TECHNICAL SPECIFICATION

ISO/IEC TS 17961

First edition
2013-11-15

Information technology — Programming languages, their environments and system software interfaces — C secure coding rules

Technologies de l'information — Langages de programmation, leur environnement et interfaces des logiciels de systèmes — Règles de programmation sécurisée en C

Reference number
ISO/IEC TS 17961:2013(E)

Contents

Page

Foreword

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

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

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

In other circumstances, particularly when there is an urgent market requirement for such documents, the joint technical committee may decide to publish an ISO/IEC Technical Specification (ISO/IEC TS), which represents an agreement between the members of the joint technical committee and is accepted for publication if it is approved by 2/3 of the members of the committee casting a vote.

An ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn.

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.

ISO/IEC TS 17961 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Introduction

Background

An essential element of secure coding in the C programming language is a set of well-documented and enforceable coding rules. The rules specified in this Technical Specification apply to analyzers, including static analysis tools and C language compiler vendors that wish to diagnose insecure code beyond the requirements of the language standard. All rules are meant to be enforceable by static analysis.

The application of static analysis to security has been done in an ad hoc manner by different vendors, resulting in nonuniform coverage of significant security issues. This specification enumerates secure coding rules and requires analysis engines to diagnose violations of these rules as a matter of conformance to this specification. These rules may be extended in an implementation-dependent manner, which provides a minimum coverage guarantee to customers of any and all conforming static analysis implementations.

The largest underserved market in security is ordinary, non-security-critical code. The security-critical nature of code depends on its purpose rather than its environment. The UNIX finger daemon (`fingerd`) is an example of ordinary code, even though it may be deployed in a hostile environment. A user runs the client program, `finger`, which sends a user name to `fingerd` over the network, which then sends a reply indicating whether the user is logged in and a few other pieces of information. The function of `fingerd` has nothing to do with security. However, in 1988, Robert Morris compromised `fingerd` by triggering a buffer overflow, allowing him to execute arbitrary code on the target machine. The Morris worm could have been prevented from using `fingerd` as an attack vector by preventing buffer overflows, regardless of whether `fingerd` contained other types of bugs.

By contrast, the function of `/bin/login` is purely related to security. A bug of any kind in `/bin/login` has the potential to allow access where it was not intended. This is security-critical code.

Similarly, in safety-critical code, such as software that runs an X-ray machine, any bug at all could have serious consequences. In practice, then, security-critical and safety-critical code have the same requirements.

There are already standards that address safety-critical code and therefore security-critical code. The problem is that because they must focus on preventing essentially all bugs, they are required to be so strict that most people outside the safety-critical community do not want to use them. This leaves ordinary code like `fingerd` unprotected.

This Technical Specification has two major subdivisions:

— preliminary elements (Clauses 1–4) and

— secure coding rules (Clause 5).

Each secure coding rule in Clause 5 has a separate numbered subsection and a unique section identifier enclosed in brackets (for example, [ptrcomp]). The unique section identifiers are mainly for use by other documents in identifying the rules should the section numbers change because of the addition or elimination of a rule. These identifiers may be used in diagnostics issued by conforming analyzers, but analyzers are not required to do so.

Annexes provide additional information. Annex C (informative) Related Guidelines and References identifies related guidelines and references per rule. A bibliography lists documents referred to during the preparation of this Technical Specification.

The rules documented in this Technical Specification do not rely on source code annotations or assumptions of programmer intent. However, a conforming implementation may take advantage of annotations to inform the analyzer. The rules, as specified, are reasonably simple, although complications can exist in identifying exceptions. An analyzer that conforms to this Technical Specification should be able to analyze code without excessive false positives, even if the code was developed without the expectation that it would be analyzed. Many analyzers provide methods that eliminate the need to research each diagnostic on every invocation of the analyzer. The implementation of such a mechanism is encouraged but not required. This Technical Specification assumes that an analyzer's visibility extends beyond the boundaries of the current function or translation unit being analyzed (see Annex A (informative) Intra- to Interprocedural Transformations).

Completeness and soundness

The rules specified in this Technical Specification are designed to provide a check against a set of programming flaws that are known from practical experience to have led to vulnerabilities. Although rule checking can be performed manually, with increasing program complexity, it rapidly becomes infeasible. For this reason, the use of static analysis tools is recommended.

It should be recognized that, in general, determining conformance to coding rules is computationally undecidable. The precision of static analysis has practical limitations. For example, the *halting theorem* of Computer Science states that there are programs whose exact control flow *cannot* be determined statically. Consequently, any property dependent on control flow—such as halting—may be indeterminate for some programs. A consequence of this undecidability is that it may be impossible for *any* tool to determine statically whether a given rule is satisfied in specific circumstances. The widespread presence of such code may also lead to unexpected results from an analysis tool. Annex D (informative) Decidability of Rules provides information on the decidability of rules in this Technical Specification.

However checking is performed, the analysis may generate

— false negatives: Failure to report a real flaw in the code is usually regarded as the most serious analysis error, as it may leave the user with a false sense of security. Most tools err on the side of caution and consequently generate false positives. However, there may be cases where it is deemed better to report some high-risk flaws and miss others than to overwhelm the user with false positives.

— false positives: The tool reports a flaw when one does not exist. False positives may occur because the code is sufficiently complex that the tool cannot perform a complete analysis. The use of features such as function pointers and libraries may make false positives more likely.

To the greatest extent feasible, an analyzer should be both complete and sound with respect to enforceable rules. An analyzer is considered sound with respect to a specific rule if it cannot give a false-negative result, meaning it finds all violations of a rule within the entire program. An analyzer is considered complete if it cannot issue false-positive results, or false alarms. The possibilities for a given rule are outlined in Table 1.

Table 1 — Completeness and soundness

		False positives	
		Y	N
False negatives	N	Sound with false positives	Complete and sound
	Y	Unsound with false positives	Complete and unsound

The degree to which conforming analyzers minimize false-positive diagnostics is a quality of implementation issue. In other words, quantitative thresholds for false positives and false negatives are outside the scope of this Technical Specification.

Security focus

The purpose of this Technical Specification is to specify analyzable secure coding rules that can be automatically enforced to detect security flaws in C-conforming applications. To be considered a security flaw, a software bug must be triggerable by the actions of a malicious user or attacker. An attacker may trigger a bug by providing malicious data or by providing inputs that execute a particular control path that in turn executes the security flaw. Implementers are encouraged to distinguish violations that operate on untrusted data from those that do not.

Taint analysis

Taint and tainted sources

Certain operations and functions have a domain that is a subset of the type domain of their operands or parameters. When the actual values are outside of the defined domain, the result might be either undefined or at least unexpected. If the value of an operand or argument may be outside the domain of an operation or function that consumes that value, and the value is derived from any external input to the program (such as a command-line argument, data returned from a system call, or data in shared memory), that value is *tainted*, and its origin is known as a *tainted source*. A tainted value is not necessarily known to be out of the domain; rather, it is not known to be in the domain. Only values, and not the operands or arguments, can be tainted; in some cases, the same operand or argument can hold tainted or untainted values along different paths. In this regard, *taint* is an attribute of a value originating from a tainted source.

Restricted sinks

Operands and arguments whose domain is a subset of the domain described by their types are called *restricted sinks*. Any pointer arithmetic operation involving an integer operand is a restricted sink for that operand. Certain parameters of certain library functions are restricted sinks because these functions perform address arithmetic with these parameters, or control the allocation of a resource, or pass these parameters on to another restricted sink. All string input parameters to library functions are restricted sinks because it is possible to pass in a character sequence that is not null terminated. The exceptions are `strncpy` and `strncpy_s`, which explicitly allow the source character sequence not to be null-terminated. For purposes of this Technical Specification, we regard `char *` as a reference to a null-terminated array of characters.

Propagation

Taint is propagated through operations from operands to results unless the operation itself imposes constraints on the value of its result that subsume the constraints imposed by restricted sinks. In addition to operations that propagate the same sort of taint, there are operations that propagate taint of one sort of an operand to taint of a different sort for their results, the most notable example of which is `strlen` propagating the taint of its argument with respect to string length to the taint of its return value with respect to range.

Although the exit condition of a loop is not normally itself considered to be a restricted sink, a loop whose exit condition depends on a tainted value propagates taint to any numeric or pointer variables that are increased or decreased by amounts proportional to the number of iterations of the loop.

Sanitization

To remove the taint from a value, it must be *sanitized* to ensure that it is in the defined domain of any restricted sink into which it flows. Sanitization is performed by *replacement* or *termination*. In replacement, out-of-domain values are replaced by in-domain values, and processing continues using an in-domain value in place of the original. In termination, the program logic terminates the path of execution when an out-of-domain value is detected, often simply by branching around whatever code would have used the value.

In general, sanitization cannot be recognized exactly using static analysis. Analyzers that perform taint analysis usually provide some extralinguistic mechanism to identify sanitizing functions that sanitize an argument (passed by address) in place, return a sanitized version of an argument, or return a status code indicating whether the argument is in the required domain. Because such extralinguistic mechanisms are outside the scope of this specification, this Technical Specification uses a set of rudimentary definitions of sanitization that is likely to recognize real sanitization but might cause nonsanitizing or ineffectively sanitizing code to be misconstrued as sanitizing. The following definition of sanitization presupposes that the analysis is in some way maintaining a set of constraints on each value encountered as the simulated execution progresses: *a given path through the code sanitizes a value with respect to a* *given restricted sink if it restricts the range of that value to a subset of the defined domain of the restricted sink type*. For example, sanitization of signed integers with respect to an array index operation must restrict the range of that integer value to numbers between zero and the size of the array minus one.

This description is suitable for numeric values, but sanitization of strings with respect to content is more difficult to recognize in a general way.

Tainted source macros

The function-like macros `GET_TAINTED_STRING` and `GET_TAINTED_INTEGER` defined in this section are used in the examples in this Technical Specification to represent one possible method to obtain a tainted string and tainted integer.

```
#define GET_TAINTED_STRING(buf, buf_size)     \
  do {                                         \
    const char *taint = getenv("TAINT");       \
    if (taint == 0) {                          \
      exit(1);                                 \
    }                                          \
                                               \
    size_t taint_size = strlen(taint) + 1;     \
    if (taint_size > buf_size) {               \
      exit(1);                                 \
    }                                          \
                                               \
    strncpy(buf, taint, taint_size);           \
  } while (0)

#define GET_TAINTED_INTEGER(type, val)              \
  do {                                               \
    const char *taint = getenv("TAINT");             \
    if (taint == 0) {                                \
      exit(1);                                       \
    }                                                \
                                                     \
    errno = 0;                                       \
    long tmp = strtol(taint, 0, 10);                 \
    if ((tmp == LONG_MIN || tmp == LONG_MAX) &&      \
        errno == ERANGE)                             \
      ; /* retain LONG_MIN or LONG_MAX */            \
    if ((type)-1 < 0) {                              \
      if (tmp < INT_MIN)                             \
        tmp = INT_MIN;                               \
      else if (tmp > INT_MAX)                        \
        tmp = INT_MAX;                               \
    }                                                \
    val = tmp;                                       \
  } while (0)
```

Information technology — Programming languages, their environments and system software interfaces — C secure coding rules

1 Scope

This Technical Specification specifies

— rules for secure coding in the C programming language and

— code examples.

This Technical Specification does not specify

— the mechanism by which these rules are enforced or

— any particular coding style to be enforced. (It has been impossible to develop a consensus on appropriate style guidelines. Programmers should define style guidelines and apply these guidelines consistently. The easiest way to consistently apply a coding style is with the use of a code formatting tool. Many interactive development environments provide such capabilities.)

Each rule in this Technical Specification is accompanied by code examples. Code examples are informative only and serve to clarify the requirements outlined in the normative portion of the rule. Examples impose no normative requirements.

Each rule in this Technical Specification that is based on undefined behavior defined in the C Standard identifies the undefined behavior by a numeric code. The numeric codes for undefined behaviors can be found in Annex B, Undefined Behavior.

Two distinct kinds of examples are provided:

— *noncompliant examples* demonstrating language constructs that have weaknesses with potentially exploitable security implications; such examples are expected to elicit a diagnostic from a conforming analyzer for the affected language construct; and

— *compliant examples* are expected not to elicit a diagnostic.

Examples are not intended to be complete programs. For brevity, they typically omit `#include` directives of C Standard Library headers that would otherwise be necessary to provide declarations of referenced symbols. Code examples may also declare symbols without providing their definitions if the definitions are not essential for demonstrating a specific weakness.

Some rules in this Technical Specification have exceptions. Exceptions are part of the specification of these rules and are normative.

2 Conformance

In this Technical Specification, "shall" is to be interpreted as a requirement on an analyzer; conversely, "shall not" is to be interpreted as a prohibition.

Various types of programs (such as compilers or specialized analyzers) can be used to check if a program contains any violations of the coding rules specified in this Technical Specification. In this Technical Specification, all such checking programs are called analyzers. An analyzer can claim conformity with this Technical Specification. Programs that do not yield any diagnostic when analyzed by a conforming analyzer cannot claim conformity to this Technical Specification.

A conforming analyzer shall produce a diagnostic for each distinct rule in this Technical Specification upon detecting a violation of that rule, except in the case that the same program text violates multiple rules simultaneously, where a conforming analyzer may aggregate diagnostics but shall produce at least one diagnostic.

NOTE 1 The diagnostic message might be of the form:

```
Accessing freed memory in function abc, file xyz.c, line nnn.
```

NOTE 2 This Technical Specification does not require an analyzer to produce a diagnostic message for any violation of any syntax rule or constraint specified by the C Standard.

Conformance is defined only with respect to source code that is visible to the analyzer. Binary-only libraries, and calls to them, are outside the scope of these rules.

For each rule, the analyzer shall report a diagnostic for at least one program that contains a violation of that rule.

For each rule, the analyzer shall document whether its analysis is guaranteed to report all violations of that rule and shall document its accuracy with respect to avoiding false positives and false negatives.

2.1 Portability assumptions

A conforming analyzer shall be able to diagnose violations of guidelines for at least one C implementation. An analyzer need not diagnose a rule violation if the result is documented for the target implementation and does not cause a security flaw. A conforming analyzer shall document which C implementation is the target.

Variations in quality of implementation permit an analyzer to produce diagnostics concerning portability issues.

EXAMPLE

```
long i;
printf("i = %d", i);
```

This example can produce a diagnostic, such as the mismatch between `%d` and `long int`. This Technical Specification does not specify that a conforming analyzer be complete or sound when diagnosing rule violations. This mismatch might not be a problem for all target implementations, but it is a portability problem because not all implementations have the same representation for `int` and `long`.

3 Normative references

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

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

ISO 80000-2:2009, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*

ISO/IEC 2382-1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*

ISO/IEC/IEEE 9945:2009, *Information technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*

4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9899:2011, ISO/IEC 2382-1:1993 and the following apply. Other terms are defined where they appear in *italic* type. Mathematical symbols not defined in this document are to be interpreted according to ISO 80000-2:2009.

