WG14 Document: N1579

ISO/IEC JTC 1/SC SC 22 N 1579

Date: 2011-09-20

ISO/IEC

ISO/IEC JTC 1/SC SC 22/WG WG 14

Secretariat: ANSI

Information Technology — Programming languages, their environments and system software interfaces — C Secure Coding Rules

Technologies de l'information — Langages de programmation, leurs environnements et interfaces du logiciel système — C Règles de codage sécurisé

Document type: Technical Specification
Document subtype:
Document stage:
Document language: E

\\ad\dfs\users\rcs\Documents\CSGR\N1579.docx STD Version 2.1c2

Contents

Foreword

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

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

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

In other circumstances, particularly when there is an urgent market requirement for such documents, a technical committee may decide to publish other types of normative document:

— an ISO/IEC Publicly Available Specification (ISO/IEC PAS) represents an agreement between technical experts in an ISO working group and is accepted for publication if it is approved by more than 50 % of the members of the parent committee casting a vote;

— an ISO/IEC Technical Specification (ISO/IEC TS) represents an agreement between the members of a 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/PAS or ISO/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order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn.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/PAS or ISO/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 was prepared by Technical Committee ISO/TC , , Subcommittee SC SC 22, .

This second/third/... edition cancels and replaces the first/second/... edition (), [clause(s) / subclause(s) / table(s) / figure(s) / annex(es)] of which [has / have] been technically revised.

Introduction

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.

This Technical Specification has two major subdivisions:

— preliminary elements (clauses 1-5) and

— secure coding rules (clauses 6-18).

The rules documented in this technical specification rely only on non-annotated source files and not upon 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. Additionally, there are significant differences in rules that are intended primarily for evaluating new code versus legacy code. Because security is the primary concern, these rules are intended first and foremost for evaluating new code and secondarily for evaluating legacy code. Consequently, the application of these rules to legacy code may result in false positives. However, legacy code is generally less volatile, and many static analysis tools provide methods that eliminate the need to research each diagnostic on every invocation of the tool. The implementation of such a mechanism is encouraged, but not required.

Information Technology — Programming languages, their environments and system software interfaces — C Secure Coding Rules

1 Scope

This document specifies

— rules for secure coding in the C programming language and

— code examples.

This document 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.)

A set of code examples accompanies each rule in this document. Code examples are informative only and serve to clarify the requirements outlined in the normative portion of the rule. Examples impose no normative requirements.

Two distinct kinds of code examples are provided:

— code 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

— code examples rewritten to avoid such constructs; such examples are expected not to elicit a diagnostic.

Code examples are not intended to be complete programs. For the sake of 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.

Code examples are typically written in the form of one or more functions taking zero or more arguments. Unless explicitly specified elsewhere in an example, the values of arguments are considered to be external to a program (and obtained, for instance, as if by reading a file or an environment variable). If the code examples used values instead of arguments, the examples might no longer demonstrate a weakness as intended.

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.

A conforming analyzer shall diagnose all violations of coding rules specified in this Technical Specification. The guidelines may be extended in an implementation-dependent manner.

Conformance is evaluated by testing the ability of analyzers to diagnose all violations of the rules represented by the non-compliant code and not diagnose compliant code and exceptions in the whole program. Conforming analyzers shall diagnose transformations of these rules as required by Annex B (informative) Undefined Behavior.

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.

2.1 Completeness and soundness

To the greatest extent possible, an analyzer should be both complete and sound with respect to enforceable guidelines. An analyzer is considered sound (with respect to a specific guideline) if it does not give a false-negative result, meaning it is able to find all violations of a guideline within the entire program. An analyzer is considered complete if it does not issue false-positive results, or false alarms. The possibilities for a given guideline are outlined in Table 1.

Table 1 — Soundness and completeness

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

There are many tradeoffs in minimizing false positives and false negatives. It is obviously better to minimize both, and there are many techniques and algorithms that do both to some degree. However, once an analysis technology reaches the efficient frontier of what is possible without fundamental breakthroughs, it must select a point on the curve trading off these two factors (and others, such as scalability and automation). For automated tools on the efficient frontier that require minimal human input and that scale to large code bases, there is often tension between false negatives and false positives.

