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.59.5;

- 6.61 Concurrent data access [CGX]

Deleted: may

Deleted: Guidance to language users

Formatted: Normal

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

Formatted: Font: Italic

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

Formatted: Normal

Deleted: *6.61 Concurrent data access [CGX]*

6.19 Unused variable [YZS]

6.19.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.19 is applicable to C .

C permits variables to be declared but never used when writing code; or the code can have been modified to remove the use of some variable. Most compilers will report this as a warning and the warning can be easily resolved by removing the unused variable.

6.19.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.19.2.
- Resolve all compiler warnings for unused variables. Having an unused variable in code indicates that either warnings were turned off during compilation or were ignored by the developer.

Deleted: Variables may

Deleted: may

Deleted: Guidance to language users

Formatted: Normal, Space After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

6.20 Identifier name reuse [YOW]

6.20.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.20 is applicable to C. C allows scoping so that a variable that is not declared locally can be resolved to some outer block and that resolution can result in the variable operating on an entity other than the one intended.

Deleted: may

Deleted: may

Deleted: cause

Deleted: to

Deleted: e

In the following example, because the variable name `var1` was reused, the printed value of `var1` is possibly unexpected.

```
int var1;            /* declaration in outer scope */
var1 = 10;
{
       int var2;
       int var1;         /* declaration in nested (inner) scope */
       var2 = 5;
       var1 = 1;         /* var1 in inner scope is 1 */
}

print ("var1=%d\n", var1);  /* will print "var1=10" as var1 refers */
                            /* to var1 in the outer scope */
```

Removing the declaration of `var2` will result in a diagnostic message being generated making the programmer aware of an undeclared variable. However, removing the declaration of `var1` in the inner block will not result in a diagnostic as `var1` will be resolved to the declaration in the outer block and a programmer maintaining the code can miss this subtlety. The removing of inner block `var1` will result in the printing of `var1=1` instead of `var1=10`.

This issue is related to *6.17 Choice of clear names [NAI]*, as they are both mechanisms by which the programmer can inadvertently use an object other than the one intended.

Deleted: *6.17 Choice of clear names [NAI]*
Formatted: Font: Italic, Underline

6.20.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.20.2.
- Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and can be used in the same context. A language-specific project coding convention can be used to ensure that such errors are detectable with static analysis.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

6.21 Namespace issues [BJL]

6.21.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.21ndoes not apply to C because C requires unique names and has a single global namespace. A diagnostic message is required for duplicate names in a single compilation unit.

Deleted: Does

6.22 Missing initialization of variables [LAV]

Deleted: Initialization

6.22.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.22 is applicable to C.

Local, automatic variables can assume unexpected values if they are used before they are initialized. The C Standard specifies, "If an object that has automatic storage duration is not initialized explicitly, its value is

Deleted: "

indeterminate". In the common case, on architectures that make use of a program stack, this value defaults to whichever values are currently stored in stack memory. While uninitialized memory can contain zeros, this is not guaranteed. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and can provide an avenue for attack.

Deleted: "
Deleted: may

Many implementations will issue a diagnostic message indicating that a variable has been used that was not initialized.

6.22.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.22.2.
- Heed compiler warning messages about uninitialized variables. These warnings should be resolved as recommended to achieve a clean compile at high warning levels.
- Do not use memory allocated by functions such as `malloc()` before the memory is initialized as the memory contents are indeterminate.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

6.23 Operator precedence and associativity [JCW]

6.23.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.23 is applicable to C.

Operator precedence and associativity in C are clearly defined, and mixing logical and arithmetic operations is allowed without parentheses. However, the language has more than 40 operators with 15 levels of precedence, and experience has shown that even experienced programmers do not always get the interpretation of complex expressions correct.

6.23.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.23.5
- Use parentheses any time arithmetic operators, logical operators, and shift operators are mixed in an expression, or where the expression is complex and difficult to parse for review or maintenance.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

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

6.24.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.24 is applicable to C. C allows expressions to have side effects. If two or more side effects modify the same expression as in:

```
int v[10];
int I;
/* … */
i = v[i++];
```

Deleted: i

the behaviour is unspecified and this can lead to unexpected results. Either the "`i++`" is performed first or the assignment `i=v[i]` is performed first, or some other unspecified behaviour occurs. Because the order of evaluation can have drastic effects on the functionality of the code, this can greatly impact portability and lead to unexpected behaviour.

There are several situations in C where the order of evaluation of subexpressions or the order in which side effects take place is unspecified including:

- The order in which the arguments to a function are evaluated (C, Section 6.5.2.2″"Function call″");
- The order of evaluation of the operands in an assignment statement (C, Section 6.5.16″"Assignment operator″");
- The order in which any side effects occur among the initialization list expressions is unspecified. In particular, the evaluation order need not be the same as the order of subobject initialization (C standard [2], clause 6.7.9, "Initializatio″").

Because these are unspecified behaviours, testing can give the false impression that the code is working and portable, when it is possible that the values provided cause evaluations to be performed in a particular order that causes side effects to occur as expected.

In general, a compiler is permitted to perform calculations and assignments in any order between sequence points. Annex C of the C language standard defines all the points in the language syntax that count as sequence points. One such sequence point is the comma operator. So, whilst described above `i = v[i++];` has unspecified behaviour, as the assignment and increment can be performed in either order, `i++, i = v[i];` does not, as the increment is always performed before the assignment.

There is also a common misconception that bracketing influences the order of evaluation. This is not true. If A, B and C are functions that return integers, then in:

```
( A() + B() ) * C()
```

the brackets don't affect the order of evaluation of A, B and C, but do affect the order in which the results of these functions are combined. A, B and C can be evaluated in any order, and if they modify common variables the result is unspecified.

6.24.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.24.5
- Write expressions so that the same effects will occur under any order of evaluation that the C standard permits since side effects can be dependent on an implementation specific order of evaluation.
- Become familiar with Annex C of the C standard [2], which is a list of the sequence points that enforce an ordering of computations within an expression.

