
**Programming languages — Guidance
to avoiding vulnerabilities in
programming languages —**

**Part 3:
C**

*Langages de programmation — Conduite pour éviter les
vulnérabilités dans les langages de programmation —*

Partie 3: C

iTeh STANDARD PREVIEW
(standards.iteh.ai)
Full standard available on iTeh at <https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020>

PROOF / ÉPREUVE



iTeh STANDARD PREVIEW
(standards.iteh.ai)
Full standard:
<https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020>



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2020

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

	Page
Foreword	vii
Introduction	viii
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
4 Language concepts	2
5 Avoiding programming language vulnerabilities in C	2
6 Specific guidance for C vulnerabilities	3
6.1 General.....	3
6.2 Type system [IHN].....	4
6.2.1 Applicability to language.....	4
6.2.2 Guidance to language users.....	5
6.3 Bit representations [STR].....	5
6.3.1 Applicability to language.....	5
6.3.2 Guidance to language users.....	5
6.4 Floating-point arithmetic [PLF].....	6
6.4.1 Applicability to language.....	6
6.4.2 Guidance to language users.....	6
6.5 Enumerator issues [CCB].....	6
6.5.1 Applicability to language.....	6
6.5.2 Guidance to language users.....	7
6.6 Conversion errors [FLC].....	8
6.6.1 Applicability to language.....	8
6.6.2 Guidance to language users.....	9
6.7 String termination [CJM].....	10
6.7.1 Applicability to language.....	10
6.7.2 Guidance to language users.....	10
6.8 Buffer boundary violation (buffer overflow) [HCB].....	10
6.8.1 Applicability to language.....	10
6.8.2 Guidance to language users.....	11
6.9 Unchecked array indexing [XYZ].....	11
6.9.1 Applicability to language.....	11
6.9.2 Guidance to language users.....	12
6.10 Unchecked array copying [XYW].....	12
6.10.1 Applicability to language.....	12
6.10.2 Guidance to language users.....	12
6.11 Pointer type conversions [HFC].....	13
6.11.1 Applicability to language.....	13
6.11.2 Guidance to language users.....	13
6.12 Pointer arithmetic [RVG].....	13
6.12.1 Applicability to language.....	13
6.12.2 Guidance to language users.....	14
6.13 Null pointer dereference [XYH].....	14
6.13.1 Applicability to language.....	14
6.13.2 Guidance to language users.....	14
6.14 Dangling reference to heap [XYK].....	15
6.14.1 Applicability to language.....	15
6.14.2 Guidance to language users.....	15
6.15 Arithmetic wrap-around error [FIF].....	16
6.15.1 Applicability to language.....	16
6.15.2 Guidance to language users.....	16
6.16 Using shift operations for multiplication and division [PIK].....	17

6.16.1	Applicability to language	17
6.16.2	Guidance to language users	17
6.17	Choice of clear names [NAI]	17
6.17.1	Applicability to language	17
6.17.2	Guidance to language users	17
6.18	Dead store [WXQ]	18
6.18.1	Applicability to language	18
6.18.2	Guidance to language users	18
6.19	Unused variable [YZS]	18
6.19.1	Applicability to language	18
6.19.2	Guidance to language users	18
6.20	Identifier name reuse [YOW]	18
6.20.1	Applicability to language	18
6.20.2	Guidance to language users	19
6.21	Namespace issues [BJL]	19
6.21.1	Applicability to language	19
6.22	Initialization of variables [LAV]	19
6.22.1	Applicability to language	19
6.22.2	Guidance to language users	19
6.23	Operator precedence and associativity [JCW]	19
6.23.1	Applicability to language	19
6.23.2	Guidance to language users	20
6.24	Side-effects and order of evaluation of operands [SAM]	20
6.24.1	Applicability to language	20
6.24.2	Guidance to language users	20
6.25	Likely incorrect expression [KOA]	21
6.25.1	Applicability to language	21
6.25.2	Guidance to language users	21
6.26	Dead and deactivated code [XYQ]	22
6.26.1	Applicability to language	22
6.26.2	Guidance to language users	22
6.27	Switch statements and static analysis [CLL]	22
6.27.1	Applicability to language	22
6.27.2	Guidance to language users	23
6.28	Demarcation of control flow [EO]	23
6.28.1	Applicability to language	23
6.28.2	Guidance to language users	23
6.29	Loop control variables [TEX]	24
6.29.1	Applicability to language	24
6.29.2	Guidance to language users	24
6.30	Off-by-one error [XZH]	25
6.30.1	Applicability to language	25
6.30.2	Guidance to language users	25
6.31	Unstructured programming [EWD]	25
6.31.1	Applicability to language	25
6.31.2	Guidance to language users	25
6.32	Passing parameters and return values [CSJ]	26
6.32.1	Applicability to language	26
6.32.2	Guidance to language users	26
6.33	Dangling references to stack frames [DCM]	27
6.33.1	Applicability to language	27
6.33.2	Guidance to language users	27
6.34	Subprogram signature mismatch [OTR]	27
6.34.1	Applicability to language	27
6.34.2	Guidance to language users	28
6.35	Recursion [GDL]	28
6.35.1	Applicability to language	28
6.35.2	Guidance to language users	28