It is easy to build tools that are in the extremes. A tool can report all of the lines in the program and have no false negatives at the expense of large numbers of false positives. Conversely, a tool can report nothing and have no false positives at the expense of not reporting real defects that could be detected automatically. Tools with a high false positive rate waste the time of developers, who can lose interest in the results and, therefore, miss the true bugs that are lost in the noise. Tools with a high number of false negatives miss many defects that should be found. In practice, tools needs to strike a balance between the two.

The degree to which tools minimize false positive and false negative diagnostics is a quality of implementation issue. In other words, quantitative thresholds for false positive and false negative ratios are outside the scope of this Technical Specification.

Analyzers are trusted processes, meaning that developers rely upon their output. Consequently, developers must ensure that this trust is not misplaced. To earn this trust, the tool supplier should, ideally, run appropriate validation tests. While it is possible to use a validation suite to test an analyzer, no formal validation scheme exists at this time.

2.2 Portability assumptions

A conforming analyzer shall be able to diagnose violations of guidelines for at least one C99 or C1X implementation.

The term "C99" designates ISO/IEC 9899:1999 as corrected by its Technical Corrigenda; the term "C1X" designates the next revision of ISO/IEC 9899, currently being developed by ISO/IEC JTC 1/SC22/WG14.

An analyzer need not diagnose a rule violation if the result is documented for the target implementation and does not cause a security flaw.

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 might not be a problem for all target implementations, but it would be a portability problem for a target implementation where `int` does not have the same representation as `long`.

2.3 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 C99-conforming and C1X-conforming applications. To be considered a security flaw, a software bug must be triggered 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 required to distinguish violations that involve tainted data from those that do not involve tainted data.

2.4 Taint analysis

2.4.1 Taintedness and tainted sources

Some operations, particularly those on multiple operands, might have a defined domain that is a subset of the domain described by the types of their operands. When the actual operand values are outside of the defined domain, the result might be either undefined or at least unexpected. Examples include, for integral data types intended to be used as array indexes, values equal to or larger than the size of the array and, if the type is signed, any negative values. A string might not include a terminating null character within the character array that the string pointer points to or into. Strings might also violate a constraint on their contents imposed by a consumer, such as being a `printf` format string meant only for printing a fixed number of values. If the value of an expression might be outside the domain of some operation and it comes from somewhere outside of the program's control, such as a command line argument, data returned from a system call, or data in shared memory, that value is said to be *tainted* and its origin is known as a *tainted source*. Again, a tainted value is not necessarily known not to be in the domain; rather, it is not known to be in the domain. Note also that only values, not expressions, can be tainted; in some cases the same expression can hold tainted or untainted values along different paths.

Tainted sources are specifically

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

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

2.4.2 Taintedness sinks

Operations whose defined domain is a subset of the domain described by their operand types are called *taintedness sinks*. Any address arithmetic operation involving an integer operand is a taintedness sink for that operand. Certain parameters of certain library functions are defined to be taintedness sinks because internally those functions perform address arithmetic with those parameters, or control the allocation of a resource, or pass those parameters on to another taintededness sink. All string input parameters to library functions are taintedness sinks because those strings are required to be null-terminated, with the exception of `strncpy` (and `strncpy_s`), which explicitly allows the source argument to not be null-terminated. Although one could classify loop bounds as taintedness sinks, we choose not to do so.

2.4.3 Propagation

Generally speaking, taintedness 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 taintedness sinks. In addition to operations that propagate the same sort of taintedness, there are also operations that propagate taintedness of one sort of an operand to taintedness of a different sort for their results, the most notable example of which is `strlen` propagating the taintedness of its argument with respect to string length to the taintedness of its return value with respect to range.