6.25 Likely incorrect expression [KOA]

Deleted: ,
Deleted: s
Deleted: ,
Deleted: s
Deleted: n
Deleted: may
Deleted: may
Deleted: be
Deleted: allowed
Deleted: may
Deleted: may
Deleted: Guidance to language users
Formatted: Normal, Space After: 0 pt
Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

6.25.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.25 is applicable to C.

C has several instances of operators which are similar in structure, but vastly different in meaning, for example confusing the comparison operator "==" with assignment "=". Using an expression that is syntactically correct, but which is a null statement, can lead to unexpected results. Consider:

```
int x, y;
/* … */
if (x = y){
  /* … */
}
```

A fair amount of analysis is often needed to determine whether the programmer intended to do an assignment as part of the if statement (valid in C) or whether the programmer made the common mistake of using an "=" instead of a "==". In order to prevent this confusion, it is suggested that any assignments in contexts that are easily misunderstood be moved outside of the Boolean expression. This would change the example code to the semantically equivalent:

```
int x, y;
/* … */
x = y;
if ( != 0) {
   /* … */
  }
```

This would clearly state what the programmer meant and that the assignment of `y` to `x` was intended.

It is also not unknown for programmers to insert the "`;`" statement terminator prematurely. However, inadvertently doing this can drastically alter the meaning of code, even though the code is valid, as in the following example:

```
int a,b;
/* … */
if (a == b);  // the semi-colon will make this a null statement
{
 /* … */
}
```

Because of the misplaced semi-colon, the code block following the if will always be executed. In this case, it is extremely likely that the programmer did not intend to put the semi-colon there.

6.25.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.25.5
- Explain statements with interspersed comments to clarify programming functionality and help future maintainers understand the intent and nuances of the code.
- Avoid assignments embedded within other statements, as these can be problematic. Each of the following would be clearer and have less potential for problems if the embedded assignments were conducted outside of the expressions:

Deleted: x

Deleted: Guidance to language users

Formatted: Normal, Space After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

```
int a,b,c,d;
/* … */
if ((a == b) || (c = (d-1)))   // the assignment to c might not
                               // occur if a is equal to b
```

or:

```
int a,b,c;
/* … */
foo (a=b, c);
```

Each is a valid C statement, but each can have unexpected results.

- Give null statements a source line of their own. This, combined with enforcement by static analysis, would make clearer the intention that the statement was meant to be a null statement.
- Consider the adoption of a coding standard that limits the use of the assignment statement within an expression.

6.26 Dead and deactivated code [XYQ]

6.26.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.26 is applicable to C. C allows the usual sources of dead code (described in 6.26 of ISO/IEC 24772-1:2019) that are common to most conventional programming languages.

C uses some operators that can be confused with other operators. For instance, the common mistake of using an assignment operator in a Boolean test as in:

```
int a;
/* … */
if (a = 1)
   { … }  else { … }
```

can cause portions of code to become dead code, because the else portion of the if statement cannot be reached.

6.26.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.26.5.

6.27 Switch statements and lack of static analysis [CLL]

6.27.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.27 is applicable to C.

Because of the way in which the switch-case statement in C is structured, it can be relatively easy to unintentionally omit the break statement between cases causing unintended execution of statements for some cases.

Deleted: may

Deleted: TR 24772

Deleted: Guidance to language users

Deleted: ¶
Follow the guidance

Formatted: Normal, No bullets or numbering, Widow/Orphan control, Don't suppress line numbers, Allow hanging punctuation

Deleted: TR 24772

Deleted: 2019 clause

C contains a switch statement of the form:

```
char abc;
/* … */
switch (abc) {
   case 1:
       sval = "a";
      break;
   case 2:
      sval = "b";
      break;
   case 3:
      sval = "c";
      break;
   default:
      printf ("Invalid selection\n");
 }
```

If there isn't a default case and the switched expression doesn't match any of the cases, then control simply shifts to the next statement after the switch statement block. Unintentionally omitting a break statement between two cases will cause subsequent cases to be executed until a break or the end of the switch block is reached. This can cause unexpected results.

Deleted: may

6.27.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the guidance contained in ISO/IEC 24772-1:2019 subclause 6.27.5.

Deleted: TR 24772

- Adopt a coding style that requires every nonempty case statement to be terminated with a `break` statement as illustrated in the following example:

```
int i;
/* … */
switch (i) {
      case 1:    /* fall through from case 1 to 2 is permitted */
      case 2:   /* since there is no intervening code */
          i++;
          break;
       case 3:
          j++;
       case 4:   /* fall through from case 3 to 4 is not permitted */
                /* as it is not a direct fall through due to the  */
                /* j++ statement */
    }
```

  If direct fall through from one nonempty case to another is required that violates this coding style, then this should be clearly documented by a comment, preferably one recognized by the analysis tool used.
- Adopt a coding style that permits your language processor and analysis tools to verify that all cases are covered. Where this is not possible, use a default clause that diagnoses the error.
- Adopt a coding style that requires the default clause to be either the first or last clause in the switch statement to assist the maintenance of complex switch statements.

6.28 Demarcation of control flow [EOJ]

6.28.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.28 is applicable to C. C lacks a keyword to be used as an explicit terminator. Therefore, it might not be readily apparent which statements are part of a loop construct or an if statement.

Deleted: may

Consider the following section of code:

```
int foo(int a, const int *b) {
   int i=0, count = 0;
   /* … */
    a = 0;
   for (i=0; i<10; i++)
       a += b[i];
       count++;
 printf("%d %d\n", a, count);
}
```

If the programmer intended both `a += b[i];` and `count++;` to be the body of the loop, the second statement will erroneously be performed only once, since there are no enclosing brackets.

If statements in C are also susceptible to control flow problems since there isn't a requirement for there to be an `else` statement for every `if` statement. An `else` statement in C always belong to the most recent `if` statement without an `else`. However, the situation can occur where it is not readily apparent to which `if` statement an `else` belongs due to the way the code is indented or aligned.