6.36	Ignored error status and unhandled exceptions [OYB]	28
6.36.1	Applicability to language	28
6.36.2	Guidance to language users	28
6.37	Type-breaking reinterpretation of data [AMV]	29
6.37.1	Applicability to language	29
6.37.2	Guidance to language users	29
6.38	Deep vs. shallow copying [YAN]	29
6.38.1	Applicability to language	29
6.38.2	Guidance to language users	29
6.39	Memory leaks and heap fragmentation [XYL]	30
6.39.1	Applicability to language	30
6.39.2	Guidance to language users	30
6.40	Templates and generics [SYM]	30
6.41	Inheritance [RIP]	30
6.42	Violations of the Liskov substitution principle or the contract model [BLP]	30
6.43	Redispatching [PPH]	30
6.44	Polymorphic variables [BKK]	30
6.45	Extra intrinsics [LRM]	30
6.46	Argument passing to library functions [TRJ]	30
6.46.1	Applicability to language	30
6.46.2	Guidance to language users	31
6.47	Inter-language calling [DJS]	31
6.47.1	Applicability to language	31
6.47.2	Guidance to language users	31
6.48	Dynamically linked code and self-modifying code [NYY]	31
6.48.1	Applicability to language	31
6.48.2	Guidance to language users	32
6.49	Library signature [NSQ]	32
6.49.1	Applicability to language	32
6.49.2	Guidance to language users	32
6.50	Unanticipated exceptions from library routines [HJW]	32
6.51	Pre-processor directives [NMP]	32
6.51.1	Applicability to language	32
6.51.2	Guidance to language users	33
6.52	Suppression of language-defined run-time checking [MXB]	33
6.53	Provision of inherently unsafe operations [SKL]	33
6.53.1	Applicability to language	33
6.53.2	Guidance to language users	33
6.54	Obscure language features [BRS]	34
6.54.1	Applicability of language	34
6.54.2	Guidance to language users	34
6.55	Unspecified behaviour [BQF]	34
6.55.1	Applicability of language	34
6.55.2	Guidance to language users	34
6.56	Undefined behaviour [EWF]	34
6.56.1	Applicability to language	34
6.56.2	Guidance to language users	35
6.57	Implementation-defined behaviour [FAB]	35
6.57.1	Applicability to language	35
6.57.2	Guidance to language users	35
6.58	Deprecated language features [MEM]	36
6.58.1	Applicability to language	36
6.58.2	Guidance to language users	36
6.59	Concurrency — Activation [CGA]	36
6.59.1	Applicability to language	36
6.59.2	Guidance to language users	36
6.60	Concurrency — Directed termination [CGT]	36
6.61	Concurrent data access [CGX]	36

6.61.1	Applicability to language	36
6.61.2	Guidance to language users	37
6.62	Concurrency — Premature termination [CGS]	37
6.62.1	Applicability to language	37
6.62.2	Guidance to language users	37
6.63	Lock protocol errors [CGM]	37
6.63.1	Applicability to language	37
6.63.2	Guidance to language users	37
6.64	Reliance on external format strings	37
6.64.1	Applicability to language	37
6.64.2	Guidance to language users	37
Bibliography		38
Index		39

iTeh STANDARD PREVIEW
(standards.itih.ai)
Full standard:
<https://standards.itih.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020>

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

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

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

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

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

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

This first edition cancels and replaces ISO/IEC TR 24772:2013, which has been split into several parts.