2.4.4 Approaches to analysis

By definition, any tainted value flowing into a taintedness sink is a security issue, so all such cases must be diagnosed. Doing so requires some form of data flow analysis. In its most basic form, such an analysis would operate intraprocedurally to determine which local tainted sources flow into local taintedness sinks. Such a limited analysis would be very weak, so the next step would be interprocedural analysis. There are multiple ways to accomplish interprocedural analysis: top-down, bottom-up, and approaches that follow global data flow more than they follow the call graph. For example, in a bottom-up analysis, the parameters identified in the first step as flowing into taintedness sinks would themselves be treated as taintedness sinks at all of their function's call sites, recursively. In addition, function return values can be identified as tainted sources and treated accordingly at each call site. This brief sketch ignores such details as recursion and programs such as libraries with multiple call graph roots. It also ignores the large issue of tainted data escaping into the heap or into global or static variables.

2.4.5 Sanitization

For a tainted value to cease being tainted, something must be done to ensure that it is in the defined domain of any operation it flows into. This process is called *sanitization*. Sanitization can be done in two basic ways: replacement and termination. In sanitization by 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 sanitization by 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 via static analysis. Tools that do taint analysis usually provide some extra-linguistic 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 extra-linguistic mechanisms are outside the scope of this specification, we will instead use a set of rudimentary definitions of sanitization that will be very likely to recognize real sanitization but might cause non-sanitizing or ineffectively sanitizing code to be construed 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 taintedness sink if it restricts the range of that value to a subset of the defined domain of the operation at that sink. 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.

2.4.6 Tainted source macros

```
#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_INT(val)                \
  do {                                      \
    const char *taint = getenv("TAINT");    \
    if (taint == 0) {                       \
      exit(1);                              \
    }                                       \
                                            \
    val = strtol(taint, 0, 10);             \
  } while (0);
```

3 Normative references

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

[ISO/IEC 9899:1999] Programming Languages – C.

[ISO/IEC 9899:1999] Cor 1:2001, Programming Languages – C – Technical Corrigendum 1.

[ISO/IEC 9899:1999] Cor 2:2004, Programming Languages – C – Technical Corrigendum 2.

[ISO/IEC 9899:1999] Cor 3:2007, Programming Languages – C – Technical Corrigendum 3.

[ISO/IEC 9899-C1X] Committee Draft of upcoming revision of Programming Languages – C.

[ISO/IEC TR 24731-1:2007] Extensions to the C Library, Part I: Bounds-checking interfaces.

[ISO/IEC TR 24731-2] ISO/IEC TR 24731-2 Extensions to the C Library, Part II: Dynamic Allocation Functions.

[ISO 31-11:1992] Quantities and units – Part 11: Mathematical signs and symbols for use in the physical sciences and technology.

[ISO/IEC 646:1991] Information technology – ISO 7-bit coded character set for information interchange.

[ISO/IEC 2382-1:1993] Information technology – Vocabulary – Part 1: Fundamental terms.

[ISO 4217] Codes for the representation of currencies and funds.

[ISO 8601] Data elements and interchange formats – Information interchange – Representation of dates and times.

[ISO/IEC 10646:2003] (all parts), Information technology — Universal Multiple-Octet Coded Character Set (UCS).

[ISO/IEC/IEEE 60559] Information technology – Microprocessor Systems – Floating-Point arithmetic.

[IEC 61508] (all parts), Functional safety of electrical/electronic/programmable electronic safety-related systems.

[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 9899:1999, ISO/IEC 9899:201x, ISO/IEC 2382-1:1993, and the following entries apply. Other terms are defined where they appear in *italic* type. Mathematical symbols not defined in this Technical Specification are to be interpreted according to ISO 31-11:1992.

4.1
analyzer
The mechanism that diagnoses coding flaws in software programs. This may include static analysis tools, tools within a compiler suite, and code reviewers.

4.2
asynchronous-safe
A function is asynchronous-safe, or asynchronous-signal safe, if it can be called safely and without side effects from within a signal handler context. That is, it must be able to be interrupted at any point to run linearly out of sequence without causing an inconsistent state. It must also function properly when global data might itself be in an inconsistent state.

4.3
data flow
Data flow is the tracking of values along specific paths through the code. It can be done 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. There are varying degrees of sophistication of data flow analysis that 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.

4.4
dereferenceable pointer
A valid pointer that points to an object in memory.