6.28.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.28.5.
- Enclose the bodies of `if`, `else`, `while`, `for`, and similar constructs in braces. This will reduce confusion and potential problems when modifying the software. For example:

```
int a,b,i;
/* … */
if (i == 10){
      a = 5;     /* this is correct */
      b = 10;
     }
 else
      a = 10;
      b = 5;
```

  If the assignments to `b` were added later and were expected to be part of each `if` and `else` clause (they are indented as such), the above code is incorrect: the assignment to `b` that was intended to be in the `else` clause is unconditionally executed.

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

6.29 Loop control variable abuse [TEX]

Deleted: s

6.29.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.29 is applicable to C as the modification of loop control variables within the loop is permitted, which can cause unexpected behaviour.

Deleted: C allows

Deleted: but

Since the modification of a loop control variable within a loop is infrequently encountered, reviewers of C code might not expect it and hence miss noticing the modification or not recognize its significance. Modifying the loop control variable can cause unexpected results, as in:

```
int a,i;
for (i=1; i<10; i++){
   …
   if (a > 7)
      i = 10;
   …
}
```

Deleted: may

which would cause the for loop to exit once `a` is greater than 7 regardless of the number of iterations that have occurred.

C doesn't require the loop control variable to be an integer type. If, for example, it is a floating point type, the test for completion should not use equality or inequality, as floating point rounding can lead to mathematically inexact results, and hence an unterminated loop. The following code either loops ten times or indefinitely:

Deleted: may

```
float j;
for (j = 0.0f; j != 10.0f; j += 1.0f){
   …
}
```

The following is an improvement:

```
float j;
for (j = 0.0f; j < 10.0f; j += 1.0f){
   …
}
```

Rounding will cause this loop to be performed either ten or eleven times. To ensure this loop is performed ten times, `j` needs to be initialized to `0.5f`.

Deleted: may

6.29.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms of ISO/IEC 24772-1:2024 6.29.5.
- Do not modify a loop control variable within a `for` loop.
- Do not use floating point types as a loop control variable.

Deleted: Follow the guidance

Deleted:

Deleted: TR 24772

Deleted: 2019 clause

6.30 Off-by-one error [XZH]

6.30.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.30 is applicable to C.

Arrays are a common place for off-by-one errors to manifest. In C, arrays are indexed starting at 0, causing the common mistake of looping from 0 to the size of the array as in:

```
int foo() {
int a[10];
int i;
for (i=0, i<=10, i++)
  …
return (0);
}
```

Strings in C are also another common source of errors in C due to the need to allocate space for and account for the string terminator. A common mistake is to expect to store an n length string in an `n` length array instead of length `n+1` to account for the terminating `'\0'` character. Interfacing with other languages that do not use terminators in strings can also lead to an off by one error.

C does not flag accesses outside of array bounds, so an off by one error might not be detectable. Several tools can be used to help detect accesses beyond the bounds of arrays. However, such tools will not help in the case where only a portion of the array is used and the access is still within the bounds of the array.

Looping one more or one less is usually detectable by good testing. Due to the structure of the C language, this can be the main way to avoid this vulnerability. Unfortunately, some cases can still slip through the development and test phase and manifest themselves during operational use.

6.30.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.30.5.
- Use careful programming, testing of boundary conditions, and static analysis tools to detect off by one errors in C.

6.31 Unstructured programming [EWD]

6.31.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.31 is applicable to C.

It is as easy to write structured programs in C as it is to write unstructured code. C contains the `goto` and `longjmp` statements, which can create unstructured code. C also has `continue`, `break`, and `return` that can create complicated control flow when used in an undisciplined manner. Unstructured {spaghetti} code can be more difficult for C static analyzers to analyze and is sometimes used on purpose to obfuscate the functionality of

Deleted:
Deleted:
Deleted: may
Deleted: Guidance to language users
Formatted: Normal, Space After: 0 pt
Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause
Deleted: not to

software. Code that has been modified multiple times by an assortment of programmers to add or remove functionality or to fix problems can be prone to become unstructured.

Because unstructured code in C can cause problems for analyzers (both automated and human), problems with the code might not be detected as readily or at all as would be the case if the software was written in a structured manner.

IEC 61508 [[5] highly recommends the use of no more than one `return` statement in a function. At times, this guidance can have the opposite effect, such as in the case of an `if` check of parameters at the start of a function that requires the remainder of the function to be encased in the `if` statement in order to reach the single exit point. If, for example, the use of multiple exit points can arguably make a piece of code clearer, then they should be used. However, the code should be able to withstand a critique that a restructuring of the code would have made the need for multiple exit points unnecessary.

6.31.2 Avoidance mechanisms for language users

Deleted: may

Deleted: Guidance to language users

Formatted: Normal, Space After: 0 pt

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.31.5.
- Write clear and concise structured code to make code as understandable as possible.
- Restrict the use of `goto`, `continue`, `break` and `longjmp` to encourage more structured programming.

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

6.32 Passing parameters and return values [CSJ]

6.32.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.32 is applicable to C.

C uses *call by value* parameter passing. The parameter is evaluated and its value is assigned to the formal parameter of the function that is being called. A formal parameter behaves like a local variable and can be modified in the function without affecting the actual argument. An object can be modified in a function by passing the address to the object to the function, for example

```
void swap(int *x, int *y) {
   int t = *x;
   *x = *y;
   *y = t;
}
```

Where `x` and `y` are integer pointer formal parameters, and `*x` and `*y` in the `swap()` function body dereference the pointers to access the integers. If it is not intended that the function should be able to modify the object whose address is passed to the function, the object of the pointer should be specified as constant, e.g. `const int *p`

C99 introduced the `restrict` keyword. This can be applied to function pointer parameters. Where a function has two or more pointer parameters marked with `restrict`, the programmer is telling the compiler that the function will never be called with arrays that have overlapping access. This allows the compiler to make use of optimizations that can lead to incorrect results if the arrays do overlap, e.g. a copy function like `strncpy` that

Deleted: may

copies a fixed number of characters from a source string to a target. If the target overlaps the source, the result depends upon whether the copying was performed from the start of the string to the end or vice versa. Conversely, where a library function is declared with `restrict` parameters, the programmer is being told never to call it so that accesses within the function overlap. There is no compile or run-time check that the parameter arrays are actually non-overlapping, so caution should be taken when using functions with restrict parameters.

