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

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

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

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].

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 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]

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.

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.

- 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.

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:

```
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;.`

  - 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 should not optimize it away, but a number of them have been found that do).

6.6 Conversion errors [FLC]

6.6.1 Applicability to language

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

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

This implicit conversion discards 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 ISO/IEC 9899. 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;
```

ISO/IEC 9899 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 that has been added to ISO/IEC 9899 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 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 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 results 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 both removes the warning and acknowledges 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`.

6.7 String termination [CJM]

6.7.1 Applicability to language

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 Guidance to language users

— Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.7.5.

— Use the safer and more secure functions for string handling that are defined in ISO/IEC 9899:2018, Annex K1) or ISO/IEC TR 24731-2. Both of these define alternative string handling library functions to the current Standard C Library (ISO/IEC 9899:2018, Annex B). 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.

6.8 Buffer boundary violation (buffer overflow) [HCB]

6.8.1 Applicability to language

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];
}
```

returns 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;
  }
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:

---

1) See comments on the correct use of ISO/IEC 9899:2018, Annex K functions in 6.8.

```
   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 then holds “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 ISO/IEC 9899:2018, Annex K.

```
   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 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 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.
- Use the safer and more secure functions for string handling from ISO/IEC 9899:2018, Annex K, 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 that 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()`.

6.9 Unchecked array indexing [XYZ]

6.9.1 Applicability to language

C does not perform bounds checking on arrays so, although arrays can be accessed outside of their bounds[2], the value returned is undefined and in some cases may result in a program termination. For

---

[2] Actually, this is a special case of 6.8.

example, in C the following code is valid, though, for example, if `i` has the value 10 it leads to unspecified behaviour:

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

The variable `t` is likely 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 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 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 ISO/IEC 9899:2018, Annex K[3]. 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.

6.10 Unchecked array copying [XYW]

6.10.1 Applicability to language

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[4]. This leads to unspecified behaviour.

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 Guidance to language users

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

---

[3] See comments on the correct use of functions of ISO/IEC 9899:2018, Annex K, in 6.8.1.

[4] This also is a special case of 6.8.

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 ISO/IEC 9899:2018, Annex K[3].

6.11 Pointer type conversions [HFC]

6.11.1 Applicability to language

C allows casting of the value of a pointer to and from another data type. These conversions can cause unexpected changes to pointer values.

If a pointer is cast to a different type and then pointer arithmetic is applied (including array indexing) then the memory accessed may not be 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 may not give the expected results because of the possible presence of padding bytes.

The one safe pointer conversion is from a pointer to some object type to `void*` and then back to the original pointer type. ISO/IEC 9899 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 Guidance to language users

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

6.12 Pointer arithmetic [RVG]

6.12.1 Applicability to language

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` 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 illustrates 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` results in `buf_ptr == 0x1238` on a host where an `int` is 4 bytes; `buf_ptr` then points to `buf[5]`. Not realizing that address operations are in terms of the size of the object being pointed to can lead to address miscalculations and undefined behaviour.

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 Guidance to language users

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

6.13 Null pointer dereference [XYH]

6.13.1 Applicability to language

C allows memory to be dynamically allocated primarily through the use of `malloc()`, `calloc()`, and `realloc()`. Each returns the address to the allocated memory. Due to a variety of situations, the memory allocation may not occur as expected and a null pointer is 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 behaviour (such as a segmentation fault).

Space for 10 000 integers can be dynamically allocated in C in the following way[5]:

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

`malloc()` returns 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 is now allocated
```

Memory allocations usually succeed, so neglecting this test and using the memory usually works. That is why neglecting the `null` test frequently goes unnoticed. However, an attacker can intentionally create a situation where the memory allocation fails, leading to undefined behaviour.

6.13.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.13.5.
- 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.

---

[5] This allocation does not follow the advice in 6.11, for simplicity.

6.14 Dangling reference to heap [XYK]

6.14.1 Applicability to language

C allows memory to be dynamically allocated primarily through the use 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 is now 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 behaviour. 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 is unlikely to be noticed. However, if the memory has been reallocated, altering of the data contained in the memory almost certainly results in data corruption. Determining that a dangling memory reference is the cause of a problem and locating it can be difficult.

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
                                        is now 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 Guidance to language users

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

6.15 Arithmetic wrap-around error [FIF]

6.15.1 Applicability to language

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, the programmer can be unaware that the value is getting too big to represent and thus not expect wrap-around behaviour. As it is impossible for the compiler or an analysis tool to determine what the programmer intended, it is better 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 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.15.2.
- 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
  ```

- 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
  ```

- Use defensive programming techniques to check whether an operation overflows or underflows 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 issues for C are well defined in ISO/IEC TR 24772-1:2019, 6.16 (see also 6.15).

6.16.2 Guidance to language users

Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.16.5 (see also 6.15).