4.1
analyzer
mechanism that diagnoses coding flaws in software programs

Note 1 to entry: Analyzers may include static analysis tools, tools within a compiler suite, or tools in other contexts.

4.2
data flow analysis
tracking of value constraints along nonexcluded paths through the code

Note 1 to entry: Tracking can be performed intraprocedurally, with various assumptions made about what happens at function call boundaries, or interprocedurally, where values are tracked flowing into function calls (directly or indirectly) as arguments and flowing back out either as return values or indirectly through arguments.

Note 2 to entry: Data flow analysis may or may not track values flowing into or out of the heap or take into account global variables. When this specification refers to values flowing, the key point is contrast with variables or expressions, because a given variable or expression may hold different values along different paths, and a given value may be held by multiple variables or expressions along a path.

4.3
exploit
technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy

4.4
in-band error indicator
library function return value on error that can never be returned by a successful call to that library function

4.5
mutilated value
result of an operation performed on an untainted value that yields either an undefined result (such as the result of signed integer overflow), the result of right-shifting a negative number, implicit conversion to an integral type where the value cannot be represented in the destination type, or unsigned integer wrapping

EXAMPLE

```
int j = INT_MAX + 1; // j is mutilated
char c = 1234; // c is mutilated if char is eight bits
unsigned int u = 0U - 1; // u is mutilated
```

Note 1 to entry: A mutilated value can be just as dangerous as a tainted value because it can differ either in sign or magnitude from what the programmer expects.

4.6
nonpersistent signal handler
signal handler running on an implementation that requires the program to again register the signal handler after occurrences of the signal to catch subsequent occurrences of that signal

4.7
out-of-band error indicator
library function return value used to indicate nothing but the error status

4.8
out-of-domain value
one of a set of values that is not in the domain of a particular operator or function

4.9
restricted sink
operands and arguments whose domain is a subset of the domain described by their types

Note 1 to entry: Undefined or unexpected behavior may occur if a tainted value is supplied as a value to a restricted sink.

Note 2 to entry: A diagnostic is required if a tainted value is supplied to a restricted sink.

Note 3 to entry: Different restricted sinks may impose different validity constraints for the same value; a given value can be tainted with respect to one restricted sink but sanitized (and consequently no longer tainted) with respect to a different restricted sink.

Note 4 to entry: Specific restricted sinks and requirements for sanitizing tainted values are described in specific rules dealing with taint analysis (see 5.8, 5.14, 5.24, 5.30, 5.39, and 5.46).

4.10
sanitize
assure by testing or replacement that a tainted or other value conforms to the constraints imposed by one or more restricted sinks into which it may flow

Note 1 to entry: If the value does not conform, either the path is diverted to avoid using the value or a different, known-conforming value is substituted.

EXAMPLE Adding a null character to the end of a buffer before passing it as an argument to the `strlen` function.

4.11
security flaw
defect that poses a potential security risk

4.12
security policy
set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources

4.13
static analysis
any process for assessing code without executing it

Note 1 to entry: See [1], p. 3.

4.14
tainted source
external source of untrusted data

Note 1 to entry: Tainted sources include

— parameters to the `main` function,

— the returned values from `localeconv`, `fgetc`, `getc`, `getchar`, `fgetwc`, `getwc`, and `getwchar`, and

— the strings produced by `getenv`, `fscanf`, `vfscanf`, `vscanf`, `fgets`, `fread`, `fwscanf`, `vfwscanf`, `vwscanf`, `wscanf`, and `fgetws`.

4.15
tainted value
value derived from a tainted source that has not been sanitized

4.16
target implementation
implementation of the C programming language whose environmental limits and implementation-defined behavior are assumed by the analyzer during the analysis of a program

4.17
UB
undefined behavior

4.18
unexpected behavior
well-defined behavior that may be unexpected or unanticipated by the programmer; incorrect programming assumptions

4.19
unsigned integer wrapping
computation involving unsigned operands whose result is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type

4.20
untrusted data
data originating from outside of a trust boundary

Note 1 to entry: See [2].

4.21
valid pointer
pointer that refers to an element within an array or one past the last element of an array

Note 1 to entry: For the purposes of this definition, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type. (See ISO/IEC 9899:2011, 6.5.8, paragraph 4.)

Note 2 to entry: For the purposes of this definition, an object can be considered to be an array of a certain number of bytes; that number is the size of the object, as produced by the `sizeof` operator. (See ISO/IEC 9899:2011, 6.3.2.3, paragraph 7.)

4.22
vulnerability
set of conditions that allows an attacker to violate an explicit or implicit security policy

5 Rules

5.1 Accessing an object through a pointer to an incompatible type [ptrcomp]

Rule

Accessing an object through a pointer to an incompatible type (other than `unsigned char`) shall be diagnosed.

Rationale

ISO/IEC 9899:2011, 6.5, paragraph 7, states,

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

— *a type compatible with the effective type of the object,*

— *a qualified version of a type compatible with the effective type of the object,*

— *a type that is the signed or unsigned type corresponding to the effective type of the object,*

— *a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,*

— *an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or*

— *a character type.*

The intent of this list is to specify those circumstances in which an object may or may not be aliased.

According to ISO/IEC 9899:2011, 6.2.6.1, *Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined.*

Accessing an object through a pointer to an incompatible type (other than `unsigned char`) is undefined behavior.

C identifies the following undefined behavior:

UB	Description
37	An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because an object of type `float` is incremented through a pointer to `int`, `ip`.

```
void f(void) {
  if (sizeof(int) == sizeof(float)) {
    float f = 0.0f;
    int *ip = (int *)&f;

    printf("float is %f\n", f);

    (*ip)++;  // diagnostic required

    printf("float is %f\n", f);
  }
}
```

5.2 Accessing freed memory [accfree]

Rule

After an allocated block of dynamic storage has been deallocated by a memory management function, the evaluation of any pointers into the freed memory, including being dereferenced or acting as an operand of an arithmetic operation, type cast, or right-hand side of an assignment, shall be diagnosed.

Rationale

C identifies the situation in which undefined behavior arises as a result of accessing freed memory:

UB	Description
177	The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is used (7.22.3).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because `head->next` is accessed after `head` has been freed.

```
struct List { struct List *next; /* ... */ };

void free_list(struct List *head) {
  for (; head != NULL; head = head->next) {  // diagnostic required
    free(head);
  }
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because `buf` is written to after it has been freed.

```
int main(int argc, char *argv[]) {
  if (argc < 2) {
```



```
    /* ... */
  }

  char *return_val = 0;

  const size_t bufsize = strlen(argv[1]) + 1;

  char *buf = (char *)malloc(bufsize);
  if (!buf) {
    /* ... */
  }
  /* ... */
  free(buf);
  /* ... */
  return_val = strncpy(buf, argv[1], bufsize);  // diagnostic required
  if (return_val) {
    /* ... */
  }
  return EXIT_SUCCESS;
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because `realloc` may free `c_str1` when it returns `NULL`, resulting in `c_str1` being freed twice.

```
void f(char * c_str1, size_t size) {
  char * c_str2 = (char *)realloc(c_str1, size);
  if (c_str2 == NULL) {
    free(c_str1); // diagnostic required
    return;
  }
}
```

5.3 Accessing shared objects in signal handlers [accsig]

Rule

Accessing values of objects that are neither lock-free atomic objects nor of type `volatile sig_atomic_t` in a signal handler shall be diagnosed.

Rationale

C identifies the situation in which undefined behavior arises as a result of accessing a static storage duration object without the correct characteristics:

UB	Description
132	A signal occurs other than as the result of calling the `abort` or `raise` function, and the signal handler refers to an object with static storage duration other than by assigning a value to an object declared as `volatile sig_atomic_t`, or calls any function in the standard library other than the `abort` function, the `_Exit` function, or the `signal` function (for the same signal number) (7.14.1.1).

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the object referred to by the shared pointer `err_msg` is accessed from the signal handler `handler` via the C Standard Library function `strcpy`.

```
#define MAX_MSG_SIZE 24
char *err_msg;

void handler(int signum) {
  if ((strcpy(err_msg, "SIGINT detected.")) == err_msg){ // diagnostic required
    /* ... */
  }
}

int main(void) {

  signal(SIGINT, handler);
```

```
  err_msg = (char *)malloc(MAX_MSG_SIZE);
  if (err_msg == NULL) {
    /* Handle error condition */
  }
  if ((strcpy(err_msg, "No errors yet.")) == err_msg) {
    /* ... */
  }

  /* Main code loop */

  return EXIT_SUCCESS;
}
```

5.4 No assignment in conditional expressions [boolasgn]

Rule

The use of the assignment operator in the following contexts shall be diagnosed:

— `if` (controlling expression)

— `while` (controlling expression)

— `do ... while` (controlling expression)

— `for` (second operand)

— `?:` (first operand)

— `&&` (either operand)

— `||` (either operand)

— comma operator (second operand) when the comma expression is used in any of these contexts

— `?:` (second or third operands) where the ternary expression is used in any of these contexts

Rationale

Mistyping or erroneously using = in Boolean expressions, where == was intended, is a common cause of program error. This rule makes the presumption that any use of = was intended to be == unless the context makes it clear that such is not the case.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the expression `x = y` is used as the controlling expression of the `while` statement.

```
while ( x = y ) { /* ... */ }  //  diagnostic required
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the expression `x = y` is used as the controlling expression of the `while` statement.

```
do { /* ... */ } while ( foo(), x = y ) ; // diagnostic required
```

EXAMPLE 3 In this compliant example, no diagnostic is required because the expression `x = y` is not used as the controlling expression of the `while` statement.

```
do { /* ... */ } while ( x = y, p == q ) ; // no diagnostic required
```

Exceptions

— EX1: Assignment is permitted where the result of the assignment is itself a parameter to a comparison expression (e.g., `x == y` or `x != y`) or relational expression and need not be diagnosed.

EXAMPLE This example shows an acceptable use of this exception.


```
if ( ( x = y )  != 0  ) { /* ... */ }
```

— EX2: Assignment is permitted where the expression consists of a single primary expression.

EXAMPLE 1 This example shows an acceptable use of this exception.

```
if ( ( x = y ) ) { /* ... */ }
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because `&&` is not a comparison or relational operator and the entire expression is not primary.

```
if ( ( v = w ) && flag ) { /* ... */ }  // diagnostic required
```

— EX3: Assignment is permitted in the above contexts where it occurs in a function argument or array index.

EXAMPLE This example shows an acceptable use of this exception.

```
if ( foo( x = y ) ) { /* ... */ }
```

5.5 Calling functions in the C Standard Library other than `abort`, `_Exit`, and `signal` from within a signal handler [asyncsig]

Rule

Calling functions in the C Standard Library other than `abort`, `_Exit`, and `signal` from within a signal handler shall be diagnosed.

Rationale

C identifies the situation in which undefined behavior arises as a result of calling other C library functions:

UB	Description
131	A signal occurs as the result of calling the `abort` or `raise` function, and the signal handler calls the `raise` function (7.14.1.1).
132	A signal occurs other than as the result of calling the `abort` or `raise` function, and the signal handler refers to an object with static storage duration other than by assigning a value to an object declared as `volatile sig_atomic_t`, or calls any function in the standard library other than the `abort` function, the `_Exit` function, or the `signal` function (for the same signal number) (7.14.1.1).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the C Standard Library function `fprintf` is called from the signal handler `handler` via the function `log_message`.

```
#define MAXLINE 1024

char info[MAXLINE];

void log_message(void) {
  fprintf(stderr, "%s\n", info);  // diagnostic required
}

void handler(int signum) {
  log_message();
}

int main(void) {
  if (signal(SIGINT, handler) == SIG_ERR) {
    /* Handle error */
  }
  /* An interactive attention signal might invoke handler() from here on. */
  while (1) {
    /* Main loop program code */
```

```
    log_message();

    /* More program code */
  }
  return EXIT_SUCCESS;
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the C Standard Library function `raise` is called from the signal handler `int_handler`.

```
void term_handler(int signum) {
  /* SIGTERM handling specific */
}

void int_handler(int signum) {
  /* SIGINT handling specific */
  if (raise(SIGTERM) != 0) {  // diagnostic required
    /* Handle error */
  }
}

int main(void) {
  if (signal(SIGTERM, term_handler) == SIG_ERR) {
    /* Handle error */
  }
  if (signal(SIGINT, int_handler) == SIG_ERR) {
    /* Handle error */
  }

  /* Program code */
  if (raise(SIGINT) != 0) {
    /* Handle error */
  }
  /* More code */

  return EXIT_SUCCESS;
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the C Standard Library function `longjmp` is called from the signal handler `handler`.

```
#define MAXLINE 1024

static jmp_buf env;

void handler(int signum) {
  longjmp(env, 1);  // diagnostic required
}

void log_message(char *info1, char *info2) {
  static char *buf = NULL;
  static size_t bufsize;
  char buf0[MAXLINE];

  if (buf == NULL) {
    buf = buf0;
    bufsize = sizeof(buf0);
  }

  /*
   *  Try to fit a message into buf, else re-allocate
   *  it on the heap and then log the message.
   */

/*** VULNERABILITY IF SIGINT RAISED HERE ***/

  if (buf == buf0) {
    buf = NULL;
  }
}
```



```
int main(void) {
  if (signal(SIGINT, handler) == SIG_ERR) {
    /* Handle error */
  }

  char *info1;
  char *info2;

  /* info1 and info2 are set by user input here */

  if (setjmp(env) == 0) {
    while (1) {
      /* Main loop program code */
      log_message(info1, info2);
      /* More program code */
    }
  }
  else {
    log_message(info1, info2);
  }

  return EXIT_SUCCESS;
}
```

5.6 Calling functions with incorrect arguments [argcomp]

Rule

Calling a function with the wrong number or type of arguments shall be diagnosed.

Rationale

C identifies four distinct situations in which undefined behavior may arise as a result of invoking a function using a declaration that is incompatible with its definition or with incorrect types or numbers of arguments:

UB	Description
26	A pointer is used to call a function whose type is not compatible with the pointed-to type (6.3.2.3).
38	For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).
39	For a call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after promotion are not compatible with the types of the parameters (6.5.2.2).
41	A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the C Standard Library function `strchr` is called through the function pointer `fp` with incorrectly typed arguments.

```
char *(*fp)();

void f(void) {
  char *c;
  fp = strchr;
  c = fp(12, 2);  // diagnostic required
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the function `copy` is defined to take two arguments but is called with three arguments.