Whilst function-like macros appear to be called, they are actually ‘executed’ by text substitution before compilation, so parameter passing doesn’t occur, see *6.51 Pre-processor directives [NMP]*.

Deleted: *6.51 Pre-processor directives [NMP]*
Formatted: Font: Italic

6.32.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.32.5.
- Do not use expressions with side effects in parameters to function-like macros, unless it can be shown that the parameter is used only once inside the macro.
- Do not use expressions with side effects for multiple parameters to functions, since the order in which the parameters are evaluated and hence the side effects occur is unspecified.
- Use caution when passing the address of an object. The object passed could be an alias[8]. Aliases can be avoided by following the respective guidelines of ISO/IEC 24772-1:2019 subclause 6.32.5.
- Do not use a function that includes the `restrict` keyword unless it can be established that the array parameters to the function can never overlap.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause
Deleted: may
Deleted: TR 24772

6.33 Dangling references to stack frames [DCM]

6.33.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.33 is applicable to C.

C allows the address of a variable to be stored in a pointer variable. Should this pointer variable contain, for example, the address of a local variable that was part of a stack frame, then using this address after the function containing the local variable has terminated leads to undefined behaviour, as the memory will have been made available for further allocation and can indeed have been allocated for some other use. The same is true for a pointer to memory allocate with `malloc` etc. and which has subsequently been freed.

Deleted: may

6.33.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.33.5.
- Do not assign the address of an object to any entity which persists after the object has ceased to exist.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

---

[8] An alias is a variable or formal parameter that refers to the same location as another variable or formal parameter.

This is done in order to avoid the possibility of a dangling reference. In particular, never return the address of a local variable as the result of a function call.

- Long lived pointers that contain block-local addresses should be assigned the null pointer value before executing a return from the block.

6.34 Subprogram signature mismatch [OTR]

6.34.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.34 is applicable to C.

In C, If it is necessary to call a function that is not yet defined in the current translation unit[9], a function prototype is required as a forward reference to the definition. Usually the prototype specifies the name of the function, its return type and the types of the parameters it requires, as in

```
void foo(int x);
```

However for compatibility with earlier C standards, compilers accept a prototype with either no parameters, as in

```
void foo();
```

or just parameter names, as in

```
void foo(x);
```

If either of these two forms is used, the compiler allows calls to the function with any number of parameters, including none, which can lead to undefined behaviour. It is therefore recommended that function prototypes should always be written with parameter types. If the function has no parameters, it should be written using `void` as the parameter list, as in

```
void goo(void);
```

C also allows a function to take a variable number of arguments, as in the `printf()` function. This is specified in the function definition by terminating the list of parameters with an ellipsis `(...)`. No information about the number or types of the parameters expected is supplied, and the compiler will accept any number and type of parameters in the call.

C compilers will attempt to perform an implicit conversion from the type of an actual parameter to the type of the formal parameter. So for `sqrt()` that is defined to expect a double:

```
double sqrt(double)
```

the call:

```
root2 = sqrt(2);
```

converts the integer `2` into the double value `2.0`.

6.34.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.34.5.

[9] For example because the function is defined in a different translation unit, or there is mutual recursion between two (or more) functions.

Deleted: may

Deleted: Guidance to language users

Formatted: Normal, Space After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

- Use a function prototype to declare a function with its expected parameters to allow the compiler to check for a matching count and types of the parameters. If the function has no parameters, show its parameter list as `(void)` rather than `()`.
- Do not use the variable argument feature except in rare instances. The variable argument feature such as is used in `printf()` is difficult to use in a type safe manner.

6.35 Recursion [GDL]

6.35.1 Applicability to language

C permits recursion, hence is subject to the issues documented in ISO/IEC 24772-1:2019 subclause 6.35.

6.35.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.35.5.

6.36 Ignored error status and unhandled exceptions [OYB]

6.36.1 Applicability to language

The vulnerability related to error status as documented in ISO/IEC 24772-1:2024 6.36 is applicable to C. The C standard does not include exception handling, therefore only error status will be covered.

C provides the header file `<errno.h>` that defines the macros `EDOM`, `EILSEQ` and `ERANGE`, which expand to integer constant expressions with type `int`, distinct positive values and which are suitable for use in `#if` preprocessing directives. C also provides the integer `errno` that can be set to a nonzero value by any library function to indicate that an error has occurred (if the use of `errno` is not documented in the description of the library function in the C Standard, `errno` can be used whether or not there is an error). Though these values are defined, inconsistencies in responding to error conditions can lead to vulnerabilities.

`errno` and the defined macros can also be used by user defined functions, but for clarity, such use should be consistent with the use by library functions.

C library functions can also return error indicator values.

6.36.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.36.5.
- Check the returned error status upon return from a function.
- Set `errno` to zero before a library function call in situations where a program intends to check `errno` before a subsequent library function call.

Deleted: problems
Deleted: described
Deleted: TR 24772
Deleted: Guidance to language users
Deleted: ¶
A
Deleted: guidance
Formatted: Space After: 10 pt
Deleted: TR 24772
Deleted: 2019 clause
Deleted: may
Deleted: may
Deleted: may
Deleted:
Deleted: may
Deleted: Guidance to language users
Formatted: Normal, Space After: 0 pt
Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

- Use errno_t to make it readily apparent that a function is returning an error code. Often a function that returns an errno error code is declared as returning a value of type int. Although syntactically correct, it is not apparent that the return code is an errno error code. The normative Annex K from ISO/IEC 9899:2011 [2] introduces the new type errno_t in <errno.h> that is defined to be type int.
- Handle an error as close as possible to the origin of the error but as far out as necessary to be able to deal with the error.
- When a function returns an error value, other than using errno (e.g. malloc that returns NULL if the requested memory allocation cannot be performed), always check the error condition returned after a call
- 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 encounters an error, finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.

6.37 Type-breaking reinterpretation of data [AMV]

6.37.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.37 is applicable to C.

The primary way in C that a reinterpretation of data can be accomplished is through a union, which can be used to interpret the same piece of memory in multiple ways. If the use of the union members is not managed carefully, then unexpected and erroneous results can occur.