6.17 Choice of clear names [NAI]

6.17.1 Applicability to language

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 on 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 truncates them to the maximum. Therefore, if two names in a program only differ in characters after the maximum, they are 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, as they are both mechanisms by which the programmer can inadvertently use an object other than the one intended.

6.17.2 Guidance to language users

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

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 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 Guidance to language users

— Follow the guidance contained in by ISO/IEC TR 24772-1:2019, 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 (see also 6.61).

6.19 Unused variable [YZS]

6.19.1 Applicability to language

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

6.19.2 Guidance to language users

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

6.20 Identifier name reuse [YOW]

6.20.1 Applicability to language

C allows scoping so that a variable that is not declared locally is resolved to some outer block and that resolution may cause the variable to operate on an entity other than the one intended.

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

This issue is related to 6.17, as they are both mechanisms by which the programmer can inadvertently use an object other than the one intended.

6.20.2 Guidance to language users

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

6.21 Namespace issues [BJL]

6.21.1 Applicability to language

Does 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.

6.22 Initialization of variables [LAV]

6.22.1 Applicability to language

Local, automatic variables can assume unexpected values if they are used before they are initialized. ISO/IEC 9899:2018 specifies: "If an object that has automatic storage duration is not initialized explicitly, its value is 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 may 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.

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

6.22.2 Guidance to language users

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

6.23 Operator precedence and associativity [JCW]

6.23.1 Applicability to language

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 Guidance to language users

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

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

6.24.1 Applicability to language

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++];
```

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 (ISO/IEC 9899:2018, 6.5.2.2);
- the order of evaluation of the operands in an assignment statement (ISO/IEC 9899:2018, 6.5.16);
- the order in which any side effects occur among the initialization list expressions is unspecified. In particular, the evaluation order does not need to be the same as the order of subobject initialization (ISO/IEC 9899:2018, 6.7.9).

Because these are unspecified behaviours, testing can give the false impression that the code is working and portable, when 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 allowed to perform calculations and assignments in any order between sequence points. ISO/IEC 9899:2018, Annex C, 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 are allowed to 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 do not 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 are allowed to be evaluated in any order, and if they modify common variables the result is unspecified.

6.24.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.24.5.
- Write expressions so that the same effects occur under any order of evaluation that ISO/IEC 9899 permits since side effects can be dependent on an implementation specific order of evaluation.
- Become familiar with ISO/IEC 9899:2018, Annex C, which is a list of the sequence points that enforce an ordering of computations within an expression.

6.25 Likely incorrect expression [KOA]

6.25.1 Applicability to language

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 (x != 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 is always executed. In this case, it is extremely likely that the programmer did not intend to put the semi-colon there.

6.25.2 Guidance to language users

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

```
int a,b,c,d;
/* … */
if ((a == b) || (c = (d-1)))   // the assignment to c will 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

C allows the usual sources of dead code (described in ISO/IEC TR 24772-1:2019, 6.26) 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 Guidance to language users

Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.26.5.

6.27 Switch statements and static analysis [CLL]

6.27.1 Applicability to language

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.

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 is no default case and the switched expression does not 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 causes subsequent cases to be executed until a break or the end of the switch block is reached. This can cause unexpected results.

6.27.2 Guidance to language users

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

C lacks a keyword to be used as an explicit terminator. Therefore, it may not be readily apparent which statements are part of a loop construct or an if statement.

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 is erroneously performed only once, since there are no enclosing brackets.

If statements in C are also susceptible to control flow problems since there is no 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 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.28.5.

- Enclose the bodies of `if`, `else`, `while`, `for`, and similar constructs in braces. This reduces 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.

6.29 Loop control variables [TEX]

6.29.1 Applicability to language

C allows the modification of loop control variables within the loop but can cause unexpected behaviour.

Since the modification of a loop control variable within a loop is infrequently encountered, reviewers of C code may not expect it and therefore 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;
   …
}
```

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

C does not 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 thus an unterminated loop. The following code either loops ten times or indefinitely:

```
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 may 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`.

6.29.2 Guidance to language users

- Follow the guidance of ISO/IEC TR 24772-1:2019, 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.

6.30 Off-by-one error [XZH]

6.30.1 Applicability to language

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 is not easily detectable. Several tools can be used to help detect accesses beyond the bounds of arrays. However, such tools do 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 Guidance to language users

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

It is as easy to write structured programs in C as it is not to. 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 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 may not be detected as readily or at all as would be the case if the software was written in a structured manner.

IEC 61508 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 Guidance to language users

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

6.32 Passing parameters and return values [CSJ]

6.32.1 Applicability to language

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`.

ISO/IEC 9899:1999 introduced the `restrict` keyword. It may 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 is never 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 copies a fixed number of characters from a source string to a target. If the target overlaps the source, the result depends on 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 does not occur (see 6.51).

