TECHNICAL REPORT

ISO/IEC TR 24772-3

First edition
2020-05

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)
ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020

Reference number
ISO/IEC TR 24772-3:2020(E)

© ISO/IEC 2020

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TR 24772-3:2020
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

© ISO/IEC 2020 – All rights reserved

Contents

Page

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-e5df-4b96-9b4a-b3958adb98/iso-iec-tr-24772-3-2020

© ISO/IEC 2020 – All rights reserved

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-0e3b58c4b98/iso-iec-tr-24772-3-2020

© ISO/IEC 2020 – All rights reserved

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-e5df-4b96-9b4a-b3058eddb981/iso-iec-tr-24772-3-2020

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

© ISO/IEC 2020 – All rights reserved

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.

iTeh STANDARD PREVIEW
(standards.iteh.ai)

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*.

ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-b2058[illegible]-24772-3-2020

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.

© ISO/IEC 2020 – All rights reserved

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)
ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020

© ISO/IEC 2020 – All rights reserved

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)

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 TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3098e1db98/iso-iec-tr-24772-3-2020

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

© ISO/IEC 2020 – All rights reserved

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.

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d8-e1af-4b90-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020

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

© ISO/IEC 2020 – All rights reserved

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 malloc 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 `strcpy_s` instead of `strcpy`.	6.8 Buffer boundary violation (buffer overflow) [HCB]
3	Use commonly available functions such as functions of ISO/IEC 9945 `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 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:
	`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 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:
	`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`
		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]

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-e9df-4b76-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020

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.

© ISO/IEC 2020 – All rights reserved

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)
ISO/IEC TR 24772-3:2020
https://standards.iteh.ai/catalog/standards/sist/60d8e2d6-c5df-4b96-9b4a-be3058efdb98/iso-iec-tr-24772-3-2020

© ISO/IEC 2020 – All rights reserved

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.

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

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.

© ISO/IEC 2020 – All rights reserved

— 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 ISO/IEC 9945 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

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.

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TR 24772-3:2020

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 do not have the expected behaviour. For example:

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

may or may 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))
```

may or may not evaluate to true. Given that **`x`** and **`y`** are constant values, it is expected that consistent results are 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.

6.4.2 Guidance to language users

Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.4.5.

6.5 Enumerator issues [CCB]

6.5.1 Applicability to language

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

 © ISO/IEC 2020 – All rights reserved

```
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 takes 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 is 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 may not behave as expected:

  — the user may expect all possible values to be covered. However, if the control expression is a variable assigned `Eighth+1`, then the code "falls through", without executing any of the case statements

  — 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 may not exist.

— 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 may be created that has a value greater than `Last_member`'s, so there is undefined behaviour if this member is used to index the array

  — the values covered by the modified enumeration type members may 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 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.

6.5.2 Guidance to language users

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TR 24772-3:2020

© ISO/IEC 2020 – All rights reserved