Reinterpretations can also result from a pointer accessing data of a type other than the type of the pointer. This occurs if the pointer has been cast, and can result in undefined behaviour, as documented in ISO/IEC 24772-1:2024 6.37. The only pointer casting that the C standard requires to provide defined results is to cast the pointer to void *, then cast the void * pointer back to the same type. So if S and T are distinct types:

```
S x;  void *pX = &x;  S *pX2 = (S*)pX;  // defined behaviour
                      T *pX3 = (T*)pX;  // undefined behaviour
```

6.37.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.37.5.
- When using unions, implement an explicit discriminant and check its value before accessing the data in the union.
- Ensure through the use of static analysis tools that arbitrary pointer casts always return objects of the correct type.

Deleted: may

Deleted: TR 24772

Deleted: 2019 clause

Deleted: Guidance to language users

Formatted: Normal, Space After: 0 pt

Deleted: Follow the guidance

Deleted:

Deleted: TR 24772

Deleted: 2019 clause

6.38 Deep vs. shallow copying [YAN]

6.38.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.38 is applicable to C.

This issue can arise where a struct or union contains a pointer to an object. If `A` and `B` are two struct objects of the same type that has a pointer member, then the statement `A = B;` copies all the members of `B` to the equivalent members of `A`. For the pointer, only the pointer itself has been copied, so `A` and `B` both now point to the same object, i.e. shallow copying.

If the required behaviour is to copy the struct and have each copy point to its own object, then a function is needed to implement deep copying, i.e. copy all the members of `B` to `A` – other than the pointer, and allocate sufficient memory to make a copy of the object pointed to by `B` and make `A` point to this new object.

6.38.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.38.5.
- Where necessary, create a function to correctly perform the deep copy.

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

6.39 Memory leaks and heap fragmentation [XYL]

6.39.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.39 is applicable to C.

C relies on the programmer to implement memory management, allocating and freeing dynamic memory as required, rather than supplying a built-in garbage collector.

Deleted: built in

Memory is dynamically allocated in C using the library calls `malloc()`, `calloc()`, and `realloc()`. When the program no longer needs the dynamically allocated memory, it can be released using the library call `free()`. Should there be a flaw in the logic of the program, memory can continue to be allocated but not freed when it is no longer needed. A common situation is where memory is allocated while in a function, the memory is not freed before the exit from the function and the lifetime of the pointer to the memory has ended upon exit from the function.

Deleted: may

6.39.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Follow the guidelines of ISO/IEC 24772-1:2024 6.39.5.
- Use debugging tools such as leak detectors to help identify unreachable memory.

Deleted: TR 24772

Deleted: 2019 clause

- Allocate and free memory in the same module and at the same level of abstraction to make it easier to determine when and if an allocated block of memory has been freed.
- Use `realloc()` only to resize dynamically allocated arrays.

6.40 Templates and generics [SYM]

The vulnerability documented in ISO/IEC 24772-1 6.40 does not apply to C, because C does not implement template or generic mechanisms.

6.41 Inheritance [RIP]

The vulnerability documented in ISO/IEC 24772-1 6.41 does not apply to C, as C does not implement struct hierarchies.

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

The vulnerability documented in ISO/IEC 24772-1 6.42 does not apply to C as C does not implement polymorphism.

6.43 Redispatching [PPH]

The vulnerability documented in ISO/IEC 24772-1 6.43 does not apply to C as C does not implement object-oriented programming, which includes redispatching.

6.44 Polymorphic variables [BKK]

The vulnerability documented in ISO/IEC 24772-1 6.44 does not apply to C as C does not implement object-oriented programming and polymorphic variables.

6.45 Extra intrinsics [LRM]

The vulnerability documented in ISO/IEC 24772-1 6.45 does not apply to C since C does not implement these mechanisms.

6.46 Argument passing to library functions [TRJ]

6.46.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.46 is applicable to C.

There isn't a guarantee that the parameters being passed to a function will be verified by either the calling or receiving functions. Therefore, values outside of the assumed range can be received by a function resulting in a

Deleted: is
Deleted: these
Deleted: This vulnerability does not apply to C, because
Deleted: This vulnerability does not apply to C,
Deleted: because
Deleted: This vulnerability does not apply to C, because
Deleted: this mechanism
Deleted: This vulnerability does not apply to C, because
Deleted: this mechanism
Deleted: This vulnerability does not apply to C, because
Deleted: So
Deleted: may

potential vulnerability. This is particularly troublesome if the parameter is a pointer with an unexpected value `NULL`.

If a parameter is received by a function that was assumed to be within a particular range and then an operation or series of operations is performed using the value of the parameter, this can result in unanticipated results and even a potential vulnerability.

Deleted: A
Deleted: may be
Deleted: ing

6.46.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.46.5.
- Do not make assumptions about the values of parameters.
- Do not assume that the calling or receiving function will be range checking a parameter. Therefore, establish a strategy for each interface to check parameters in either the calling or receiving routines.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

6.47 Inter-language calling [DJS]

6.47.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.47 is applicable to C.

The C Standard defines the calling conventions, data layout, error handing and return conventions needed to use C from another language. Ada has developed a standard for interfacing with C. Fortran has included a Clause 15 that explains how to call C functions. Calls from C into other languages become the responsibility of the programmer.

6.47.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.47.5.
- Minimize the use of those issues known to be error-prone when interfacing from C, such as:
  1. passing character strings;
  2. dimension, bounds and layout issues of arrays;
  3. interfacing with other parameter formats such as call by reference or name;
  4. receiving return codes; and
  5. bit representation.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

6.48 Dynamically-linked code and self-modifying code [NYY]

6.48.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.48 is applicable to C.

Most loaders allow dynamically linked libraries also known as shared libraries. Code is designed and tested using a suite of shared libraries which are loaded at execution time. The process of linking and loading is outside the scope of the C standard.

C permits self-modifying code. In C there isn't a distinction between data space and code space, executable commands can be altered as desired during the execution of the program. Although self-modifying code is easy to write in C, it can be difficult to understand, test and fix leading to potential vulnerabilities in the code.