```
/* in another source file */
void copy(char *dst, const char *src) {
  if (!strcpy(dst, src)) {
```

```
    /* report error */
  }
}



/* in this source file -- no copy prototype in scope */
void copy();

void g(const char *s) {
  char buf[20];
  copy(buf, s, sizeof buf);  // diagnostic required
  /* ... */
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the function `buginf` is defined to take a variable number of arguments but is declared in another file with no prototype and is called.

```
/* in another source file */
void buginf(const char *fmt, ...) {
   /* ... */
}



/* in this source file -- no buginf prototype in scope */
void buginf();

void h(void) {
  buginf("bug in function %s, line %d\n", __func__, __LINE__);  // diagnostic required
  /* ... */
}
```

EXAMPLE 4 In this noncompliant example, a diagnostic is required because the function `f` is defined to take an argument of type `long`, but `f` is called from another file with an argument of type `int`.

```
/* in somefile.c */

long f(long x) {
  return x < 0 ? -x : x;
}



/* in otherfile.c */

int g(int x) {
  return f(x);  // diagnostic required
}
```

5.7 Calling `signal` from interruptible signal handlers [sigcall]

Rule

On systems with nonpersistent signal handlers, calling `signal` from within a signal handler whose execution can be interrupted by receipt of a signal shall be diagnosed.

Rationale

Calling `signal` under these conditions presents a race condition.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required on implementations with nonpersistent signal handlers because the C Standard Library function `signal` is called from the signal handler `handler`.

```
void handler(int signum) {
  if (signal(signum, handler) == SIG_ERR) {  // diagnostic required
```



```
    /* ... */
  }

  /* ... */
}

void f(void) {
  if (signal(SIGUSR1, handler) == SIG_ERR) {
    /* ... */
  }

  /* ... */
}
```

5.8 Calling `system` [syscall]

Rule

All calls to the `system` function shall be diagnosed.

Rationale

Use of the `system` function can result in exploitable vulnerabilities

— when passing an unsanitized or improperly sanitized command string originating from a tainted source, or

— if a command is specified without a path name and the command processor path name resolution mechanism is accessible to an attacker, or

— if a relative path to an executable is specified and control over the current working directory is accessible to an attacker, or

— if the specified executable program can be spoofed by an attacker.

Although exceptions to this rule are necessary, they can only be identified on a case-by-case basis during a code review and are consequently outside the scope of this rule.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because a string consisting of `any_cmd` and the tainted value stored in `input` is copied into `cmdbuf` and then passed as an argument to the `system` function to execute.

```
void f(char *input) {
  char cmdbuf[512];
  int len_wanted = snprintf(
    cmdbuf, sizeof(cmdbuf), "any_cmd '%s'", input
  );

  if (len_wanted >= sizeof(cmdbuf)) {
    perror("Input too long");
  } else if (len_wanted < 0) {
    perror("Encoding error");
  } else if (system(cmdbuf) == -1) {   // diagnostic required
    perror("Error executing input");
  }
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because `system` is used to remove the `.config` file in the user's home directory.

```
void g(void) {
  system("rm ~/.config");   // diagnostic required
}
```


5.9 Comparison of padding data [padcomp]

Rule

Comparison of padding data shall be diagnosed.

Rationale

The value of padding bits is unspecified and may contain data initially provided by an attacker.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the C Standard Library function `memcmp` is used to compare the structures `s1` and `s2`, including padding data.

```
struct my_buf {
  char buff_type;
  size_t size;
  char buffer[50];
};

unsigned int buf_compare(
  const struct my_buf *s1,
  const struct my_buf *s2)
{
  if (!memcmp(s1, s2, sizeof(struct my_buf))) {  // diagnostic required
    /* ... */
  }

  return 0;
}
```

5.10 Converting a pointer to integer or integer to pointer [intptrconv]

Rule

Converting an integer type to a pointer type shall be diagnosed if the resulting pointer is incorrectly aligned, does not point to an entity of the referenced type, or is a trap representation.

Converting a pointer type to an integer type shall be diagnosed if the result cannot be represented in the integer type.

Rationale

C identifies the situation in which undefined behavior arises as a result of converting a pointer to an integer type:

UB	Description
24	Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required on an implementation where pointers are 64 bits and unsigned integers are 32 bits because the result of converting the 64-bit `ptr` cannot be represented in the 32-bit integer type.

```
void f(void) {
  char *ptr;
  /* ... */
  unsigned int number = (unsigned int)ptr;  // diagnostic required
  /* ... */
}
```


EXAMPLE 2 In this noncompliant example, a diagnostic is required because the conversion of the integer literal 0xdeadbeef to a pointer that is not known to point to an entity of the referenced type.

```
unsigned int *g(void) {
  unsigned int *ptr = (unsigned int *)0xdeadbeef;  // diagnostic required
  /* ... */
  return ptr;
}
```

Exceptions

— EX1: A null pointer can be converted to an integer; it takes on the value 0. Likewise, a 0 integer can be converted to a pointer; it becomes the null pointer.

— EX2: Any valid pointer (4.21) to void can be converted to intptr_t or uintptr_t and back with no change in value. (This includes the underlying types if intptr_t and uintptr_t are typedefs and any typedefs that denote the same types as intptr_t and uintptr_t.)

EXAMPLE

```
void h(void) {
  intptr_t i = (intptr_t)(void *)&i;
  uintptr_t j = (uintptr_t)(void *)&j;

  void *ip = (void *)i;
  void *jp = (void *)j;

  assert(ip == &i);
  assert(jp == &j);
}
```

5.11 Converting pointer values to more strictly aligned pointer types [alignconv]

Rule

Converting a pointer value to a pointer type that is more strictly aligned than the type the value actually points to shall be diagnosed.

Rationale

Converting a pointer value to a pointer type that is more strictly aligned than the type the value actually points to results in undefined behavior if the actual value is unaligned with respect to the destination type.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the char pointer &c is converted to the more strictly aligned int pointer i_ptr.

```
void f(void) {
  int *i_ptr;
  char c;

  i_ptr = (int *)&c;  // diagnostic required
  /* ... */
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because the value referenced by the char pointer c_ptr has the alignment of type int.

```
void f(void) {
  char *c_ptr;
  int *i_ptr;
  int i;

  c_ptr = (char *)&i;
  i_ptr = (int *)c_ptr;
  /* ... */
```

```
}
```

5.12 Copying a `FILE` object [filecpy]

Rule

Copying a `FILE` object shall be diagnosed.

Rationale

According to ISO/IEC 9899:2011, 7.21.3, paragraph 6,

> *The address of the `FILE` object used to control a stream may be significant; a copy of a `FILE` object need not serve in place of the original.*

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the `FILE` object `stdout` is copied.

```
int main(void) {
  FILE my_stdout = *(stdout);  // diagnostic required
  if (fputs("Hello, World!\n", &my_stdout) == EOF) {
    /* ... */
  }
  return EXIT_SUCCESS;
}
```

5.13 Declaring the same function or object in incompatible ways [funcdecl]

Rule

Two or more incompatible declarations of the same function or object that appear in the same program shall be diagnosed.

Rationale

C identifies three distinct situations in which undefined behavior may arise as a result of incompatible declarations of the same function or object:

UB	Description
15	Two declarations of the same object or function specify types that are not compatible (6.2.7).
37	An object has its stored value accessed other than by an lvalue of an allowable type (6.5).
41	A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

While the effects of two incompatible declarations simply appearing in the same program may be benign on most implementations, the effects of invoking a function through an expression whose type is incompatible with the function definition are typically catastrophic. Similarly, the effects of accessing an object using an lvalue of a type that is incompatible with the object definition may range from unintended information exposure to memory overwrite to a hardware trap.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the variable `i` has two incompatible declarations.

```
/* in a.c */
extern int i;  // diagnostic required

int f(void) {
  return ++i;
}
```



```
/* in b.c */
short i;  // diagnostic required
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the variable a has two incompatible declarations.

```
/* in a.c */
extern int *a;  // diagnostic required

int g(unsigned i, int x) {
  int tmp = a[i];
  a[i] = x;
  return tmp;
}

/* in b.c */
int a[] = { 1, 2, 3, 4 };  // diagnostic required
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the function h has two incompatible declarations.

```
/* in a.c */
extern int h(int a);  // diagnostic required

int main(void) {
  printf("%d", h(10));
  return EXIT_SUCCESS;
}

/* in b.c */
long h(long a) {  // diagnostic required
  return a * 2;
}
```

EXAMPLE 4 According to ISO/IEC 9899:2011, 5.2.4.1, external identifiers need to be unique only within the first 31 characters. In this noncompliant example, a diagnostic is required on implementations where the external identifiers bash_groupname_completion_function and bash_groupname_completion_funct are identical, because it results in incompatible declarations.

```
/* in bash/bashline.h */
extern char* bash_groupname_completion_function(const char *, int);
// diagnostic required
// the identifier exceeds 31 characters
/* in a.c */
#include <bashline.h>

void w(const char *s, int i) {
  bash_groupname_completion_function(s, i);
}

/* in b.c */
int bash_groupname_completion_funct;  // diagnostic required
// identifier not unique within 31 characters
```

NOTE The identifier bash_groupname_completion_function referenced here was taken from GNU Bash version 3.2.

Exception

No diagnostic need be issued if a declaration that is incompatible with the definition occurs in a translation unit that does not contain any definition or uses of the function or object other than additional declarations, if any.

EXAMPLE

```
/* a.c: */
int x = 0; /* the definition */

/* b.c: */
```

```
extern char x; /* incompatible declaration */
/* but no other references to 'x' */
```

5.14 Dereferencing an out-of-domain pointer [nullref]

Rule

Dereferencing a tainted or out-of-domain pointer shall be diagnosed.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because if `malloc` returns `NULL`, then the call to `memcpy` will dereference the null pointer `c_str`.

```
void f(const char *input_str) {
  size_t size = strlen(input_str) + 1;
  char *c_str = (char *)malloc(size);
  if ((memcpy(c_str, input_str, size)) == c_str) {  // diagnostic required
    /* ... */
  }
  /* ... */
  free(c_str);
  c_str = NULL;
}
```

5.15 Escaping of the address of an automatic object [addrescape]

Rule

The address of an object with automatic storage duration returned from a function or held in any pointer variable whose lifetime extends past the lifetime of the referenced object at the time the automatic object goes out of scope shall be diagnosed.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the address of the automatic object `c_str` remains in the pointer variable `p` when `c_str` goes out of scope in the function `dont_do_this`.

```
const char *p;
void dont_do_this(void) {
  const char c_str[] = "This will change";
  p = c_str;   // diagnostic required
}

void innocuous(void) {
  const char c_str[] = "Surprise, surprise";
  puts(c_str);
}

int main(void) {
  dont_do_this();
  innocuous();
  puts(p);

  return EXIT_SUCCESS;
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the address of the automatic object `array` is returned.

```
int *init_array(void) {
  int array[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  return array;  // diagnostic required
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the address of the automatic object `fmt` remains in the pointer variable `ptr_param` when `fmt` goes out of scope in the function `squirrel_away`.


```
void squirrel_away(char **ptr_param) {
  char fmt[] = "Error: %s\n";

  /* ... */
  *ptr_param = fmt;  // diagnostic required
}

int main(void) {
  char *ptr;
  squirrel_away(&ptr);

  /* ... */
  return EXIT_SUCCESS;
}
```

5.16 Conversion of signed characters to wider integer types before a check for `EOF` [signconv]

Rule

Converting a tainted value of type `char` or `signed char` to a larger integer type without having first cast the value to `unsigned char` shall be diagnosed if the value is subsequently compared with the value of `EOF`.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the character of type `char` pointed to by `c_str` is converted to `int` without first being cast to `unsigned char`.

```
int yy_string_get(char *c_str) {
  int c = EOF;

  if (c_str && *c_str) {
    c = *c_str++;  // if char is signed, a 0xFF char can be confused with EOF
  }
  return c;
}

/* ... */

char string[BUFSIZ];
GET_TAINTED_STRING(string, BUFSIZ);
if (yy_string_get(string) == EOF) // diagnostic required
```

5.17 Use of an implied default in a `switch` statement [swtchdflt]

Rule

A `switch` statement with a controlling expression of enumerated type that does not include a default case and does not include cases for all enumeration constants of that type shall be diagnosed.

Rationale

A `switch` statement with a controlling expression of enumerated type that does not include a default case and does not include cases for all enumeration constants of that type indicates logical incompleteness.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because not all possible values of `widget_type` are checked for in the `switch` statement.

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z };

void f(enum WidgetEnum widget_type) {
  switch (widget_type) {  // diagnostic required
    case WE_X:
      /* ... */
```



```
      break;
    case WE_Y:
      /* ... */
      break;
    case WE_Z:
      /* ... */
      break;
  }
}
```

5.18 Failing to close files or free dynamic memory when they are no longer needed [fileclose]

Rule

A call to the `fopen` or `freopen` function shall be diagnosed after the lifetime of the last pointer object that stores the return value of the call has ended without a call to `fclose` with that pointer value.

A call to a standard memory allocation function shall be diagnosed after the lifetime of the last pointer object that stores the return value of the call has ended without a call to a standard memory deallocation function with that pointer value.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the resource allocated by the call to `fopen` is not closed.

```
int f(void) {
  const char *filename = "secure.dat";

  FILE *f = fopen(filename, "r");  // diagnostic required
  if (f == NULL) {
    /* ... */
  }

  /* ... */
  return 0;
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the resource allocated by the call to `malloc` is not freed.

```
int f(void) {
  char *text_buffer = (char *)malloc(BUFSIZ); // diagnostic required

  if (text_buffer == NULL) {
    return -1;
  }
  return 0;
}
```

5.19 Failing to detect and handle standard library errors [liberr]

Rule

Failure to branch conditionally on detection or absence of a standard library error condition shall be diagnosed.

The successful completion or failure of each of the standard library functions listed in Table 2 shall be determined either by comparing the function's return value with the value listed in the column labeled "Error Return" or by calling one of the library functions mentioned in the footnotes to the same column.