6.32.2 Guidance to language users

— Follow the guidance contained in ISO/IEC TR 24772-1:2019, 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 may be an alias[6]. Aliases can be avoided by following the respective guidelines of ISO/IEC TR 24772-1:2019, 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.

---

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

6.33 Dangling references to stack frames [DCM]

6.33.1 Applicability to language

C allows the address of a variable to be stored in a pointer variable. If this pointer variable contains, 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 has been made available for further allocation and may 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.

6.33.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.33.5.
- Do not assign the address of an object to any entity which persists after the object has ceased to exist. 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

If it is necessary to call a function that is not yet defined in the current translation unit[7], 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 accepts any number and type of parameters in the call.

C compilers 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:

---

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

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

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

6.34.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.34.5.
- 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 and is therefore subject to the problems described in ISO/IEC TR 24772-1:2019, 6.35.

6.35.2 Guidance to language users

Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.35.5.

6.36 Ignored error status and unhandled exceptions [OYB]

6.36.1 Applicability to language

ISO/IEC 9899 does not include exception handling. Therefore, only error status is 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 is expected to 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 ISO/IEC 9899, `errno` may be used whether or not there is an error). Though these values are defined by ISO/IEC 9899, inconsistencies in responding to error conditions can lead to vulnerabilities.

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

C library functions may also return error indicator values.

6.36.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 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.
- 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. ISO/IEC 9899:2018, Annex K, 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 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 TR 24772-1:2019, 6.37. The only pointer casting that ISO/IEC 9899 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 Guidance to language users

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

6.38 Deep vs. shallow copying [YAN]

6.38.1 Applicability to language

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 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.38.5.
- Where necessary, create a function to correctly perform the deep copy.

6.39 Memory leaks and heap fragmentation [XYL]

6.39.1 Applicability to language

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

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()`. If there is a flaw in the logic of the program, memory may 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.

6.39.2 Guidance to language users

- Follow the guidelines of ISO/IEC TR 24772-1:2019, 6.39.5.
- Use debugging tools such as leak detectors to help identify unreachable memory.
- 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]

This vulnerability does not apply to C because C does not implement these mechanisms.

6.41 Inheritance [RIP]

This vulnerability does not apply to C because C does not implement struct hierarchies.

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

This vulnerability does not apply to C because C does not implement polymorphism.

6.43 Redispatching [PPH]

This vulnerability does not apply to C because C does not implement this mechanism.

6.44 Polymorphic variables [BKK]

This vulnerability does not apply to C because C does not implement this mechanism.

6.45 Extra intrinsics [LRM]

This vulnerability does not apply to C because C does not implement these mechanisms.

6.46 Argument passing to library functions [TRJ]

6.46.1 Applicability to language

There is no guarantee that the parameters being passed to a function are verified by either the calling or receiving functions. So, values outside of the assumed range may be received by a function resulting in a 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 but is outside of the expected range, and then an operation or series of operations is performed, using that value of the parameter can result in unanticipated results and even a potential vulnerability.

6.46.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.46.5.
- Do not make assumptions about the values of parameters.
- Do not assume that the calling or receiving function is range checking a parameter. Therefore, establish a strategy for each interface to check parameters in either the calling or receiving routines.

6.47 Inter-language calling [DJS]

6.47.1 Applicability to language

ISO/IEC 9899 defines the calling conventions, data layout, error handing and return conventions needed to use C from another language. Ada has developed ISO/IEC 8652 for interfacing with C. Fortran that explains how to call C functions ISO/IEC 1539-1:2018, Clause 15. Calls from C into other languages become the responsibility of the programmer.

6.47.2 Guidance to language users

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

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

6.48.1 Applicability to language

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 ISO/IEC 9899.

C permits self-modifying code. In C, there is no 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 Guidance to language users

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

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 need to 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.

Similarly, every other programming language has differences in representations with C so different issues arise with each interface that the programmer must be aware of and make adjustments before the invocation of any foreign language code.

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

6.49.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.49.5.
- Use a tool, if possible, to automatically create interface wrappers.

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 does not apply. See 6.36 for a discussion of Ignored errors. See ISO/IEC TR 24772-1:2019, 6.47, in the case where libraries written in languages that use exceptions are called.

6.51 Pre-processor directives [NMP]

6.51.1 Applicability to language

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 may 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, 6.10. Additionally, the arguments and body of function-like macros should be fully parenthesized to avoid unintended and unspecified behaviour.

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):

```
#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 it is not known in which order the compiler evaluates 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:

```
#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 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019, 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 needs to be used, 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.

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

Does not apply to C since there are no language-defined runtime checks.

6.53 Provision of inherently unsafe operations [SKL]

6.53.1 Applicability to language

C was designed for implementing system software where some "unsafe" operations are inherent and common.

6.53.2 Guidance to language users

Follow the guidance contained in ISO/IEC TR 24772-1:2019, 6.53.5.