Self-modifying code can be done intentionally in C to obfuscate the effect of a program or in some special situations to increase performance. Modification of C code can occur if pointers are misdirected to access the code space instead of data space or code is executed in data space. Accidental modification usually leads to a program crash. Intentional modification can also lead to a program crash, but used in conjunction with other vulnerabilities can lead to more serious problems that affect the entire host.

6.48.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.48.5.
- Do not use self-modifying code, unless it has a documented rationale and is carefully reviewed.
- Verify that the dynamically linked or shared code being used is the same as that which was tested.
- Retest when it is possible that the dynamically linked or shared code has changed before using the application.

6.49 Library signature [NSQ]

6.49.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.49 is applicable to C since C programs are commonly used to provide libraries to other language systems, and programs written in C can often call into library systems written in other languages.

Integrating C and another language into a single executable relies on knowledge of how to interface the function calls, argument lists and data structures so that symbols match in the object code during linking. Byte alignments can be a source of data corruption.

For instance, when calling Fortran from C, several issues arise:

- Neither C nor Fortran check for mismatch argument types or even the number of arguments.
- C passes arguments by value and Fortran passes arguments by reference, so addresses must be passed to Fortran rather than values in the argument list.
- Multidimensional arrays in C are stored in row major order, whereas Fortran stores them in column major order.
- Strings in C are terminated by a null character, whereas Fortran uses the declared length of a string.

These are just some of the issues that arise when calling Fortran programs from C. Each language has its differences with C, so different issues arise with each interface.

Deleted: Guidance to language users

Formatted: Normal, Space After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

Writing a library wrapper is the traditional way of interfacing with code from another language. However, this can be quite tedious and error-prone.

6.49.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.49.5.
- Use a tool, if possible, to automatically create interface wrappers.

Deleted: Guidance to language users

Formatted: Normal, Space After: 0 pt

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

6.50 Unanticipated exceptions from library routines [HJW]

Since C does not have exceptions and so cannot handle exceptions passed from other language systems, this vulnerability as documented in ISO/IEC 24772-1 6.50 does not apply to C. See 6.36 for a discussion of Ignored errors. See ISO/IEC 24772-1:2024 6.47 for the case where libraries written in languages that use exceptions can be called.

Deleted: TR 24772

Deleted: 2019 clause

Deleted: in

Deleted: may be

6.51 Pre-processor directives [NMP]

6.51.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.51 is applicable to C. The C pre-processor allows the use of macros that are text-replaced before compilation.

Function-like macros look similar to functions but have different semantics. Because the arguments are text-replaced, expressions passed to a function-like macro can be evaluated multiple times. This can result in unintended and unspecified behaviour , if the arguments have side effects or are pre-processor directives as described by ISO/IEC 9899:2011 clause 6.10 [2]. Additionally, the arguments and body of function-like macros should be fully parenthesized to avoid unintended and unspecified behaviour [2].

Deleted: may

The following code example demonstrates unspecified behaviour when a function-like macro is called with arguments that have side-effects (in this case, the increment operator) [2]:

```
#define foo(X) ((X) * (X) + (X))
/* ... */
int i = 2;
int a = foo(++i);
```

The above example expands to:

```
int a = ((++i) * (++i) + (++i));
```

this has unspecified behaviour, as its not known in which order the compiler will evaluate the three `++i` subexpressions.

Another mechanism of failure can occur when the arguments within the body of a function-like macro are not fully parenthesized. The following example shows a macro without parenthesized arguments [2]:

```
#define CUBE(X) (X * X * X)
/* ... */
int a = CUBE(2 + 1);
```

This example expands to:

```
int a = (2 + 1 * 2 + 1 * 2 + 1)
```

which evaluates to 7 instead of the intended 27.

6.51.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space Before: 0 pt, After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.51.5.
- Replace function-like macros with inline functions where possible. Inline functions offer consistent semantics and allow for better analysis by static analysis tools.
- Ensure that if a function-like macro must be used, that its arguments and body are parenthesized, and use a naming convention that clearly identifies it as a macro.
- Do not use pre-processor directives or expressions with side-effects (such as assignment, increment/decrement, volatile access, or function calls) in the parameter of a function-like macro.

Deleted: Follow the guidance

Deleted: TR 24772

Deleted: 2019 clause

6.52 Suppression of language-defined run-time checking [MXB]

The vulnerability documented in ISO/IEC 24772-1:2024 6.52 does not apply to C since C does not provide language-defined runtime checks.

Deleted: Does not apply to

Deleted: since there are no

6.53 Provision of inherently unsafe operations [SKL]

6.53.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.53 is applicable to C since C was designed for implementing system software where some ‘unsafe’ operations are inherent and common.

6.53.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.53.5.

Deleted: ¶
Follow the guidance

Formatted: Space After: 10 pt, Widow/Orphan control, Don't suppress line numbers, Allow hanging punctuation

Deleted:

Deleted: TR 24772

Deleted: 019

Deleted: clause

6.54 Obscure language features [BRS]

6.54.1 Applicability of language

The vulnerability documented in ISO/IEC 24772-1:2024 6.54 is applicable to C. C is a relatively small language with a limited syntax set, lacking many of the complex features of some other languages. Many of the complex features in C are not implemented as part of the language syntax, but rather implemented as library routines. As such, most of the available features in C are used relatively frequently.

Problems are more likely to arise from the use of a combination of features that are rarely used together or fraught with issues if not used correctly. This can cause unexpected results and potential vulnerabilities.

6.54.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.54.5.

6.55 Unspecified behaviour [BQF]

6.55.1 Applicability of language

The vulnerability documented in ISO/IEC 24772-1:2024 6.55 is applicable to C. The C standard [2] has documented in Annex J.1, 54 instances of unspecified behaviour. Examples of unspecified behaviour include:

- The order in which parameters of a function call are evaluated
- The order in which any side effects occur among the initialization list expressions in an initializer
- The layout of storage for function parameters

Reliance on a particular observed behaviour that is unspecified not only leads to portability problems when the same code is compiled with a different compiler but is not required to be consistent within the same program. Many cases of unspecified behaviour have to do with the order of evaluation of subexpressions and side effects. For example, in the function call

```
f1(f2(x), f3(x));
```

the functions `f2` and `f3` can be called in any order, possibly yielding different results.