Table 2 — Library functions and returns

Function	Successful return	Error return
aligned_alloc	pointer to space	NULL
asctime_s	zero	nonzero
at_quick_exit	zero	nonzero
atexit	zero	nonzero
bsearch	pointer to matching element	NULL
bsearch_s	pointer to matching element	NULL
btowc	converted wide character	WEOF
c16rtomb	number of bytes	(size_t)(-1)
c32rtomb	number of bytes	(size_t)(-1)
calloc	pointer to space	NULL
clock	processor time	(clock_t)(-1)
cnd_broadcast	thrd_success	thrd_error
cnd_init	thrd_success	thrd_nomem or thrd_error
cnd_signal	thrd_success	thrd_error
cnd_timedwait	thrd_success	thrd_timedout or thrd_error
cnd_wait	thrd_success	thrd_error
ctime_s	zero	nonzero
fclose	zero	EOF (negative)
fflush	zero	EOF (negative)
fgetc	character read	EOF[a]
fgetpos	zero	nonzero
fgets	pointer to string	NULL
fgetwc	wide character read	WEOF[a]
fopen	pointer to stream	NULL
fopen_s	zero	nonzero
fprintf	number of characters (nonnegative)	negative
fprintf_s	number of characters (nonnegative)	negative
fputc	character written	EOF[b]
fputs	nonnegative	EOF (negative)
fputws	nonnegative	EOF (negative)
fread	elements read	elements read
freopen	pointer to stream	NULL
freopen_s	zero	nonzero
fscanf	number of conversions (nonnegative)	EOF (negative)
fscanf_s	number of conversions (nonnegative)	EOF (negative)
fseek	zero	nonzero
fsetpos	zero	nonzero
ftell	file position	–1L
fwprintf	number of wide characters (nonnegative)	negative
fwprintf_s	number of wide characters (nonnegative)	negative
fwrite	elements written	elements written

Table 2 *(continued)*

Function	Successful return	Error return
fwscanf	number of conversions (nonnegative)	EOF (negative)
fwscanf_s	number of conversions (nonnegative)	EOF (negative)
getc	character read	EOF[a]
getchar	character read	EOF[a]
getenv	pointer to string	NULL
getenv_s	pointer to string	NULL
gets_s	pointer to string	NULL
getwc	wide character read	WEOF
getwchar	wide character read	WEOF
gmtime	pointer to broken-down time	NULL
gmtime_s	pointer to broken-down time	NULL
localtime	pointer to broken-down time	NULL
localtime_s	pointer to broken-down time	NULL
malloc	pointer to space	NULL
mblen, s != NULL	number of bytes	−1
mbrlen, s != NULL	number of bytes or status	(size_t)(-1)
mbrtoc16	number of bytes or status	(size_t)(-1),errno == EILSEQ
mbrtoc32	number of bytes or status	(size_t)(-1),errno == EILSEQ
mbrtowc, s != NULL	number of bytes or status	(size_t)(-1),errno == EILSEQ
mbsrtowcs	number of non-null elements	(size_t)(-1),errno == EILSEQ
mbsrtowcs_s	zero	nonzero
mbstowcs	number of non-null elements	(size_t)(-1)
mbstowcs_s	zero	nonzero
mbtowc, s != NULL	number of bytes	−1
memchr	pointer to located character	NULL
mktime	calendar time	(time_t)(-1)
mtx_init	thrd_success	thrd_error
mtx_lock	thrd_success	thrd_error
mtx_timedlock	thrd_success	thrd_timedout or thrd_error
mtx_trylock	thrd_success	thrd_busy or thrd_error
mtx_unlock	thrd_success	thrd_error
printf_s	number of characters (nonnegative)	negative
putc	character written	EOF[b]
putwc	wide character written	WEOF
raise	zero	nonzero
realloc	pointer to space	NULL
remove	zero	nonzero
rename	zero	nonzero
setlocale	pointer to string	NULL
setvbuf	zero	nonzero
scanf	number of conversions (nonnegative)	EOF (negative)

Table 2 *(continued)*

Function	Successful return	Error return
scanf_s	number of conversions (nonnegative)	EOF (negative)
signal	pointer to previous function	SIG_ERR, errno > 0
snprintf	number of characters that would be written (nonnegative)	negative
snprintf_s	number of characters that would be written (nonnegative)	negative
sprintf	number of non-null characters written	negative
sprintf_s	number of non-null characters written	negative
sscanf	number of conversions (nonnegative)	EOF (negative)
sscanf_s	number of conversions (nonnegative)	EOF (negative)
strchr	pointer to located character	NULL
strerror_s	zero	nonzero
strftime	number of non-null characters	zero
strpbrk	pointer to located character	NULL
strrchr	pointer to located character	NULL
strstr	pointer to located string	NULL
strtod	converted value	zero, errno == ERANGE
strtof	converted value	zero, errno == ERANGE
strtoimax	converted value	INTMAX_MAX or INTMAX_MIN, errno == ERANGE
strtok	pointer to first character of a token	NULL
strtok_s	pointer to first character of a token	NULL
strtol	converted value	LONG_MAX or LONG_MIN, errno == ERANGE
strtold	converted value	zero, errno == ERANGE
strtoll	converted value	LLONG_MAX or LLONG_MIN, errno == ERANGE
strtoumax	converted value	UINTMAX_MAX, errno == ERANGE
strtoul	converted value	ULONG_MAX, errno == ERANGE
strtoull	converted value	ULLONG_MAX, errno == ERANGE
strxfrm	length of transformed string	>= n
swprintf	number of non-null wide characters	negative
swprintf_s	number of non-null wide characters	negative
swscanf	number of conversions (nonnegative)	EOF (negative)
swscanf_s	number of conversions (nonnegative)	EOF (negative)
thrd_create	thrd_success	thrd_nomem or thrd_error
thrd_detach	thrd_success	thrd_error
thrd_join	thrd_success	thrd_error
thrd_sleep	zero	negative
time	calendar time	(time_t)(-1)
timespec_get	base	zero
tmpfile	pointer to stream	NULL

Table 2 *(continued)*

Function	Successful return	Error return
tmpfile_s	zero	nonzero
tmpnam	non-null pointer	NULL
tmpnam_s	zero	nonzero
tss_create	thrd_success	thrd_error
tss_get	value of thread-specific storage	zero
tss_set	thrd_success	thrd_error
ungetc	character pushed back	EOF (negative; see below)
ungetwc	character pushed back	WEOF (negative)
vfprintf	number of characters (nonnegative)	negative
vfprintf_s	number of characters (nonnegative)	negative
vfscanf	number of conversions (nonnegative)	EOF (negative)
vfscanf_s	number of conversions (nonnegative)	EOF (negative)
vfwprintf	number of wide characters (nonnegative)	negative
vfwprintf_s	number of wide characters (nonnegative)	negative
vfwscanf	number of conversions (nonnegative)	EOF (negative)
vfwscanf_s	number of conversions (nonnegative)	EOF (negative)
vprintf_s	number of characters (nonnegative)	negative
vscanf	number of conversions (nonnegative)	EOF (negative)
vscanf_s	number of conversions (nonnegative)	EOF (negative)
vsnprintf	number of characters that would be written (nonnegative)	negative
vsnprintf_s	number of characters that would be written (nonnegative)	negative
vsprintf	number of non-null characters (nonnegative)	negative
vsprintf_s	number of non-null characters (nonnegative)	negative
vsscanf	number of conversions (nonnegative)	EOF (negative)
vsscanf_s	number of conversions (nonnegative)	EOF (negative)
vswprintf	number of non-null wide characters	negative
vswprintf_s	number of non-null wide characters	negative
vswscanf	number of conversions (nonnegative)	EOF (negative)
vswscanf_s	number of conversions (nonnegative)	EOF (negative)
vwprintf_s	number of wide characters (nonnegative)	negative
vwscanf	number of conversions (nonnegative)	EOF (negative)
vwscanf_s	number of conversions (nonnegative)	EOF (negative)
wcrtomb	number of bytes stored	(size_t)(-1)
wcschr	pointer to located wide character	NULL
wcsftime	number of non-null wide characters	zero
wcspbrk	pointer to located wide character	NULL
wcsrchr	pointer to located wide character	NULL
wcsrtombs	number of non-null bytes	(size_t)(-1),errno == EILSEQ

Table 2 *(continued)*

Function	**Successful return**	**Error return**
wcsrtombs_s	zero	nonzero
wcsstr	pointer to located wide string	NULL
wcstod	converted value	zero, errno == ERANGE
wcstof	converted value	zero, errno == ERANGE
wcstoimax	converted value	INTMAX_MAX or INTMAX_MIN, errno == ERANGE
wcstok	pointer to first wide character of a token	NULL
wcstok_s	pointer to first wide character of a token	NULL
wcstol	converted value	LONG_MAX or LONG_MIN, errno == ERANGE
wcstold	converted value	zero, errno == ERANGE
wcstoll	converted value	LLONG_MAX or LLONG_MIN, errno == ERANGE
wcstombs	number of non-null bytes	(size_t)(-1)
wcstombs_s	zero	nonzero
wcstoumax	converted value	UINTMAX_MAX, errno == ERANGE
wcstoul	converted value	ULONG_MAX, errno == ERANGE
wcstoull	converted value	ULLONG_MAX, errno == ERANGE
wcsxfrm	length of transformed wide string	>= n
wctob	converted character	EOF
wctomb, s != NULL	number of bytes stored	–1
wctomb_s, s != NULL	number of bytes stored	–1
wctrans	valid argument to towctrans	zero
wctype	valid argument to iswctype	zero
wmemchr	pointer to located wide character	NULL
wprintf_s	number of wide characters (nonnegative)	negative
wscanf	number of conversions (nonnegative)	EOF (negative)
wscanf_s	number of conversions (nonnegative)	EOF (negative)

[a] Use feof and ferror.
[b] Use ferror.

The ungetc function does not set the error indicator, even when it fails, so it is not possible to check for errors reliably unless it is known that the argument is not equal to EOF. C states that "one character of pushback is guaranteed," so this should not be an issue if, at most, one character is ever pushed back before reading again.

NOTE A cumulative error check satisfies the rule as long as undefined behavior is not triggered (for example, by using the contents of the fgets or fgetws array or using the file position indicator after fread or fwrite without first checking for error).

Rationale

Failure to branch conditionally on detection or absence of a standard library error condition can result in unexpected behavior.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the return value of fseek is not checked for an error condition.

```
void test_unchecked_return(FILE *file, long offset) {
  fseek(file, offset, SEEK_SET);  // diagnostic required
}
```

NOTE Return values from the following functions (Table 3) do not need to be checked because their historical use has overwhelmingly omitted error checking, and the consequences are not relevant to security.

Table 3 — Example library functions and returns

Function	Successful return	Error return
`printf`	number of characters (nonnegative)	negative
`putchar`	character written	`EOF`
`puts`	nonnegative	`EOF` (negative)
`putwchar`	wide character written	`WEOF`
`vprintf`	number of characters (nonnegative)	negative
`vwprintf`	number of wide characters (nonnegative)	negative
`wprintf`	number of wide characters (nonnegative)	negative

Exceptions

— EX1: The use of a `void` cast to signify programmer intent to ignore a return value from a function need not be diagnosed.

EXAMPLE This example shows an acceptable use of this exception.

```
void foo(FILE *file) {
  (void)fputs("foo", file);
  /* ... */
}
```

— EX2: Ignoring the return value of a function that cannot fail or whose return value cannot signify an error condition need not be diagnosed. For example, `strcpy` is one such function.

5.20 Forming invalid pointers by library function [libptr]

Rule

Invoking a C library function with an argument or arguments that causes the function to form a pointer that does not point into or just past the end of the object shall be diagnosed.

Rationale

Many C Standard Library functions manipulate individual objects or arrays of objects either one element at a time or one byte at a time. With a few exceptions, such functions typically take at least two arguments for each object (or array) they manipulate:

— a valid pointer into the object or storage for an object and

— an integer argument indicating how many elements or bytes of the object to manipulate.

C identifies the following undefined behavior:

UB	Description
109	The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

5.20.1 Library functions that take a pointer and an integer

The following standard library functions take a pointer argument and a size argument, with the constraint that the pointer must point to a valid memory object of at least the number of bytes or wide characters (as appropriate) indicated by the size argument.

`fgets`	`fread`	`fwrite`	`mblen`
`memchr`	`memset`	`fgetws`	`wmemchr`
`wmemset`	`mbrlen`	`tmpnam_s`	`gets_s`
`getenv_s`	`memset_s`	`strerror_s`	`strnlen_s`
`asctime_s`	`ctime_s`	`wcscpy_s`	`wcsncpy_s`
`wmemcpy_s`	`wmemmove_s`	`wcscat_s`	`wcsncat_s`
`wcsnlen_s`			

5.20.2 Library functions that take two pointers and an integer

The following standard library functions take two pointer arguments and a size argument, with the constraint that both pointers must point to valid memory objects of at least the number of bytes or wide characters as appropriate, indicated by the size argument.

`mbtowc`	`wctomb`	`mbtowcs`	`wcstombs`
`memcpy`	`memmove`	`strncpy`	`strncat`
`memcmp`	`strncmp`	`strxfrm`	`mbrtoc16`
`mbrtoc32`	`wcsncpy`	`wmemcpy`	`wmemmove`
`wcsncat`	`wcsncmp`	`wcsxfrm`	`wmemcmp`
`mbrtowc`	`wcrtomb`	`mbsrtowcs`	`wcsrtombs`
`wctomb_s`	`mbtowcs_s`	`wcstombs_s`	`memcpy_s`
`memmove_s`	`strcpy_s`	`strncpy_s`	`strcat_s`
`strncat_s`	`wcscpy_s`	`wcsncpy_s`	`wmemcpy_s`
`wmemmove_s`	`wcscat_s`	`wcsncat_s`	`wcrtomb_s`
`mbsrtowcs_s`	`wcsrtombs_s`		

5.20.3 Library functions that take a pointer and two integers

The following standard library functions take a pointer argument and two size arguments, with the constraint that the pointer must point to a valid memory object containing at least as many bytes as the product of the two size arguments.

`bsearch`

`qsort`

`bsearch_s`

`qsort_s`

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the size argument used in the `memset()` call is one longer than the size allocated for `p`.

```
void f1(size_t nchars) {
  char *p = (char *)malloc(nchars);
  const size_t n = nchars + 1;
  if (p) {
    memset(p, 0, n); // diagnostic required
    /* ... */
  }
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the size argument used in the `memset()` call is possibly miscalculated. There is no requirement that the size of an `int` is the same as the size of a `float`.

```
void f2(void) {
  float a[4];
  const size_t n = sizeof(int) * 4;
  void *p = a;

  memset(p, 0, n);  // diagnostic required

  /* ... */
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the size argument used in the `memcpy()` call is possibly miscalculated. The size of an `int` does not need to be smaller than that of a `double`.

```
void f3(int *a) {
  double b = 3.14;
  const size_t n = sizeof(*a);
  void *p = a;
  void *q = &b;

  if ((memcpy(p, q, n)) == p) {  // diagnostic required
    /* ... */
  }
  /* ... */
}
```

EXAMPLE 4 In this noncompliant example, a diagnostic is required because the value of `n` is not computed correctly, allowing a possible write past the end of the object referenced by `p`.

```
void f4(char p[], const char *q) {
  const size_t n = sizeof(p);
  if ((memcpy(p, q, n)) == p) {  // diagnostic required
    /* ... */
  }

  /* ... */
}
```


5.21 Allocating insufficient memory [insufmem]

Rule

A call to a standard memory allocation function is *presumed to be intended for type* `T *` when it appears in any of the following contexts.

— in the right operand of an assignment to an object of type `T *`, or

— in an initializer for an object of type `T *`, or

— in an expression that is passed as an argument of type `T *`, or

— in the expression of a `return` statement for a function returning type `T *`.

A call to a standard memory allocation function taking a size integer argument `n` and presumed to be intended for type `T *` shall be diagnosed when `n < sizeof(T)`.

The following are the standard memory allocation functions that take a size integer argument and return a pointer.