4.5
derived type
Given an integer expression *E*, the derived type *T* of *E* is determined as follows:

— if *E* is a `sizeof` expression, then *T* is the type of the operand of the expression;

— otherwise, if *E* is an identifier, then *T* is the derived type of the expression last used to store a value in *E*;

— otherwise, if the derived type of each of *E*'s subexpressions is the same, then T is that type;

— otherwise, the derived type is an unspecified character type compatible with any of `char`, `signed char`, and `unsigned char`.

EXAMPLE For the following declarations:

```
double a[40];
size_t n0 = sizeof (int);
size_t n1 = 256;
size_t n2 = sizeof a / sizeof (*a);
```

The derived type of `n0` is `int`, and the derived type of `n1` and `n2` is a (hypothetical) unspecified character type that is compatible with any of `char`, `signed char`, and `unsigned char`.

4.6
exploit
A piece of software or a technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy.

4.7
invalid pointer
A pointer that is not a valid pointer.

4.8
non-dereferenceable pointer
A pointer that is not dereferenceable. The behavior of a program that attempts to use a non-dereferenceable pointer as an operand of the indirection operator `*` in a context where the pointer to an object is evaluated is undefined.

4.9
out-of-domain value
One of a set of values that is not in the domain of a particular operator or function.

4.10
persistent signal handler
A signal handler is persistent when it is running on a platform where the operating system reinstalls it each time it is called, meaning the programmer only has to install the handler once. A signal handler is non-persistent when it is running on a platform where it is not automatically reinstalled, meaning that the programmer has to reinstall the handler each time he or she wants to catch a signal.

4.11
sanitize
A value, typically one that previously was tainted, is said to be sanitized when testing has confirmed that it conforms to the constraints imposed by one or more taintedness sinks that it may flow into. One way or another, if the value does not conform, either the path will be diverted so as not to use the value or a different, known-conforming value will be substituted.

4.12
security flaw
A software defect that poses a potential security risk.

4.13
security policy
A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources.

4.14
static analysis
Any process for assessing code without executing it [Chess 2007, p. 3].

4.15
tainted data
A value is said to be *tainted* if it comes from an untrusted source (outside of the program's control) and has not been sanitized to ensure that it conforms to any constraints on its value that consumers of the value require, such as that a signed integer is non-negative or that a string is null terminated. Because different sinks for the same value may impose different validity constraints on it, the notions of taintedness and sanitization are not single-valued; a given value can be tainted with respect to one class of taintedness sinks but sanitized (and consequently no longer tainted) with respect to a different class of sinks.

4.16
taintedness sink
A *taintedness sink* is an expression that uses a value for some specific purpose, typically in a way that places some run-time constraint on that value necessary for the correctness of some operation. Problems occur when a value that is not known to conform to the given constraint, such as a tainted value, can flow into a taintedness sink.

4.17
untrusted data
Data originating from an untrusted source; for analysis purposes, any input external to the program.

4.18
valid pointer
A pointer that refers to an element within an array or one past the last element of an array. 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 C99, Section 6.5.8 p. 4.)

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 C99, Section 6.3.2.3 p. 7.)

4.19
vulnerability
A 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 (EXP11-C, EXP39-C)

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.

Accessing an object through a pointer to an incompatible type (other than `unsigned char`) is undefined behavior (see undefined behavior 34 in Annex B).

According to Section 6.2.6.1 of C99,

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.

EXAMPLE 1 In this example, a diagnostic is required because an object of type `float` is incremented through a pointer to `int`, `ip`.

```
void f() {
  assert(sizeof(int) == sizeof(float));

  float f = 0.0;
  int *ip = (int *)&f;

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

  (*ip)++;  // diagnostic required

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

EXAMPLE 2 In this example, a diagnostic is required because the second and third arguments to the conditional operator, `&u.i` and `&u.f`, have incompatible types.