This document is intended to be used with ISO/IEC TR 24772-1, which discusses programming language vulnerabilities in a language independent fashion.

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

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

Introduction

This document provides guidance for the programming language C, so that application developers considering or using C can better 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.

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^{[4][5][7][9][11]to[15]}.

iTeh STANDARD PREVIEW
(standards.iteh.ai)
Full standard:
<https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020>

Programming languages — Guidance to avoiding vulnerabilities in programming languages —

Part 3: 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 ISO/IEC TR 24772-1 are manifested or avoided in the C language.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 2382, *Information technology — Vocabulary*

ISO/IEC 9899, *Information Technology — Programming Languages — C*

ISO/IEC TR 24772-1, *Programming languages — Guidance to avoiding vulnerabilities in programming languages — Part 1: Language-independent guidance*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO/IEC 9899, ISO/IEC TR 24772-1 and the following apply

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

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <http://www.electropedia.org/>

3.1

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

3.2

runtime-constraint

requirement on a program when calling a library function

3.3 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 1 to entry: 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.

4 Language concepts

The C programming language was developed in the early 1970s at Bell Labs, in support of the development of the Unix operating system. It was conceived as a "high-level assembler", with a small semantic gap between code and executable.

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
- maintaining 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.

The following are some key features of the language.

- 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 "shortcuts" by allowing component statements to be constructed that would not be syntactically legal using a function definition.
- Since ISO/IEC 9899:2011, 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 Avoiding programming language vulnerabilities in C

ISO/IEC TR 24772-1:2019, 5.4, supplies what are regarded as the most relevant language-independent rules, as a summary of that document. These obviously apply to C, however in addition, this clause

lists those rules from [Clause 6](#) that are stated most frequently, or that are considered as particularly noteworthy.

Index		Subclause in this document
1	Use a macro to ensure that the size of memory allocated with <code>malloc</code> matches the intended type of the object.	6.11 Pointer type conversions [HFC]
2	Use bounds checking interfaces from ISO/IEC 9899:2018, Annex K, in favour of non-bounds checking interfaces, such as <code>strcpy_s</code> instead of <code>strcpy</code> .	6.8 Buffer boundary violation (buffer overflow) [HCB]
3	Use commonly available functions such as functions of ISO/IEC 9945 <code>htonl()</code> , <code>htons()</code> , <code>ntohl()</code> and <code>ntohs()</code> 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 <code>memcpy</code> and <code>memmove</code>), 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 <code>free</code> , set the pointer to null to prevent multiple deallocation or use of a dangling reference via this pointer, as illustrated in the following code: <pre>free (ptr); ptr = NULL;</pre>	6.14 Dangling reference to heap [XYK]
7	Do not read uninitialized memory, including memory allocated by functions such as <code>malloc</code> .	6.22 Initialization of variables [LAV]
8	Check that the result of an operation on an unsigned integer value does 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: <pre>a + b a - b a * b a++ ++a a-- --a a += b a -= b a *= b a << b a <=<= b -a</pre>	6.15 Arithmetic wrap-around error [FIF]
9	Check that the result of an operation on a signed integer value does 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: <pre>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</pre>	6.15 Arithmetic wrap-around error [FIF]
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 TR 24772-1 and provides specific guidance on how to avoid them in C code. This clause mirrors ISO/IEC TR 24772-1:2019, Clause 6, in that the vulnerability “Type System [IHN]” is found in ISO/IEC TR 24772-1:2019, 6.2, and C specific guidance is found in [6.2](#).

6.2 Type system [IHN]

6.2.1 Applicability to language

C is a statically typed language. In some ways, C is both strongly and weakly typed as it requires all variables to be typed, but sometimes allows implicit or automatic conversion between types. For example, C 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 does not discard any digits since they are the same size. In some implementations, all integer types can be implemented as the same size.

iTeh STANDARD PREVIEW
(standards.iteh.ai)
Full standard:
<https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020>

C allows implicit conversions as in the following example:

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

If an implicit conversion could 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 issue a warning message.

C has a set of rules to determine how conversion between data types occurs. 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 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 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.

6.3 Bit representations [STR]

6.3.1 Applicability to language

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.

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 are in either little-endian or big-endian format and unknowingly switching between the two can unexpectedly alter values.

6.3.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.3.5.