`aligned_alloc`

`calloc`

`malloc`

`realloc`

Rationale

Returning insufficient memory from a memory allocation function is likely to result in undefined behavior when that memory is accessed.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the value of `n` that is used in the `malloc()` call has been possibly miscalculated.

```
wchar_t *f1(void) {
  const wchar_t *p = L"Hello, World!";
  const size_t n = sizeof(p) * (wcslen(p) + 1);
  wchar_t *q = (wchar_t *)malloc(n);  // diagnostic required

  /* ... */
  return q;
}
```

5.22 Forming or using out-of-bounds pointers or array subscripts [invptr]

Rule

Using pointer arithmetic so that the result does not point into or just past the end of the same object, using invalid pointers in arithmetic expressions, or dereferencing pointers that do not point to a valid object shall be diagnosed.

Likewise, using an array subscript so that the resulting reference does not refer to an element in the array also shall be diagnosed.

Rationale

C identifies five distinct situations in which undefined behavior may arise as a result of invalid pointer operations:

UB	Description
46	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).
47	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).
49	An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression a[1][7] given the declaration int a[4][5]) (6.5.6).

UB	Description
62	An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).
109	The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because if `f` is called with a negative argument for `index`, an out-of-bounds pointer is formed.

```
#define TABLESIZE 100

static int table[TABLESIZE];

int *f(int index) {
  if (index < TABLESIZE) {
    return table + index;  // diagnostic required
  }

  return NULL;
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because when the parameter `index` is negative, an out-of-bounds pointer cannot be returned.

```
#define TABLESIZE 100

static int table[TABLESIZE];

int *f(int index) {
  if (0 <= index && index < TABLESIZE) {
    return table + index;
  }

  return NULL;
}
```

EXAMPLE 3 In this compliant example, a diagnostic is not required because the parameter `index` cannot be negative and an out-of-bounds pointer cannot be returned.

```
#define TABLESIZE 100

static int table[TABLESIZE];

int *f(size_t index) {
  if (index < TABLESIZE) {
    return table + index;
  }

  return NULL;
}
```

EXAMPLE 4 In this noncompliant example, a diagnostic is required because if the string `path` does not contain the backslash character in the first `MAX_MACHINE_NAME_LENGTH + 1` characters, then `machine_name` will be dereferenced past the end pointer.

```
#define MAX_MACHINE_NAME_LENGTH 64

char *get_machine_name(const char *path) {
  char *machine_name = (char *)malloc(MAX_MACHINE_NAME_LENGTH + 1);
  if (machine_name == NULL) {
    return NULL;
  }

  while (*path != '\\') {
    *machine_name++ = *path++;  // diagnostic required
  }
```



```
  *machine_name = '\0';

  return machine_name;
}
```

EXAMPLE 5 In this compliant example, a diagnostic is not required because the string `path` is guaranteed to contain a backslash character within the first `MAX_MACHINE_NAME_LENGTH` characters when the string is copied to `machine_name`.

```
#define MAX_MACHINE_NAME_LENGTH 64

char *get_machine_name(const char *path) {
  const char *machine_name_end = strchr(path, '\\');
  if (machine_name_end == NULL
    || machine_name_end >= path + MAX_MACHINE_NAME_LENGTH) {
    return NULL;
  }

  char *machine_name = (char *)malloc(MAX_MACHINE_NAME_LENGTH + 1);
  if (machine_name == NULL) {
    return NULL;
  }

  const char *p = machine_name;

  while (path != p) {
    *p++ = *path++;
  }

  *p = '\0';

  return machine_name;
}
```

EXAMPLE 6 In this noncompliant example, a diagnostic is required because a value is stored beyond the end of the array `table` when the parameter `pos` equals the variable `size`.

```
static int *table = NULL;
static size_t size = 0;

int insert_in_table(size_t pos, int value) {
  if (pos > size) {
    int *tmp = (int *)realloc(table, sizeof(table[0]) * (pos + 1));
    if (tmp == NULL) {
      /* ... */
    }

    size = pos + 1;
    table = tmp;
  }

  table[pos] = value;  // diagnostic required
  return 0;
}
```

EXAMPLE 7 In this compliant example, a diagnostic is not required because a value is stored within the bounds of the array `table` when the parameter `pos` equals the variable `size`.

```
static int *table = NULL;
static size_t size = 0;

int insert_in_table(size_t pos, int value) {
  if (pos >= size) {
    int *tmp = (int *)realloc(table, sizeof(table[0]) * (pos + 1));
    if (tmp == NULL) {
      /* ... */
    }

    size  = pos + 1;
    table = tmp;
```

```
  }

  table[pos] = value;
  return 0;
}
```

EXAMPLE 8 In this noncompliant example, a diagnostic is required because a value is stored beyond the end of the arrays in `matrix[0]` through `matrix[4]` when `j` has values greater than 4.

```
enum { COLS = 5, ROWS = 7 };
static int matrix[ROWS][COLS];

void init_matrix(int x) {
  for (size_t i = 0; i != COLS; ++i) {
    for (size_t j = 0; j != ROWS; ++j) {
      matrix[i][j] = x;  // diagnostic required
    }
  }
}
```

EXAMPLE 9 In this compliant example, a diagnostic is not required because all values are stored within the bounds of the arrays in `matrix[0]` through `matrix[4]`.

```
enum { COLS = 5, ROWS = 7 };
static int matrix[ROWS][COLS];

void init_matrix(int x) {
  for (size_t i = 0; i != ROWS; ++i) {
    for (size_t j = 0; j != COLS; ++j) {
      matrix[i][j] = x;
    }
  }
}
```

EXAMPLE 10 In this noncompliant example, a diagnostic is required because the expression `first++` results in a pointer beyond the end of the array `buf` when `buf` contains no elements.

```
struct S {
  size_t len;
  char buf[];
};

char *find(struct S *s, int c) {
  char *first = s->buf;
  char *last = s->buf + s->len;

  while (first++ != last) {  // diagnostic required
    if (*first == (unsigned char)c) {
      return first;
    }
  }

  return NULL;
}

void g(void) {
  struct S *s = (struct S *)malloc(sizeof(struct S));
  if (s != NULL) {
    s->len = 0;
    /* ... */
    char *where = find(s, '.');
    if (where == NULL) {
      return;
    }
  }

  /* ... */
}
```


EXAMPLE 11 In this compliant example, a diagnostic is not required because the expression `first++` does not occur unless `buf` contains elements.

```
struct S {
  size_t len;
  char buf[];
};

char *find(struct S *s, int c) {
  char *first = s->buf;
  char *last = s->buf + s->len;

  while (first != last) {
    if (*first++ == (unsigned char)c) {
      return first;
    }
  }

  return NULL;
}

void g(void) {
  struct S *s = (struct S *)malloc(sizeof(struct S));
  if (s) {
    s->len = 0;
    /* ... */
    char *where = find(s, '.');
    if (where == NULL) {
      return;
    }
  }

  /* ... */
}
```

EXAMPLE 12 In this noncompliant example, a diagnostic is required because the expression `buf[strlen(buf) - 1]` assumes that the first byte of the parameter to `fgets`, `buf`, is non-null.

```
void f(void) {
  char buf[BUFSIZ];

  if (fgets(buf, sizeof(buf), stdin)) {
    buf[strlen(buf) - 1] = '\0';   // diagnostic required
    puts(buf);
  }
}
```

EXAMPLE 13 In this noncompliant example, a diagnostic is required because the integer `skip` is scaled when added to the pointer `s` and may point outside the bounds of the object referenced by `s`.

```
struct big {
  unsigned long long ull_1;
  unsigned long long ull_2;
  unsigned long long ull_3;
  int si_4;
  int si_5;
};

void g(void) {
  size_t skip = offsetof(struct big, ull_2);
  struct big *s = (struct big *)malloc(4 * sizeof(struct big));
  if (!s) {
    /* ... */
  }

  memset(s + skip, 0, sizeof(struct big) - skip);   // diagnostic required

  /* ... */
}
```

5.23 Freeing memory multiple times [dblfree]

Rule

Freeing memory multiple times shall be diagnosed.

Rationale

Freeing memory multiple times results in *double-free* vulnerabilities.

C identifies the following undefined behavior:

UB	Description
179	The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to `free` or `realloc` (7.22.3.3, 7.22.3.5).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because `x` could be freed twice depending on the value of `error_condition`.

```
void f(size_t num_elem) {
  int error_condition = 0;

  int *x = (int *)malloc(num_elem * sizeof(int));
  if (x == NULL) {
    /* ... */
  }
  /* ... */
  if (error_condition == 1) {
    /* ... */
    free(x);
  }
  /* ... */
  free(x);  // diagnostic required
  x = NULL;
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because `realloc` may free `c_str1` when it returns `NULL`, resulting in `c_str1` being freed twice.

```
void g(char *c_str1, size_t size) {
  char *c_str2 = (char *)realloc(c_str1, size);
  if (c_str2 == NULL) {
    free(c_str1);  // diagnostic required
    return;
  }
}
```

According to ISO/IEC 9899:2011, 7.22.3, paragraph 1,

> *If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.*

And according to ISO/IEC 9899:2011, 7.22.3.5, paragraph 3,

> *If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.*

If `realloc` is called with `size` equal to 0, then if a `NULL` pointer is returned, the old value should be unchanged. However, there are some common but nonconforming implementations that free the pointer, which means that calling `free` on the original pointer might result in a double-free vulnerability. However, not calling `free` on the original pointer might result in a memory leak.

Exception

Some library implementations accept and ignore a deallocation of already-free memory. If all libraries used by a project have been validated as having this behavior, then this violation does not need to be diagnosed.

5.24 Including tainted or out-of-domain input in a format string [usrfmt]

Rule

Invoking any of the formatted input/output functions identified in ISO/IEC 9899:2011, 7.21.6, where the format argument references string data that is tainted (4.15) or out-of-domain with respect to character content, shall be diagnosed.

An empty string is not tainted.

NOTE Any comparison of a character in the string to a value other than the null character may be considered to sanitize the string.

Rationale

Invoking a formatted input/output function where the format argument references string data that is tainted (4.15) or out-of-domain with respect to character content can result in undefined or unexpected behavior.

An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location and consequently execute arbitrary code with the permissions of the vulnerable process.[3]

Formatted output functions are particularly dangerous because many programmers are unaware of their capabilities. (For example, they can write an integer value to a specified address using the `%n` conversion specifier.)

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because a format string is read from a tainted source and passed as an argument to the `vfprintf` function.

```
void format_error(const char *filename, ...) {
  FILE *fd = fopen(filename, "r");
  if (fd == NULL) {
    /* ... */
  }

  char fmt[BUFSIZ];
  if (fgets(fmt, BUFSIZ, fd) == NULL) {
    /* ... */
  }

  va_list va;
  va_start(va, filename);
  vfprintf(stderr, fmt, va);  // diagnostic required
  va_end(va);

  fclose(fd);
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because the format string that is read from a tainted source and passed as an argument to the `vfprintf` function is first sanitized.

```
void safe_format_error(const char *filename, ...) {
  FILE *fd = fopen(filename, "r");
  if (fd == NULL) {
    /* ... */
  }

  char fmt[BUFSIZ];
  if (fgets(fmt, BUFSIZ, fd) == NULL) {
    /* ... */
  }
```

```
  /* only allow %d in the format string: */
  const char *fc;
  for (fc = fmt; *fc != '\0'; ++fc) {
    if (*fc == '%' && (fc[1] != '%' && fc[1] != 'd')) {
      fclose(fd);
      return;
    }
  }

  va_list va;
  va_start(va, filename);
  vfprintf(stderr, fmt, va);
  va_end(va);

  fclose(fd);
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the string `user` may contain a tainted value.

```
int incorrect_password(void) {
  int ret;
  char user[BUFSIZ];

  GET_TAINTED_STRING(user, BUFSIZ);

  static const char MSG_FORMAT[] = "%s cannot be authenticated.\n";

  size_t size = strlen(user) + sizeof(MSG_FORMAT);
  char *msg = (char *)malloc(size);

  if (msg == NULL) {
    fprintf(stderr, "Could not malloc memmory in %s\n", __func__);
    return -1;
  } else {
    ret = snprintf(msg, size, MSG_FORMAT, user); // MSG_FORMAT is not tainted

    if (ret < 0) {
      fprintf(stderr, "snprintf encoding error occurred in %s\n", __func__);
    } else if (ret >= size) {
      printf("snprintf returned %d in %s\n", ret, __func__);
    }

    fprintf(stderr, msg); // diagnostic required

    free(msg);
  }

 return 0;

}
```

EXAMPLE 4 In this compliant example, a diagnostic is not required because the argument `fmt` is constrained to be one of the elements of the `formats` array, none of which are tainted values.

```
enum int_tag { I_char, I_shrt, I_int, I_long, I_llong };
static const char *const formats[] = { "%hhi", "%hi", "%i", "%li", "%lli" };

static int fmtintv(enum int_tag tag, const char *fmt, va_list va) {
  return vfprintf(stdout, fmt, va);
}

int format_integer(enum int_tag tag, ...) {
  va_list va;
  int n;
  if (tag < I_char || I_llong < tag)
    return -1;
  va_start(va, tag);
  n = fmtintv(tag, formats[tag], va);
  va_end(va);
```



```
  return n;
}
```

5.25 Incorrectly setting and using errno [inverrno]

Rule

Incorrectly setting and using errno shall be diagnosed.

The correct way to set and check errno is defined in the following cases.

5.25.1 Library functions that set errno and return an in-band error indicator

A program that uses errno for error checking shall set errno to zero before calling one of these library functions, and then it shall inspect errno before a subsequent library function call.

The functions in Table 4 set errno and return an in-band error indicator.

Table 4 — Functions that set errno and return an in-band error indicator

Function name	Return value	errno value
ftell	-1L	positive
stroumax	UINTMAX_MAX	ERANGE
strtod,[a] wcstod	zero or ±HUGE_VAL	ERANGE
strtof, wcstof	zero or ±HUGE_VALF	ERANGE
strtoimax	INTMAX_MIN or INTMAX_MAX	ERANGE
strtol, wcstol	LONG_MIN or LONG_MAX	ERANGE
strtold, wcstold	zero or ±HUGE_VALL	ERANGE
strtoll, wcstoll	LLONG_MIN or LLONG_MAX	ERANGE
strtoul, wcstoul	ULONG_MAX	ERANGE
strtoull, wcstoull	ULLONG_MAX	ERANGE
wcstoimax	INTMAX_MIN or INTMAX_MAX	ERANGE
wcstoumax	UINTMAX_MAX	ERANGE

[a] However, according to the C Standard, if the result of strtod, strtof, or strtold (and the related wide character functions) underflows, "the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether errno acquires the value ERANGE is implementation-defined."

5.25.2 Library functions that set errno and return an out-of-band error indicator

A program may use errno for error checking after calling one of these functions without setting errno to zero before calling the function. Then the program shall inspect errno conditional on the return value and before a subsequent library function call.