```
void g(int which) {
  void *p;
  int j;
  union {
    int i;
    float f;
  } u;

  u.i = 1;
  p = which ? &u.i : &u.f;  // diagnostic required
  if (which == 0) {
    j = *(int *)p;
  }
}
```

Related guidelines

CERT C Secure Coding Standard:

— EXP11-C. Do not apply operators expecting one type to data of an incompatible type

— EXP39-C. Do not access a variable through a pointer of an incompatible type

ISO/IEC TR 24772 "STR Bit Representations"

MISRA-C 2004, Rule 3.5

Bibliography

[Plum 1985] Rule 6-5

5.2 Accessing freed memory (MEM30-C)

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.

C99 identifies the situation in which undefined behavior (UB) arises as a result of accessing freed memory:

UB **Description**

168 *The value of a pointer that refers to space deallocated by a call to the free or realloc function is used (7.20.3).*

EXAMPLE 1 In this 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 example, a diagnostic is required because `buf` is written to after it has been freed.

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

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

  char *buf = (char *)malloc(bufsize);
  if (!buf) {
     /* ... */
  }
  /* ... */
  free(buf);
  /* ... */
  strncpy(buf, argv[1], bufsize);  // diagnostic required
  /* ... */
  return 0;
```

EXAMPLE 3 In this example, a diagnostic is required because `realloc` may free `str1` when it returns `NULL`, resulting in `str1` being freed twice.

```
void f(char *str1, size_t size) {
  char *str2 = (char *)realloc(str1, size);
  if (str2 == NULL) {
    free(str1); // diagnostic required
    return;
  }
}
```

Related guidelines

CERT C Secure Coding Standard: MEM30-C. Do not access freed memory

ISO/IEC TR 24772 "DCM Dangling references to stack frames" and "XYK Dangling Reference to Heap"

MISRA-C 2004, Rule 17.6

MITRE CWE: CWE-416: Use After Free

Bibliography

[Kernighan 1988] Section 7.8.5, "Storage Management"

[OWASP] Freed Memory

[Seacord 2005] Chapter 4, "Dynamic Memory Management"

[Viega 2005] Section 5.2.19, "Using freed memory"

5.3 Accessing shared objects in signal handlers (SIG31-C)

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 because this results in undefined behavior.

EXAMPLE In this 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`.

```
char *err_msg;
enum { MAX_MSG_SIZE = 24 };

void handler(int signum) {
  strcpy(err_msg, "SIGINT encountered.");  // diagnostic required
}

int main(void) {
  signal(SIGINT, handler);

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

  /* Main code loop */

  return 0;
}
```

Related guidelines

CERT C Secure Coding Standard: SIG31-C. Do not access or modify shared objects in signal handlers

ISO/IEC 2003 "Signals and Interrupts"

MITRE CWE: CWE-662: Improper Synchronization

Bibliography

[Dowd 2006] Chapter 13, Synchronization and State

[Open Group 2004] `longjmp`

[OpenBSD] `signal` Man Page

[Zalewski 2001]

5.4 Accessing volatile objects through a non-volatile pointer (EXP32-C)

Accessing a volatile object through a non-volatile pointer shall be diagnosed because this results in undefined behavior.

EXAMPLE In this example, a diagnostic is required because a `volatile`-qualified object is accessed through a non-`volatile`-qualified pointer, `ip`.

```
static volatile int **ipp;
static int *ip;
static volatile int i = 0;

void f() {
  ipp = &ip;
  *ipp = &i;
  if (*ip != 0) {  // diagnostic required
    /* ... */
  }
}
```

Related guidelines

CERT C Secure Coding Standard: EXP32-C. Do not access a volatile object through a non-volatile reference

ISO/IEC TR 24772 "HFC Pointer casting and pointer type changes" and "IHN Type system"

MISRA-C 2004, Rule 11.5

5.5 Adding or subtracting a byte count integer to an element pointer (EXP08-C)

A byte count integer counts some number of bytes. An element count integer counts some number of array elements. An element pointer points to an array that contains elements whose sizes are each greater than one byte.

Adding or subtracting a byte count integer value to a pointer shall be diagnosed because this can result in an invalid or incorrect pointer. Similarly, dividing the difference of two element pointers (which yields an element count integer) by the size of the type that the pointers point to shall be diagnosed because this operation yields a dimensionally nonsensical value. In both cases the resulting value may be either larger or smaller than the intended value, depending on the types involved, and so may lead to either overruns or underruns, depending on how the value is used.