6.55.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.55.5.
- Avoid reliance on unspecified behaviour because the behaviour can change at each instance. Any code that makes assumptions about the behaviour of something that is unspecified should be replaced.

6.56 Undefined behaviour [EWF]

6.56.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.56 is applicable to C. The C standard does not impose any requirements on code with undefined behaviour. Typical undefined behaviours include doing nothing, producing arbitrary results, and terminating the program.

The C standard has documented, in Annex J.2, 191 instances of undefined behaviour that exist in C. One example of undefined behaviour occurs when the value of the second operand of the `/` or `%` operator is zero. This is generally not detectable through static analysis of the code, but can easily be prevented by a check for a zero divisor before the operation is performed. Leaving this behaviour as undefined lessens the burden on the implementation of the division and modulo operators.

Deleted: 6.54.2 Guidance to language users
Formatted: Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 2 + Alignment: Left + Aligned at: 0 cm + Indent at: 1.31 cm
Deleted: ¶
Follow the guidance
Formatted: Font: (Default) Calibri
Formatted: Normal, No bullets or numbering, Widow/Orphan control, Don't suppress line numbers, Allow hanging punctuation
Deleted: TR 24772
Deleted: 2019 clause
Formatted: Font: (Default) Calibri
Formatted: Font: (Default) Calibri
Deleted: compiler, but
Deleted: may
Deleted: Guidance t
Deleted: o
Formatted: Normal, Space Before: 0 pt, After: 0 pt
Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause
Deleted: Do not rely

Other examples of undefined behaviour include:

- Referring to an object outside of its lifetime
- The conversion to or from an integer type that produces a value outside of the range that can be represented
- The use of two identifiers that differ only in non-significant characters

Relying on undefined behaviour makes a program unstable and non-portable. Whilst it is possible that the code generated by a particular compiler shows consistent behaviour in cases that the standard specifies as "undefined", it is still dangerous to rely on this behaviour.

6.56.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.56.5.

6.57 Implementation–defined behaviour [FAB]

6.57.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.57 is applicable to C. The C standard has documented, in Annex J.3, 112 instances of implementation-defined behaviour. Examples of implementation-defined behaviour include:

- The number of bits in a byte
- The direction of rounding when a floating-point number is converted to a narrower floating-point number
- The rules for composing valid file names

Relying on implementation-defined behaviour can make a program less portable across implementations. However, this is less true than for unspecified behaviour (6.55) and undefined behaviour (6.56). Also, as many basic properties, such as the sizes of the basic types, are implementation defined, it is virtually impossible to avoid using implementation defined features. The header `stdint.h` provides the definition of fixed width integers, so `uint32_t` is an unsigned 32-bit integer. Which sizes are supported is implementation defined, but some safety-critical coding standards recommend the use of these types in preference to the basic types where available.

The following code shows an example of reliance upon implementation-defined behaviour:

```
unsigned char x = 100;
x = 5*x +1;
```

Since the width of `unsigned char` is implementation-defined, the computation on `x` will yield different results for implementations with different widths.

6.57.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Deleted: Guidance to language users

Deleted: ¶
Follow the guidance

Formatted: Space After: 10 pt, Widow/Orphan control, Don't suppress line numbers, Allow hanging punctuation

Deleted:

Deleted: TR 24772

Deleted: 2019 clause

Deleted: Guidance to language users

Formatted: Normal, Space Before: 0 pt, After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.57.5.
- Eliminate to the extent possible any reliance on implementation-defined behaviour from programs in order to increase portability. Even programs that are specifically intended for a particular implementation might in the future be ported to another environment or sections reused for future implementations.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause
Deleted: may

6.58 Deprecated language features [MEM]

6.58.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.58 is applicable to C.

C deprecated one function, the function `gets()` and removed it from the standard in 2011.

C has deprecated several language features primarily by tightening the requirements for the feature:

- Implicit `int` declarations are no longer allowed.
- Functions cannot be implicitly declared. They must be defined before use or have a prototype.
- The use of the function `ungetc()` at the beginning of a binary file is deprecated.
- A `return` without expression is not permitted in a function that returns a value (and vice versa).

6.58.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

Formatted: Normal, Space Before: 0 pt, After: 0 pt

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.58.5.
- Rewrite code that uses deprecated language features to remove such use, whenever possible.

Deleted: Follow the guidance
Deleted: TR 24772
Deleted: 2019 clause

6.59 Concurrency – Activation [CGA]

6.59.1 Applicability to language

The C standard, in clause 7.26.5.1, requires a conforming implementation to set specific return codes to indicate whether or not a thread activation succeeded; therefore the vulnerability as specified in ISO/IEC 24772-1 6.59 does not apply to the C language.

However, if the program fails to check the return code and fails to take appropriate action to handle a failed thread creation, the vulnerability described in clause 6.36 Ignored error status and unhandled exceptions [OYB] applies.

Deleted: of this document

6.59.2 Avoidance mechanisms for language users

Deleted: Guidance to language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.59.5.
- Always check return codes after thread activation.

Deleted: <#>¶
Follow the guidance
Deleted: <#>TR 24772
Deleted: <#>2019 clause
Deleted: <#>.
Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.71 cm + Indent at: 1.35 cm, Widow/Orphan control, Don't suppress line numbers, Allow hanging punctuation

6.60 Concurrency – Directed termination [CGT]

6.60.1 Applicability to language

This vulnerability as documented in ISO/IEC 24772-1 6.60 does not apply to C as far as language-provided mechanisms to terminate another thread since C does not implement a such mechanisms and relies on thread-to-thread communication or the setting of global flags to instruct another thread to terminate itself. In this case the termination-directing thread and the terminating thread are both responsible to ensure that that such termination doesn't occur until all critical activities are completed.