The functions in Table 5 set errno and return an out-of-band error indicator.

Table 5 — Library functions that set errno value and return an out-of-band error indicator

Function name	Return value	errno value
fgetpos	nonzero	positive
fsetpos	nonzero	positive
mbrtowc	(size_t)(-1)	EILSEQ
mbsrtowcs	(size_t)(-1)	EILSEQ
signal[a]	SIG_ERR	positive
wcrtomb	(size_t)(-1)	EILSEQ
wcsrtombs	(size_t)(-1)	EILSEQ

[a] The value of errno is indeterminate if signal returns SIG_ERR from within a signal handler that was triggered by a signal that occurred other than as the result of a call to abort or raise.

5.25.3 Library functions that occasionally set errno and return an out-of-band error indicator

The fgetwc and fputwc functions return WEOF in multiple cases, only one of which results in setting errno. Therefore, the program shall set errno to zero before calling these functions.

The functions in Table 6 set errno and return an out-of-band error indicator.

Table 6 — Library functions that occasionally set errno value and return an out-of-band error indicator

Function name	Return value	errno value
fgetwc	WEOF	EILSEQ
fputwc	WEOF	EILSEQ

5.25.4 Library functions that may or may not set errno

Programs shall not rely on errno after calling a function that might set errno when an error occurs because the function might have altered errno in an implementation-defined way.

The functions defined in <complex.h> could or could not set errno when an error occurs.

The functions defined in <math.h> set errno in the following conditions:

— If there is a domain error and the integer expression math_errhandling & MATH_ERRNO is nonzero, then errno is set to EDOM.

— According to ISO/IEC 9899:2011, 7.12.1, paragraph 5, "If a floating result overflows and default rounding is in effect, then the function returns the value of the macro HUGE_VAL, HUGE_VALF, or HUGE_VALL according to the return type, with the same sign as the correct value of the function; if the integer expression math_errhandling & MATH_ERRNO is nonzero, the integer expression errno acquires the value ERANGE."

— Similarly, according to ISO/IEC 9899:2011, 7.12.1, paragraph 6, "The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression math_errhandling & MATH_ERRNO is nonzero, whether errno acquires the value ERANGE is implementation-defined."

The functions atof, atoi, atol, and atoll may or may not set errno when an error occurs.

5.25.5 Library functions that do not explicitly set `errno`

Programs shall not rely on `errno` to determine whether an error occurred after calling a Standard C Library function that does not explicitly set `errno`. Such a function may set `errno` even when no error has occurred. All library functions that have not been discussed yet are functions that do not explicitly set `errno`.

Rationale

Incorrectly setting and using `errno` can result in undefined or unexpected behavior.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because `errno` is used for error checking and `errno` is not set to zero before the C Standard Library function `strtoul` is called.

```
void f(const char *c_str) {
  char *endptr = NULL;
  unsigned long number = strtoul(c_str, &endptr, 0);

  if (endptr == c_str
  || (number == ULONG_MAX && errno == ERANGE)) {  // diagnostic required
    /* ... */
  } else {
    /* ... */
  }

  /* ... */
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because `errno` is used for error checking and the return value of the call to the C Standard Library function `signal` is not checked before checking `errno`.

```
void g(void) {
  errno = 0;
  signal(SIGINT, SIG_DFL);
  if (errno != 0) {  // diagnostic required
    /* ... */
  }
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because `errno` is used for error checking and `errno` is checked after the call to the C Standard Library function `setlocale` because `setlocale` does not explicitly set `errno`.

```
void h(void) {
  errno = 0;
  setlocale(LC_ALL, "");
  if (errno != 0) {  // diagnostic required
    /* ... */
  }
}
```

5.26 Integer division errors [diverr]

Rule

Integer values that are used as operands to the / or % operators where the second operand to the / operator or the % operator is tainted shall be diagnosed.

Tainted values that are used as either operand to the / operator or the % operator shall be diagnosed if the quotient of the two operands is not representable.

Rationale

C identifies two conditions under which division and remainder operations result in undefined behavior:

UB	Description
45	The value of the second operand of the / or % operator is zero (6.5.5).
n/a	If the quotient a/b is not representable, the behavior of both a/b and a%b is undefined (6.5.5).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the expression x / y can result in a divide-by-zero error or in a quotient that is not representable.

```
int divide(int x) {
  int y;
  GET_TAINTED_INTEGER(int, y);

  return x / y;  // diagnostic required
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the expression x % y can result in a divide-by-zero error or a quotient of the two operands that is not representable.

```
int remainder(int x) {
  int y;
  GET_TAINTED_INTEGER(int, y);

  return x % y;  // diagnostic required
}
```

EXAMPLE 3 In this compliant example, the expression x / y cannot result in a divide-by-zero error or in a quotient that is not representable.

```
int divide(int x) {
  int y;
  GET_TAINTED_INTEGER(int, y);
  if ( (y == 0) || ( (x == INT_MIN) && (y == -1) ) ) {
     /* Handle error */
  }
  else {
    return x / y;
  }
}
```

EXAMPLE 4 In this compliant example, the expression x % y cannot result in a divide-by-zero error or in a quotient that is not representable.

```
int remainder(int x) {
  int y;
  GET_TAINTED_INTEGER(int, y);
  if ( (y == 0) || ( (x == INT_MIN) && (y == -1) ) ) {
     /* Handle error */
  }
  else {
    return x % y;
  }
}
```

5.27 Interleaving stream inputs and outputs without a flush or positioning call [ioileave]

Rule

The following scenarios shall be diagnosed:

— receiving input from a stream directly following an output to that stream without an intervening call to fflush, fseek, fsetpos, or rewind, if the file is not at end-of-file or

— outputting to a stream after receiving input from that stream without a call to `fseek`, `fsetpos`, or `rewind`, if the file is not at end-of-file

Rationale

C identifies the following undefined behavior:

UB	Description
151	An output operation on an update stream is followed by an input operation without an intervening call to the `fflush` function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.21.5.3).

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because `fread` and `fwrite` are called on the same file without an intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind` on the file.

```
void f(const char *filename, char append_data[BUFSIZ]) {
  char data[BUFSIZ];
  FILE *file;

  file = fopen(filename, "a+");
  if (file == NULL) {
    /* ... */
  }

  if (fwrite(append_data, sizeof(char), BUFSIZ, file) != BUFSIZ) {
    /* ... */
  }

  if (fread(data, sizeof(char), BUFSIZ, file) != 0) {  // diagnostic required
    /* ... */
  }

  fclose(file);
}
```

5.28 Modifying string literals [strmod]

Rule

Directly modifying any portion of a string literal, assigning a string literal to a pointer to non-`const`, or casting a string literal to a pointer to non-`const` shall be diagnosed. For the purposes of this rule, the returned value of the library functions `strpbrk`, `strchr`, `strrchr`, `wcspbrk`, `wcschr`, and `wcsrchr` shall be treated as a string literal if the first argument is a string literal. For the purposes of this rule, a pointer to (or array of) `const` characters shall be treated as a string literal.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the string literal `"string literal"` is modified through the pointer `p`.

```
void f1(void) {
  char *p  = "string literal";
  p[0] = 'S';  // diagnostic required
  /* ... */
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the string literal `"/tmp/edXXXXXX"` is modified by the C Standard Library function `tmpnam`.

```
void f2(void) {
  if (tmpnam("/tmp/edXXXXXX")) {  // diagnostic required
    /* ... */
  }
```

```
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the string literal "/tmp/filename" is modified through the pointer returned from the C Standard Library function strrchr.

```
void f3(void) {
  char *last_slash = strrchr("/tmp/filename", '/');
  *last_slash = '\0';  // diagnostic required
  /* ... */
}
```

EXAMPLE 4 In this noncompliant example, a diagnostic is required because the string literal "/tmp/filename" is modified through the pointer returned from the C Standard Library function strrchr.

```
void f4(void) {
  *strrchr("/tmp/filename", '/') = '\0';  // diagnostic required
  /* ... */
}
```

EXAMPLE 5 In this noncompliant example, a diagnostic is required because the string literal "/tmp/filename" is modified.

```
void f5(void) {
  "/tmp/filename"[4] = '\0';  // diagnostic required
  /* ... */
}
```

Exception

No diagnostic need be issued if the analyzer can determine that the value of the pointer to non-const is never used to attempt to modify the characters of the string literal.

EXAMPLE

```
int main(void) {
  char *p = "abc";
  printf("%s\n", p);
  return EXIT_SUCCESS;
}
```

5.29 Modifying the string returned by getenv, localeconv, setlocale, and strerror [libmod]

Rule

Modifying the objects or strings returned by the library functions listed in getenv, localeconv, setlocale, and strerror shall be diagnosed.

Rationale

C identifies the following three instances of undefined behavior, which arise as a result of modifying the data structures or strings returned from getenv, localeconv, setlocale, and strerror:

UB	Description
120	The program modifies the string pointed to by the value returned by the setlocale function (7.11.1.1).
121	The program modifies the structure pointed to by the value returned by the localeconv function (7.11.2.1).
184	The string set up by the getenv or strerror function is modified by the program (7.22.4.6, 7.24.6.2).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the string returned from the C Standard Library function setlocale is modified.


```
void f1(void) {
  char *locale = setlocale(LC_ALL, 0);
  if (locale != NULL) {
    char *cats[8];
    char *sep = locale;
    cats[0] = locale;
    int i;

    if (sep) {
      for (i = 0; (sep = strstr(sep, ";:")) && i < 8; ++i) {
        *sep = '\0';  // diagnostic required
        cats[i] = ++sep;
      }
    }
  }

  /* ... */
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the object returned from the C Standard Library function `localeconv` is modified.

```
void f2(void) {
  struct lconv *conv = localeconv();

  if ('\0' == conv->decimal_point[0]) {
    conv->decimal_point = ".";  // diagnostic required
  }

  if ('\0' == conv->thousands_sep[0]) {
    conv->thousands_sep = ",";  // diagnostic required
  }

  /* ... */
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the string returned from the C Standard Library function `getenv` is modified.

```
void f3(void) {
  char *shell_dir = getenv("SHELL");

  if (shell_dir != NULL) {
    char *slash = strrchr(shell_dir, '/');
    if (slash) {
      *slash = '\0';  // diagnostic required
    }

    /* use shell_dir */
  }
}
```

EXAMPLE 4 In this noncompliant example, a diagnostic is required because the string returned from the C Standard Library function `strerror` is modified.

```
const char *f4(int error) {
  char buf[BUFFER_SIZE];
  sprintf(buf, "(errno = %d)", error);

  char *error_str = strerror(error);

  strcat(error_str, buf);  // diagnostic required
  return error_str;
}
```

5.30 Overflowing signed integers [intoflow]

Rule

Whenever at least one operand is tainted, signed integer operations that trap and that can overflow shall be diagnosed.

Rationale

Signed integer overflow is undefined behavior.

UB	Description
36	An exceptional condition occurs during the evaluation of an expression (6.5).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required on implementations that trap on signed integer overflow because the expression `x + 1` may result in signed integer overflow.

```
int add(void) {
  int x;
  GET_TAINTED_INTEGER(int, x);

  return x + 1;  // diagnostic required
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because the expression `x + 1` cannot result in signed integer overflow.

```
int add(void) {
  int x;
  GET_TAINTED_INTEGER(int, x);

  if (x < INT_MAX) {
    return x + 1;
  } else {
    return INT_MIN;
  }
}
```

5.31 Passing a non-null-terminated character sequence to a library function that expects a string [nonnullcs]

Rule

Passing a character sequence or wide character sequence that is not null-terminated to a library function that expects a string or wide string argument shall be diagnosed.

Rationale

Many library functions accept a string or wide string argument with the constraint that the string they receive is properly null-terminated. Passing a character sequence or wide character sequence that is not null-terminated to such a function can result in accessing memory that is outside the bounds of the object.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the character sequence `str` will not be null-terminated when passed as an argument to `printf`.

```
char str[3] = "abc";
printf("%s\n", str);            // diagnostic required
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the wide character sequence `cur_msg` will not be null-terminated when passed to `wcslen`. This will occur if `lessen_memory_usage` is invoked while `cur_msg_size` still has its initial value of 1024.

```
wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;
```



```
void lessen_memory_usage(void) {
  wchar_t *temp;
  size_t temp_size;

  /* ... */

  if (cur_msg != NULL) {
    temp_size = cur_msg_size / 2 + 1;
    temp = realloc(cur_msg, temp_size * sizeof(wchar_t));
           // temp & cur_msg might not be null-terminated
    if (temp == NULL) {
      /* ... */
    }

    cur_msg = temp;
    cur_msg_size = temp_size;
    cur_msg_len = wcslen(cur_msg);  // diagnostic required
  }
}
```

EXAMPLE 3 In this compliant example, a diagnostic is not required because `cur_msg` will always be null-terminated when passed to `wcslen`.

```
wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;

void lessen_memory_usage(void) {
  wchar_t *temp;
  size_t temp_size;

  /* ... */

  if (cur_msg != NULL) {
    temp_size = cur_msg_size / 2 + 1;
    temp = realloc(cur_msg, temp_size * sizeof(wchar_t));
           // temp & cur_msg might not be null-terminated
    if (temp == NULL) {
      /* ... */
    }

    cur_msg = temp;
    cur_msg[ temp_size - 1] = L'\0'; // cur_msg now properly null-terminated
    cur_msg_size = temp_size;
    cur_msg_len = wcslen(cur_msg);  // diagnostic not required
  }
}
```


5.32 Passing arguments to character-handling functions that are not representable as `unsigned char` [chrsgnext]

Rule

Arguments to the character-handling functions in `<ctype.h>` that are not representable as `unsigned char` shall be diagnosed.

The following character classification functions are affected:

`isalnum`	`isalpha`	`isascii`	`isblank`
`iscntrl`	`isdigit`	`isgraph`	`islower`
`isprint`	`ispunct`	`isspace`	`isupper`
`isxdigit`	`toascii`	`toupper`	`tolower`

Rationale

These character classification functions are defined only for values representable as `unsigned char` and the macro `EOF`.

Example(s)

EXAMPLE On implementations where plain `char` is signed, a diagnostic is required because the parameter to `isspace`, `*t`, is defined as a `const char *`, and this value may not be representable as an `unsigned char`.

```
size_t count_preceding_whitespace(const char *s) {
  const char *t = s;
  size_t length = strlen(s) + 1;

  while (isspace(*t) && (t - s < length)) {  // diagnostic required
    ++t;
  }
  return t - s;
}
```

5.33 Passing pointers into the same object as arguments to different `restrict`-qualified parameters [restrict]

Rule

Function arguments that are `restrict`-qualified pointers and reference overlapping objects shall be diagnosed.

Rationale

C identifies the following undefined behavior:

UB	Description
68	An object which has been modified is accessed through a restrict-qualified pointer to a const-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the `restrict`-qualified pointer parameters to `memcpy`, `ptr1`, and `ptr2` reference overlapping objects.