The quotient of the size of an entire array object, which is itself a byte count integer, and the size of a single element of that same array, which is also a byte count integer, is an element count integer, so it is allowed to add or subtract such a quotient to or from an element pointer.

EXAMPLE 1 In this example, a diagnostic is required because the byte count integer `sizeof(buf)` is added to the element pointer `buf`.

```
int get_int(int *data) {
  char buf[BUFSIZ];
  if (fgets(buf, BUFSIZ, stdin) == NULL) {
    return 1;
  }

  *data = strtol(buf, NULL, 0);

  return 0;
}

void collect_ints() {
  int buf[BUFSIZ];
  int *buf_ptr = buf;

  while (buf_ptr < (buf + sizeof(buf))) {  // diagnostic required
    int data;
    if (get_int(&data) != 0) {
      break;
    }
    *buf_ptr++ = data;
  }

  /* ... */
}
```

EXAMPLE 2 In this example, a diagnostic is required because the element pointer `end - begin` is divided by the size of the pointed-to type, `sizeof(struct s)`.

```
struct s {
  int a;
  int b;
};

void f(struct s *begin, struct s *end) {
  size_t nelem = (end - begin) / sizeof(struct s);  // diagnostic required
  size_t size = nelem * sizeof(struct s);

  struct s *s_copy = (struct s *)malloc(size);
  if (!s_copy) {
    /* ... */
  }

  memcpy(s_copy, begin, size);

  /* ... */
}
```

EXAMPLE 3 In this example, a diagnostic is required because the byte count integer `skip` is added to the element pointer `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() {
  size_t skip = offsetof(struct big, ull_2);
  struct big *s = (struct big *)malloc(99 * sizeof(struct big));
  if (!s) {
    /* ... */
  }

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

  /* ... */
}
```

EXAMPLE 4 In this example, a diagnostic is required because the byte count integer `wcslen(error_msg) * sizeof(wchar_t)` is added to the element pointer `error_msg`.

```
void h() {
  wchar_t error_msg[BUFSIZ];

  const wchar_t *prefix = L"Error: ";

  wcscpy(error_msg, prefix);
  fgetws(error_msg + wcslen(error_msg) * sizeof(wchar_t),  // diagnostic required
    BUFSIZ - wcslen(prefix), stdin);

  /* ... */
}
```

Exception: EXP08-EX1

No diagnostic is required if the pointer is a pointer to `char`.

Related guidelines

CERT C Secure Coding Standard: EXP08-C. Ensure pointer arithmetic is used correctly

ISO/IEC TR 24772 "HFC Pointer casting and pointer type changes" and "RVG Pointer Arithmetic"

MITRE CWE: CWE-468: Incorrect Pointer Scaling

MISRA-C 2004, Rules 17.1-17.4

Bibliography

[Dowd 2006] Chapter 6, "C Language Issues"

[Seacord 2005] Seacord, Robert C. *Secure Coding in C and C++*.

5.6 Assigning in conditional expressions (EXP18-C)

Using the assignment operator in the outermost expression of a conditional expression shall be diagnosed because this typically indicates programmer error and can result in unexpected behavior.

EXAMPLE In this example, a diagnostic is required because the assignment expression is the outermost expression of a conditional expression.

```
if (a = b) {   // diagnostic required
 /* ... */
}
```

Bibliography

[Hatton 1995] Section 2.7.2, "Errors of omission and addition"

[ISO/IEC TR 24772] "KOA Likely Incorrect Expressions"

[MITRE 2007] CWE ID 482, "Comparing instead of Assigning," CWE ID 480, "Use of Incorrect Operator"

[CERT C Secure Coding Standard] "EXP18-C. Do not perform assignments in selection statements"

5.7 Assigning in controlling expressions (EXP15-C)

Assigning a constant in the outermost expression of the controlling expression of the following shall be diagnosed because this typically indicates programmer error and can result in unexpected behavior:

— `if`

— `switch`

— `while`

- `do`
- `for`
- `?:`

EXAMPLE In this example, a diagnostic is required because the expression `a = 42` contains an assignment of a constant in the outermost expression of the controlling expression of an `if` statement.