6.60.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.59.5;

6.61 Concurrent data access [CGX]

6.61.1 Applicability to language

As stated in clause 5.1.2.4 of the C standard, a program that contains a data race exhibits undefined behaviour. In addition to threads, signal handlers also pose a risk of concurrent data access. It is the responsibility of the application to use atomic variables or mutexes to ensure that one thread or signal handler cannot modify an object while another thread or signal handler is attempting to access the same object. For signal handling, "`volatile sig_atomic_t`" or atomic variables can be used to prevent this vulnerability.

6.61.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.61.5.
- Use atomic variables where appropriate to avoid data races.
- Use mutexes appropriately to protect accesses to non-atomic shared objects. Where mutexes are used, the programmer must show that there are no paths in the program where a release can be missed, either because of conditional code or other mechanisms.
- Use mutexes to model Hoare monitors or similar high level abstractions of synchronization.
- Use "`volatile sig_atomic_t`" to protect data shared with signal handlers in a single-threaded environment.

6.62 Concurrency – Premature termination [CGS]

6.62.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.62 is applicable to C since the C standard does not provide a mechanism to determine whether a thread has terminated.

Deleted: because
Deleted: to directly terminate a thread
Deleted: A similar effect can be achieved by a global flag requesting that a thread terminate itself,
Deleted: but
Deleted: is
Formatted: Normal
Deleted: 6.61
Formatted: Font: Italic
Formatted: Heading 3
Deleted: ¶
6.61.1 Applicability to language ¶
Deleted: Guidance to language users
Formatted: Normal
Deleted: Follow the guidance
Deleted: contained
Deleted: TR 24772
Deleted: 2019 clause
Deleted: This vulnerability applies to C because

6.62.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.62.5.
- Use low-level operating system primitives or other APIs where available to check that a required thread is still active.

6.63 Lock protocol errors [CGM]

6.63.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.63 is applicable to C. Applications in C can contain lock protocol errors such as a missing release of a mutex. See ISO/IEC 24772-1:2024 6.63 for descriptions and mitigations of protocol lock errors.

6.63.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.63.5.
- Be aware of the operation of each synchronization mechanism, such as the cases where accesses to atomic variables can occur more than once in a statement.

6.64 Reliance on external format strings [SHL]

6.64.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.64 is applicable to C. The standard C libraries provide a large family of input and output functions that use a control string to interpret the data read or format the output. These strings include all the feature described in ISO/IEC 24772-1:2024 6.64.1.

6.64.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.64.5.

6.645 Modifying constants [UJO]

6.65.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2024 6.65 exists in C.

C provides two ways to declare a constant:

- Using the preprocessor to set a name to a value, in which the preprocessor will replace the name everywhere in the program with the explicit value for the unit being compiled.
- Declaring a variable const as in

```
int const my_age = 42;
```

In the first case, the name is completely replaced with the numeric value, and any attempts to modify it will result in a compilation error, such as

```
my_age = my_age + 1;
```

would result in the underlying C code

```
42 = 42 + 1;
```

And produce compilation error. Similarly,na later code section could attempt to redefine the constant, which will result in a compiler warning, but not an error and the new value will be used in subsequent code.

In the second case, a direct assignment to a variable defined as `const` will result in a compiler error, but other paths can be used to access and modify the supposedly constant variable. For example

```
int const my_age = 42;
int *variable_age = &my_age;

*variable_age = 75;  //will also set my_age to 75
```

It Is possible to define a constant pointer to a constant object, as in

```
const int * const some_age( &my_age  );
```

Non it will be impossible to update either `my_age` directly or via `some_age`, but it is still possible to declare a non-constant pointer, assign it to the value of the constant pointer, and then update the no-longer-constant value, as in

```
int const my_age = 42;
const int * const some_age( &my_age  );

int *variable_age = some_age;

*variable_age = 75; // sets my_age to 75.
```

- **Formatted:** Font: Cambria
- **Formatted:** Font: Cambria
- **Formatted:** Font: Cambria
- **Formatted:** Font: Cambria
- **Formatted:** Font: 10.5 pt
- **Formatted:** Font: (Default) Courier New, 10.5 pt
- **Formatted:** Font: (Default) Courier New, 10.5 pt
- **Formatted:** Font: (Default) Courier New, 10.5 pt, Font color: Custom Color(RGB(31,35,40))
- **Formatted:** List Paragraph, Indent: Left: 2.13 cm
- **Formatted:** Font: Cambria, 12 pt
- **Formatted:** Font: (Default) Courier New, 10.5 pt, Font color: Custom Color(RGB(31,35,40))
- **Formatted:** Normal, Indent: Left: 0 cm
- **Formatted:** Font: (Default) Courier New, 10.5 pt, Font color: Custom Color(RGB(31,35,40))
- **Formatted:** Font: (Default) Courier New, 10.5 pt, Font color: Custom Color(RGB(31,35,40))
- **Formatted:** Normal, Indent: Left: 1.42 cm, First line: 0.71 cm
- **Formatted:** Font: Courier New, 10.5 pt
- **Formatted:** List Paragraph, Indent: Left: 2.13 cm

Formatted: Normal

6.65.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects C software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.65.5.

Formatted: Font: (Default) Calibri

Formatted: Font: (Default) Calibri

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.71 cm + Indent at: 1.34 cm

Bibliography

[1] Kernighan, Ritchie, *The C Programming Language (1st Edition)*, Prentice Hall 1978

[2] ISO/IEC 9899:2011, *Information technology — Programming languages* — C

[3] 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*

[4] ISO/IEC 9945:2009 -- *Information Technology -- Portable Operating System Interface(POSIX) with TC 1:2013*

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

[6] ISO/IEC 9899:2011/Cor.1:2012, *Technical Corrigendum 1*

[7] Hatton, L., *Safer C: developing software for high-integrity and safety-critical systems*. McGraw-Hill 1995

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

[9] ISO/IEC TS 17961, *Information technology – Programming languages, their environments and system software interfaces – C secure coding rules*

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

[11] MISRA, Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2012 (third edition).

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

[13] RTCA, EUROCAE, RTCA DO-178B/EUROCAE ED-12B, *Software Considerations in Airborne Systems and Equipment Certification*. December 1992.

[14] Seacord, R. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005. See http://www.cert.org/books/secure-coding for news and errata.

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

Index