```
void abcabc(void) {
  char c_str[]= "abc123edf";
  char *ptr1 = c_str;
  char *ptr2 = c_str + strlen("abc");

  memcpy(ptr2, ptr1, 6);  // diagnostic required
  puts(c_str);
}
```

EXAMPLE 2 In this noncompliant example, the `src` operand is used twice to refer to unmodified memory, which is allowed by the C Standard; the aliasing restrictions apply only when the object is modified. A diagnostic is required nontheless because the pointer `src` is twice a `restrict`-qualified pointer parameter to `dual_memcpy`, referencing overlapping objects.

```
void *dual_memcpy(
  void *restrict  s1, const void *restrict  s2, size_t n1,
  void *restrict  s3, const void *restrict  s4, size_t n2
) {
  memcpy(s1, s2, n1);
  memcpy(s3, s4, n2);

  return s1;
}

void f(void) {
  char dest1[10];
```



```
  char dest2[10];
  char src[] = "hello";

  dual_memcpy(dest1, src, sizeof(src),
              dest2, src, sizeof(src));  // diagnostic required
  puts(dest1);
  puts(dest2);
}
```

5.34 Reallocating or freeing memory that was not dynamically allocated [xfree]

Rule

Calling `realloc` or `free` in cases where the `ptr` argument to either function may refer to memory that was not dynamically allocated shall be diagnosed.

Rationale

C identifies the following undefined behavior:

UB	Description
179	The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to `free` or `realloc` (7.22.3.3, 7.22.3.5).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the pointer parameter to `realloc`, buf, does not refer to dynamically allocated memory.

```
#define BUFSIZE 256

void f(void) {
  char buf[BUFSIZE];
  char *p;
  /* ... */
  p = (char *)realloc(buf, 2 * BUFSIZE);  // diagnostic required
  /* ... */
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the pointer parameter to `free`, `c_str` may not refer to dynamically allocated memory.

```
#define MAX_ALLOCATION 1000

int main(int argc, char *argv[]) {
  char *c_str = NULL;
  size_t len;

  if (argc == 2) {
    len = strlen(argv[1]) + 1;
    if (len > MAX_ALLOCATION) {
      /* Handle error */
    }
    c_str = (char *)malloc(len);
    if (c_str == NULL) {
      /* Handle allocation error */
    }
    strcpy(c_str, argv[1]);
  }
  else {
    c_str = "usage: $>a.exe[string]";
    printf("%s\n", c_str);
  }
  /* ... */
  free(c_str);  // diagnostic required
  return EXIT_SUCCESS;
```

```
}
```

Exception

Some library implementations accept and ignore a deallocation of non-allocated memory (or, alternatively, cause a runtime-constraint violation). If all libraries used by a project have been validated as having this behavior, then this violation does not need to be diagnosed.

5.35 Referencing uninitialized memory [uninitref]

Rule

Accessing uninitialized memory by an lvalue of a type other than `unsigned char` shall be diagnosed.

Rationale

There are two main sources of uninitialized memory:

— uninitialized automatic variables and

— uninitialized memory returned by the memory management functions `malloc`, `realloc`, and `aligned_alloc`.

Uninitialized memory has indeterminate value, which for objects of some types can be a trap representation. Accessing uninitialized memory by an lvalue of a type other than `unsigned char` has undefined behavior. Typical consequences of accessing uninitialized memory relevant to security range from denial of service to information exposure as a result of leaking sensitive data previously stored in a memory region.

C identifies the following undefined behaviors:

UB	Description
11	The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.9, 6.8).
12	A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).

It should be noted that while it is safe to copy a region of uninitialized storage into another location using a function such as `memcpy`, after the copy, the destination region has the same "uninitialized" contents as the source region even if it had been initialized to a determinate value before the copy.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the variable `sign` may be uninitialized when it is accessed in the `return` statement of the function `is_negative`.

```
void get_sign(int number, int *sign) {
  if (sign == NULL) {
    /* ... */
  }

  if (number > 0) {
    *sign = 1;
  } else if (number < 0) {
    *sign = -1;
  } // If number == 0, sign is not changed.
}

int is_negative(int number) {
  int sign;
  get_sign(number, &sign);

  return (sign < 0);  // diagnostic required
}
```


EXAMPLE 2 In this noncompliant example, a diagnostic is required because the variable `error_log` is uninitialized when it is passed to `sprintf`.

```
int do_auth(void) {
  int result = -1;

  /* ... */
  return result;
}

void report_error(const char *msg) {
  const char *error_log;
  char buffer[24];

  sprintf(buffer, "Error: %s", error_log);  // diagnostic required
  printf("%s\n", buffer);
}

int main(void) {
  if (do_auth() == -1) {
    report_error("Unable to login");
  }

  return EXIT_SUCCESS;
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the elements of the array `a` are uninitialized when they are accessed in the `for` loop.

```
void f(size_t n) {
  int *a = (int *)malloc(n * sizeof(int));
  if (a != NULL) {
    for (size_t i = 0; i != n; ++i) {
      a[i] = a[i] ^ a[i];  // diagnostic required
    }

    /* ... */
    free(a);
  }
}
```

EXAMPLE 4 In this noncompliant example, a diagnostic is required because the array elements `a[n..2n]` are uninitialized when they are accessed in the `for` loop.

```
void g(double *a, size_t n) {
  a = (double *)realloc(a, (n * 2 + 1) * sizeof(double));
  if (a != NULL) {
    for (size_t i = 0; i != n * 2 + 1; ++i) {
      if (a[i] < 0) {
        a[i] = -a[i];  // diagnostic required
      }
    }

    /* ... */
    free(a);
  }
}
```

5.36 Subtracting or comparing two pointers that do not refer to the same array [ptrobj]

Rule

Subtracting or relationally comparing two pointers that do not refer to the same array object, or one element past the same array object, shall be diagnosed. The relational operators are >, <, >=, and <=.

Rationale

C identifies two distinct situations in which undefined behavior may arise as a result of using pointers that do not point to the same object:

UB	Description
48	Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).
53	Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the pointers `c_str` and `(char **)next_num_ptr` are subtracted and do not refer to the same array.

```
#define SIZE 256

void f(void) {
  int nums[SIZE];
  char *c_str[SIZE];
  int *next_num_ptr = nums;
  int free_bytes;

  /* ... */
  /* increment next_num_ptr as array fills */

  free_bytes = c_str - (char **)next_num_ptr;  // diagnostic required
  /* ... */
}
```

Exceptions

— EX1: Comparing two pointers within the same object does not need to be diagnosed.

— EX2: Subtracting two pointers to `char` within the same object does not need to be diagnosed.

5.37 Tainted strings are passed to a string copying function [taintstrcpy]

Rule

Tainted values that are referenced by the source argument to the `strcpy`, `strcat`, `wcscpy`, or `wcscat` function and that exceed the size of the destination array shall be diagnosed.

Rationale

Passing wide or narrow strings as the source argument to the `strcpy`, `strcat`, `wcscpy`, or `wcscat` function that exceed the size of the destination array can result in writing to memory that is outside the bounds of existing objects.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the size of the string referenced by `argv[0]` might be greater than the size of the destination array `pgm`.

```
void main(int argc, char *argv[]) {
  char pgm[BUFSIZ];

  if (argc > 1) {
    strcpy(pgm, argv[0]);  // diagnostic required
  }
}
```

5.38 Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr]

Rule

Using the `sizeof` operator on an array parameter shall be diagnosed.

Rationale

Using the `sizeof` operator on an array parameter frequently indicates a programmer error and can result in unexpected behavior.

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the `sizeof` operator is applied to the pointer parameter `array`.

```
void clear(int array[]) {
  for (size_t i = 0;
       i < sizeof(array) / sizeof(array[0]); // diagnostic required
       ++i) {
     array[i] = 0;
  }
}
```

5.39 Using a tainted value as an argument to an unprototyped function pointer [taintnoproto]

Rule

Passing a tainted value as an argument to a call through a pointer to a function that was declared without a prototype shall be diagnosed.

Rationale

A pointer to a function declared without a prototype may refer to a function whose parameters ultimately flow into a restricted sink. When a prototype is available, an analyzer might be able to determine if the pointer points to such a function.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the tainted argument `tainted` is passed as an argument to a call through an unprototyped pointer to function `pf`. The initialization of `pf` and the definition of `restricted_sink` are informational and not necessary for this diagnosis.

```
void restricted_sink(int i) {
  int array[2];
  array[i] = 0;
}

void (*pf)() = restricted_sink;

void f(void) {
  int tainted;
  GET_TAINTED_INTEGER(int, tainted);
  (*pf)(tainted);   // diagnostic required
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because the tainted argument `tainted2` is passed as an argument to a call through a properly prototyped pointer to function `pf2`.

```
void (*pf2)(int);

void g(void) {
  int tainted2;
  GET_TAINTED(int, tainted2);
  (*pf2)(tainted2);
}
```


5.40 Using a tainted value to write to an object using a formatted input or output function [taintformatio]

Rule

Calls to the `fscanf`, `scanf`, `vfscanf`, and `vscanf` functions that pass tainted values as arguments and that can result in writes outside the bounds of the specified object shall be diagnosed.

Calls to the `sscanf` and `vsscanf` functions that can result in writes outside the bounds of the specified object shall also be diagnosed when the input string is tainted.

Calls to the `sprintf` function that can result in writes outside the bounds of the destination array shall be diagnosed when any of its variadic arguments are tainted.

Rationale

Calls to the standard C formatted input functions declared in `<stdio.h>`, as well as to the `sprintf` function, can corrupt memory.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the call to `fscanf` can result in a write outside the character array `buf`.

```
char buf[BUF_LENGTH];
fscanf(stdin, “%s”, buf); // diagnostic required
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the `sprintf` function will write outside the bounds of the character array `buf`.

```
int rc = 0;
int x;
GET_TAINTED_INTEGER(int, x);
char buf[sizeof(“999”)];
rc = sprintf(buf, “%d”, x); // diagnostic required
if (rc == -1 || rc >= sizeof(buf)) {
  /* handle error */
}
```

5.41 Using a value for `fsetpos` other than a value returned from `fgetpos` [xfilepos]

Rule

Using an offset value for `fsetpos`, other than a value returned from `fgetpos`, shall be diagnosed.

Rationale

C identifies the following undefined behavior:

UB	Description
175	The `fsetpos` function is called to set a position that was not returned by a previous successful call to the `fgetpos` function on a stream associated with the same file (7.21.9.3).

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because an offset value other than one returned from `fgetpos` is used in a call to `fsetpos`.

```
FILE *opener(const char *filename) {
  fpos_t offset;

  if (filename == NULL) {
    /* ... */
  }
```

```
  FILE *file = fopen(filename, "r");
  if (file == NULL) {
    /* ... */
  }

  memset(&offset, 0, sizeof(offset));

  if (fsetpos(file, &offset) != 0) {  // diagnostic required
    /* ... */
  }

  return file;
}
```

5.42 Using an object overwritten by `getenv`, `localeconv`, `setlocale`, and `strerror` [libuse]

Rule

Using the object pointed to by the pointer returned by the `getenv`, `localeconv`, `setlocale`, and `strerror` functions after a subsequent call to the function shall be diagnosed.

Rationale

The object pointed to by the pointer returned by the `getenv`, `localeconv`, `setlocale`, and `strerror` functions may be overwritten by the subsequent call to the function.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the string returned by the first call to the C Standard Library function `getenv` is accessed, after the second call to `getenv`, in the call to the C Standard Library function `strcmp`.

```
int f(void) {
  char *tmpvar = getenv("TMP");
  char *tempvar = getenv("TEMP");

  if (!tmpvar || !tempvar) {
    /* ... */
  }

  return strcmp(tmpvar, tempvar) == 0;  // diagnostic required
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the string returned by the first call to the C Standard Library function `setlocale` is accessed, after the second call to `setlocale`, in the third call to `setlocale`.

```
void g(const char *name) {
  const char *save = setlocale(LC_ALL, 0);
  if (setlocale(LC_ALL, name)) {
    /* ... */
  }

  setlocale(LC_ALL, save);  // diagnostic required
}
```

EXAMPLE 3 In this noncompliant example, a diagnostic is required because the pointer returned from the first call to the C Standard Library function `strerror` is accessed, in the call to `fprintf`, after the second call to `strerror`.

```
void h(const char *a, const char *b) {
  errno = 0;
  unsigned long x = strtoul(a, NULL, 0);
  int e1 = ULONG_MAX == x ? errno : 0;

  errno = 0;
  unsigned long y = strtoul(b, NULL, 0);
  int e2 = ULONG_MAX == y ? errno : 0;
```



```
  char* err1 = strerror(e1);
  char* err2 = strerror(e2);
  fprintf(stderr, "parsing results: %s, %s", err1, err2); // diagnostic required
}
```

5.43 Using character values that are indistinguishable from EOF [chreof]

Rule

The following library character functions have return type int and return character values and the value EOF.

fgetc   getc   getchar

If the return value of one of the above library functions is stored into a variable of type char, any comparison of that stored value to a constant equal to the value of EOF shall be diagnosed.

Similarly, the following library wide character functions have return type wint_t and return wide character values and the value WEOF.

fgetwc   getwc   getwchar

If the return value of one of the above library functions is stored into a variable of type wchar_t, any comparison of that stored value to a constant equal to the value of WEOF shall be diagnosed.

Rationale

A character type cannot represent all character values plus the value of EOF, and a wide character type cannot represent all character values plus the value of WEOF.

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the result of the call to the C Standard Library function getchar is stored into a variable of type char, c, and c is compared to EOF.

```
void f(void) {
  char buf[BUFSIZ];
  char c;
  size_t i = 0;

  while ((c = getchar())
    != '\n' && c != EOF) {  // diagnostic required
    if (i < BUFSIZ - 1) {
      buf[i++] = c;
    }
  }

  buf[i] = '\0';
  printf("%s\n", buf);
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the result of the call to the C Standard Library function getwc is stored into a variable of type wchar_t, wc, and wc is compared to WEOF.

```
void g(void) {
  wchar_t buf[BUFSIZ];
  wchar_t wc;
  size_t i = 0;

  while ((wc = getwc(stdin))
    != '\n' && wc != WEOF) {  // diagnostic required
    if (i < BUFSIZ - 1) {
      buf[i++] = wc;
    }
```



```
  }

  buf[i] = ‘\0’;
  wprintf(“%s\n”, buf);
}
```

5.44 Using identifiers that are reserved for the implementation [resident]

Rule

Declaring or defining an identifier in a context in which it is reserved or defines a reserved identifier as a macro name shall be diagnosed.

Rationale

According to ISO/IEC 9899:2011, 7.1.3, on reserved identifiers,

— All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use.

— All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary and tag name spaces.

— Each macro name in any of the subclauses (including the future library directions) is reserved for use as specified if any one of its associated headers is included, unless explicitly stated otherwise.

— All identifiers with external linkage . . . (including the future library directions) and `errno` are always reserved for use as identifiers with external linkage.

— Each identifier with file scope listed in any of the above subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

No other identifiers are reserved. The behavior of a program that declares or defines an identifier in a context in which it is reserved or defines a reserved identifier as a macro name is undefined. Trying to define a reserved identifier can result in its name conflicting with that used in implementation, which may or may not be detected at compile time.

C identifies the following undefined behavior:

UB	Description
106	The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).

NOTE The POSIX® standard extends the set of identifiers reserved by C to include an open-ended set of its own.[4]

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the reserved identifier `errno` is redefined.