```
void f(int a) {
  if (a = 42) {  // diagnostic required
    /* ... */
  }

  /* ... */
}
```

Related guidelines

CERT C Secure Coding Standard: MSC02-C. Avoid errors of omission

ISO/IEC TR 24772 "KOA Likely Incorrect Expressions"

MITRE CWE:

- CWE-480: Use of Incorrect Operator
- CWE-482: Comparing instead of Assigning

Bibliography

[Hatton 1995] Section 2.7.2, "Errors of omission and addition"

5.8 Assuming a positive remainder when using the % operator (INT10-C)

Assuming a positive remainder when using the `%` (modulo) operator shall be diagnosed because the remainder from the `%` operator can be non-positive.

EXAMPLE In this example, a diagnostic is required because the result of the expression `(index + 1) % size` can be negative and is used as an index to the array `list`.

```
int insert(int index, int *list, int size, int value) {
  if (size != 0) {
    index = (index + 1) % size;
    list[index] = value;  // diagnostic required
    return index;
  } else {
    return -1;
  }
}
```

Related guidelines

CERT C Secure Coding Standard: INT10-C. Do not assume a positive remainder when using the % operator

MITRE CWE:

- CWE-129: Improper Validation of Array Index

— CWE-682: Incorrect Calculation

Bibliography

[Beebe 2005] Re: Remainder (%) operator and GCC

[Microsoft 2007] C Multiplicative Operators

[Sun 2005] Appendix E, "Implementation-Defined ISO/IEC C90 Behavior"

5.9 Assuming character data does not contain a null byte (FIO37-C)

Assuming a non-zero number of non-null bytes stored by the `fgets` and `fread` functions into the initial elements of the array shall be diagnosed because these functions can return character data that contains null bytes.

EXAMPLE 1 In this 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() {
  char buf[BUFSIZ];

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

EXAMPLE 2 In this compliant example, a diagnostic is not required because no assumption is made about the first byte of the string `buf` being non-null.

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

  if (fgets(buf, sizeof(buf), stdin)) {
    char *nl = strchr(buf, '\n');
    if (nl) {
      *nl = '\0';
    }

    puts(buf);
  }
}
```

Related guidelines

CERT C Secure Coding Standard: FIO37-C. Do not assume that fgets() returns a nonempty string when successful

MITRE CWE:

— CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

— CWE-241: Improper Handling of Unexpected Data Type

Bibliography

[Lai 2006] "Reading Between the Lines"

[Seacord 2005] Chapter 2, "Strings"

5.10 atexit-registered handler does not return (ENV32-C)

A handler registered using the `atexit` or `at_quick_exit` function that makes a call to the `longjmp` function that would terminate the call to the handler shall be diagnosed because the behavior of such a handler is undefined.

A handler registered using the `atexit` function that makes a call to the `exit` function, or a handler registered using the `at_quick_exit` function that makes a call to the `at_quick_exit` function, shall be diagnosed because the behavior of such a handler is undefined.

EXAMPLE 1 In this example, a diagnostic is required because the `atexit`-registered handler `exit_handler` does not return.

```
void exit_handler(void) {
  exit(0);  // diagnostic required
}

int main(void) {
  atexit(exit_handler);

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

EXAMPLE 2 In this example, a diagnostic is required because the `at_quick_exit`-registered handler `quick_exit_handler` does not return.

```
jmp_buf env;

void quick_exit_handler(void) {
  longjmp(env, 1);  // diagnostic required
}

int main(void) {
  if (at_quick_exit(quick_exit_handler) == 0) {
    if (setjmp(env) == 0) {
      quick_exit(0);
    }
  }

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

Related guidelines

CERT C Secure Coding Standard: ENV32-C. All atexit handlers must return normally

ISO/IEC TR 24772 "EWD Structured Programming" and "REU Termination Strategy"

MITRE CWE: CWE-705: Incorrect Control Flow Scoping

5.11 Calling functions in the C standard library other than abort, _Exit, and signal from within a signal handler (SIG30-C)

Calling functions in the C standard library other than `abort`, `_Exit`, and `signal` from within a signal handler shall be diagnosed because doing so results in undefined behavior.