```
extern int errno;  // diagnostic required
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the identifier `_MY_HEADER_H_` defined in the header guard is reserved because it begins with an underscore and an uppercase letter.

```
#ifndef _MY_HEADER_H_
#define _MY_HEADER_H_  // diagnostic required

/* contents of <my_header.h> */

#endif /* _MY_HEADER_H_ */
```

EXAMPLE 3 In this compliant example, a diagnostic is not required because the identifier `MY_HEADER_H` defined in the header guard is not reserved.


```
#ifndef MY_HEADER_H
#define MY_HEADER_H

/* contents of <my_header.h> */

#endif /* MY_HEADER_H */
```

EXAMPLE 4 In this noncompliant example, a diagnostic is required because the file scope identifiers `_max_limit` and `_limit` are reserved because they begin with an underscore.

```
static const unsigned int _max_limit = 1024;  // diagnostic required
unsigned int _limit = 100;  // diagnostic required

unsigned int getValue(unsigned int count) {
  return count < _limit ? count : _limit;
}
```

EXAMPLE 5 In this compliant example, a diagnostic is not required because the file scope identifiers `max_limit` and `limit` are not reserved because they do not begin with an underscore.

```
static const unsigned int max_limit = 1024;
unsigned int limit = 100;

unsigned int getValue(unsigned int count){
  return count < limit ? count : limit;
}
```

EXAMPLE 6 In this noncompliant example, a diagnostic is required if header `<stdint.h>` is included because the identifier `SIZE_MAX` is reserved and the identifier `INTFAST16_LIMIT_MAX` is reserved because it begins with `INT` and ends with `_MAX`.

```
#define BUFSIZE 80

static const int_fast16_t INTFAST16_LIMIT_MAX = 12000;  // diagnostic required

void print_fast16(int_fast16_t val) {
  enum { SIZE_MAX = 80 };  // diagnostic required
  char buf[BUFSIZE];

  if (INTFAST16_LIMIT_MAX < val) {
    sprintf(buf, “The value is too large”);
  } else {
    char fmt[BUFSIZE];
    snprintf(fmt, BUFSIZE, “%s %s”, “The Value is %”, PRIdFAST16);
    snprintf(buf, BUFSIZE, fmt, val);
  }
  fprintf(stdout, “\t%s\n”, buf);
}
```

EXAMPLE 7 In this compliant example, a diagnostic is not required because the identifiers `BUFSIZE` and `MY_INTFAST16_UPPER_LIMIT` are not reserved.

```
static const int_fast16_t MY_INTFAST16_UPPER_LIMIT = 12000;

void print_fast16(int_fast16_t val) {
  enum { BUFSIZE = 80 };
  char buf[BUFSIZE];
  if (MY_INTFAST16_UPPER_LIMIT < val) {
    sprintf(buf, “The value is too large”);
  } else {
    char fmt[BUFSIZE];
    snprintf(fmt, BUFSIZE, “%s %s”, “The Value is %”, PRIdFAST16);
    snprintf(buf, BUFSIZE, fmt, val);
  }
  fprintf(stdout, “\t%s\n”, buf);
}
```

EXAMPLE 8 In this noncompliant example, a diagnostic is required because the identifiers for the C Standard Library functions `malloc` and `free` are reserved.


```
void *malloc(size_t nbytes) {  // diagnostic required
  void *ptr;
  /* ... */
  /* allocate storage from own pool and set ptr */
  return ptr;
}

void free(void *ptr) {  // diagnostic required
  /* ... */
  /* return storage to own pool */
}
```

EXAMPLE 9 In this compliant example, a diagnostic is not required because the reserved identifiers `malloc` and `free` are not used to define functions.

```
char *my_malloc(size_t nbytes) {
  if (nbytes > 0) {
    return (char *)malloc(nbytes);
  } else {
    return NULL;
  }
}

void my_free(char *p) {
  if(p != NULL) {
    free(p);
    p = NULL;
  }
}
```

5.45 Using invalid format strings [invfmtstr]

Rule

Supplying an unknown or invalid *conversion specification*; an invalid combination of *flag character*, *precision*, *length modifier*, *conversion specifier*; or a number and type of arguments to a formatted IO function that do not match the conversion specifiers in the format string shall be diagnosed.

Rationale

C identifies the following undefined behaviors:

UB	Description
155	In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).
156	A conversion specification for a formatted output function uses an asterisk to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.21.6.1, 7.29.2.1).
157	A conversion specification for a formatted output function uses a `#` or `0` flag with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).
158	A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).
159	An `s` conversion specifier is encountered by one of the formatted output functions, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.21.6.1, 7.29.2.1).
160	An `n` conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).
161	A `%` conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly `%%` (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).
162	An invalid conversion specification is found in the format for one of the formatted input/output functions, or the `strftime` or `wcsftime` function (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

Example(s)

EXAMPLE In this noncompliant example, a diagnostic is required because the arguments to `printf` do not match the conversion specifiers in the supplied format string.

```
void f(void) {
  const char *error_msg = "Resource not available to user.";
  int error_type = 3;
  /* ... */
  printf("Error (type %s): %d\n", error_type, error_msg);  // diagnostic required
}
```

5.46 Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink [taintsink]

Rule

Values that are tainted (4.16), potentially mutilated, or out-of-domain integers and are used in an integer restricted sink shall be diagnosed.

Rationale

Using tainted (4.16), potentially mutilated, or out-of-domain integers in an integer restricted sink can result in accessing memory that is outside the bounds of existing objects.

Restricted sinks for integers include

— integer operands of any pointer arithmetic, including array indexing;

— the assignment expression for the declaration of a variable length array;

— the postfix expression preceding square brackets `[]` or the expression in square brackets `[]` of a subscripted designation of an element of an array object; and

— function arguments of type `size_t` or `rsize_t` (for example, an argument to a memory allocation function).

Example(s)

EXAMPLE 1 In this noncompliant example, a diagnostic is required because the tainted integer `size` is used to declare the size of the variable length array `vla`.

```
void f(const char *c_str) {
  size_t size;
  GET_TAINTED_INTEGER(size_t, size);
  char vla[size];  // diagnostic required

  strncpy(vla, c_str, size);
  vla[size - 1] = '\0';

  /* ... */
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the tainted integer `color_index` is used in pointer arithmetic to index into the array `table`.

```
const char *table[] = { "black", "white", "blue", "green" };

const char *set_background_color(void) {
  int color_index;
  GET_TAINTED_INTEGER(int, color_index);

  const char *color = table[color_index];  // diagnostic required

  /* ... */
  return color;
}
```


Annex A
(informative)

Intra- to Interprocedural Transformations

Most of the examples given in the individual rules are simple and most often intraprocedural, only requiring analysis of a single function to either diagnose the violation or determine that there is no violation. Real-world code is often more complicated, requiring some form of interprocedural analysis. Collecting evidence for or against a violation may require examining the flow of data from one function to another, either explicitly through argument passing and return values or implicitly through global variables and the heap. There are many different approaches to interprocedural analysis, and this technical specification does not advocate one approach over another. This annex describes a transformational framework that can be used to generate more complex interprocedural examples from the simple intraprocedural examples given in the rules. These generated examples can then be used as an extended set of requirements for interprocedural analysis.

A.1 Function arguments and return values

The simplest case is a rule involving only one value, such as **Failing to detect and handle standard library errors** [liberr]. The following is an intraprocedural example:

```
FILE *fp = fopen(name, mode);
if (fp != NULL) /* checking for success */
  /* ... */
```

The basic interprocedural transformations are to pass the value into a function or return it from a function:

```
void check_it(FILE *fp) {
  if (fp != NULL) /* checking for success */
    /* ... */
}

/* ... */
check_it(fopen(name, mode));
```

or

```
/* a wrapper around fopen */
FILE *xfopen(const char *name, const char *mode) {
  return fopen(name, mode); /* return for checking elsewhere */
}

...
FILE *fp = xfopen(name, mode);
if (fp != NULL) /* checking for success */
  /* ... */
```

An important special case combines argument passing and returning to form an identity operation:

```
/* trivial example */
FILE *identity(FILE *fp) {
  return fp;
}

FILE *fp = identity(fopen(name, mode));
if (fp != NULL) /* checking for success */
  /* ... */
```

A.2 Indirection

An additional transformation is indirection:

```
void check_indirect(FILE **pfp) {
  if (*pfp != NULL) /* checking for success */
    /* ... */
}

/* ... */
FILE *fp = fopen(name, mode);
check_indirect(&fp);

void return_result_thru_param(const char *name, const char *mode,
                              FILE **result) {
  *result = fopen(name, mode);
}

/* ... */
FILE *fp;
return_result_thru_param(name, mode, &fp);
if (fp != NULL) /* checking for success */
  /* ... */
```

Indirection can also involve fields of structs or unions. Indirection can be applied recursively, although precise handling of indirection often becomes increasingly expensive as the number of levels of indirection increases.

When a rule involves multiple values, such as **Forming or using out-of-bounds pointers or array subscripts** [invptr] (where a violation is an interaction between an array and an index), these transformations apply separately or in combination to each of the values. The following is a simple intraprocedural example:

```
int array[2];
int index = 2; /* part of violation */
array[index] = 0; /* violation */
```

Applying some of the interprocedural transformations yields

```
void indexer(int *array, int index) {
  array[index] = 0; /* part of violation */
}

/* ... */
int array[2];
int index = 2; /* part of violation */
indexer(array, index); /* violation */
```

or

```
static int array[2];
int *get_array() {
  return array; /* part of violation */
}

/* ... */
get_array()[2] = 0; /* violation */
```

or

```
struct array_params {
  int *array;
  int index;
};

void indexer(struct array_params *ap) {
  ap->array[ap->index] = 0; /* part of violation */
}
```



```
/* ... */
int array[2];
struct array_params params;
params.array = array; /* part of violation */
params.index = 2; /* part of violation */
indexer(&params); /* violation */
```

These violations involve three steps: the array, the index, and the address arithmetic, where each could occur in a different function:

```
int *add(int *base, int offset) {
  return base + offset; /* part of violation */
}

/* ... */
int array[2];
int index = 2; /* part of violation */
*add(array, index) = 0; /* violation */
```

Indirection may also be applied to a pointer being returned from a function:

```
const int *ptr_to_index() {
  static int rv;
  rv = 2; /* part of violation */
  return &rv; /* part of violation */
}
/* ... */
int array[2];
array[*ptr_to_index()] = 0; /* violation */
```

A.3 Transformation involving standard library functions

The following transformation involves tracing the flow of data through the C Standard Library function `strchr()` that returns a pointer to an element in the array specified by its first argument if the element's value equals that of the second argument and a null pointer otherwise. Because the effects and the return value of the function are precisely specified, an analyzer can determine that the assignment to the `*slash` object modifies an element of the `const` string `pathname`, potentially causing undefined behavior.

```
const char* basename(const char *pathname) {
  char *slash;

  slash = strchr(pathname, '/');
  if (slash) {
    *slash++ = '\0'; /* violates EXP40-C. Do not modify constant values */
    return slash;
  }

  return pathname;
}
```

Note that interprocedural analysis could identify such data flow through an arbitrary function, but in the case of standard library functions, it is not necessary to analyze the function's implementation to derive the flow.

A.4 Data flow through globals

Interprocedural data flow may not be explicit through argument passing and value returning; it may involve `extern` or `static` variables:

```
int global_index;

void set_it() {
  global_index = 2; /* part of violation */
}

/* ... */
```

```
int array[2];
set_it(); /* part of violation */
array[global_index] = 0; /* violation */
```

A.5 Data flow through the heap

Data may flow from one function to another through heap locations:

```
struct some_record {
  int index;
  /* ... */
};

/* returns the heap-allocated record (created elsewhere)
 * matching 'key'
 */
struct some_record *find_record(const char *key);

void set_it() {
  struct some_record *record = find_record("xyz");
  record->index = 2;
}

/* ... */
int array[2];
set_it(); /* part of violation */
struct some_record *record = find_record("xyz"); /* part of violation */
array[record->index] = 0; /* violation */

/* note that find_record() could even be called before set_it() */
```

A.6 Combined example

This example combines some of these transformations in a way that a reasonably sophisticated interprocedural analysis might diagnose:

```
struct trouble {
  int *array;
  int index;
  int *effective_address;
};

void set_array(struct trouble *t, int *array) {
  t->array = array; /* part of violation */
}

void set_index(struct trouble *t, int *index) {
  t->index = *index; /* part of violation */
}

void compute_effective_address(struct trouble *t) {
  t->effective_address = t->array + t->index; /* part of violation */
}

void store(struct trouble *t, int value) {
  *t->effective_address = value; /* part of violation */
}

...
int array[2];
int index = 2; /* part of violation */
struct trouble t;
set_array(t, array); /* part of violation */
set_index(t, &index); /* part of violation */
compute_effective_address(&t); /* violation */
store(&t, 0); /* violation */
```


Annex B
(informative)

Undefined Behavior

According to C (as summarized in ISO/IEC 9899:2011, J.2), the behavior of a program is undefined in the circumstances outlined in Table B.1. The parenthesized section numbers refer to the section of ISO/IEC 9899:2011 that identifies the undefined behavior.

Table B.1 — Undefined behaviors

UB	Description
1	A “shall” or “shall not” requirement that appears outside of a constraint is violated (clause 4).
2	A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).
3	Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).
4	A program in a hosted environment does not define a function named `main` using one of the specified forms (5.1.2.2.1).
5	The execution of a program contains a data race (5.1.2.4).
6	A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).
7	An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).
8	The same identifier has both internal and external linkage in the same translation unit (6.2.2).
9	An object is referred to outside of its lifetime (6.2.4).
10	The value of a pointer to an object whose lifetime has ended is used (6.2.4).
11	The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.9, 6.8).
12	A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).
13	A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).
14	The operands to certain operators are such that they could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).
15	Two declarations of the same object or function specify types that are not compatible (6.2.7).
16	A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).
17	Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).
18	Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).
19	An lvalue does not designate an object when evaluated (6.3.2.1).
20	A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).
21	An lvalue designating an object of automatic storage duration that could have been declared with the `register` storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1).
22	An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).
23	An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to `void`) is applied to a void expression (6.3.2.2).