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

```
enum { MAXLINE = 1024 };
char info[MAXLINE];

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

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

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

  while (1) {
    /* Main loop program code */

    log_message();

    /* More program code */
  }
  return 0;
}
```

EXAMPLE 2 In this 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 0;
}
```

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

```
enum { 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 0;
}
```

Exception: SIG30-EX1

A signal handler that does not occur as a result of calling the `abort` or `raise` function does not need to be diagnosed because this does not result in undefined behavior.

Related guidelines

CERT C Secure Coding Standard:

— SIG30-C. Call only asynchronous-safe functions within signal handlers

— SIG33-C. Do not recursively invoke the raise() function

ISO/IEC 2003 Section 5.2.3, "Signals and interrupts"

MITRE CWE: CWE-479: Signal Handler Use of a Non-reentrant Function

Bibliography

[Dowd 2006] Chapter 13, "Synchronization and State"

[Open Group 2004] `longjmp`

[OpenBSD] `signal` Manual Page

[Zalewski 2001] Delivering Signals for Fun and Profit

5.12 Calling functions with incorrect arguments (EXP37-C)

Calling a function with the wrong number or type of arguments shall be diagnosed because this results in undefined behavior.

C99 identifies three distinct situations in which undefined behavior (UB) 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
23	*A pointer is used to call a function whose type is not compatible with the pointed-to type (6.3.2.3).*
36	*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).*
37	*For 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).*
39	*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 1 In this 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() {
  char *c;
  fp = strchr;
  c = fp(12, 2);  // diagnostic required
}
```

EXAMPLE 2 In this 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) {
  strcpy(dst, src);
}

/* 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 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 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
}
```

Related guidelines

CERT C Secure Coding Standard: EXP37-C. Call functions with the arguments intended by the API

ISO/IEC TR 24772 "OTR Subprogram Signature Mismatch"

MISRA-C 2004, Rule 16.6

MITRE CWE: CWE-628: Function Call with Incorrectly Specified Arguments

Bibliography

[MITRE 2011] CVE-2006-1174

[Spinellis 2006] Section 2.6.1, "Incorrect Routine or Arguments"

5.13 Calling signal from interruptible signal handlers (SIG34-C)

Calling `signal` from within an interruptible signal handler on platforms where `signal` handlers are non-persistent shall be diagnosed because doing so presents a race window.

EXAMPLE In this example, a diagnostic is required on implementations where signal handlers are non-persistent 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() {
  if (signal(SIGUSR1, handler) == SIG_ERR) {
    /* ... */
  }

  /* ... */
}
```

Related guidelines

CERT C Secure Coding Standard: SIG34-C. Do not call signal() from within interruptible signal handlers

MITRE CWE: CWE-479: Signal Handler Use of a Non-reentrant Function

5.14 Calling system (ENV04-C)

All calls to the `system` function shall be diagnosed. Use of the `system` function can result in exploitable vulnerabilities

— when passing an unsanitized or improperly sanitized command-string originating from an untrusted 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, these can only be identified on a case-by-case basis during a code review and are, consequently, outside the scope of this guideline.

EXAMPLE 1 In this example, a diagnostic is required because a string consisting of `any_cmd` and untrusted data 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 example, a diagnostic is required because `system` is used to remove the `.config` file in the user's home directory.

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

Related guidelines

CERT C Secure Coding Standard: ENV04-C. Do not call system() if you do not need a command processor

ISO/IEC TR 24772 "XZQ Unquoted Search Path or Element"

MITRE CWE:

— CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

— CWE-88: Argument Injection or Modification

Bibliography

[Open Group 2004] `environ`, `execl`, `execv`, `execle`, `execve`, `execlp`, `execvp` — execute a file, `popen`, `unlink`, XCU Section 2.8.2, "Exit Status for Commands"

[Wheeler 2004] Secure programmer: Call components safely.

5.15 Comparing function addresses to zero (EXP18-C)

Comparing, either explicitly or implicitly, an expression taking the address of a function to a constant zero that is implicitly converted to a function pointer shall be diagnosed because this typically indicates programmer error and can result in unexpected behavior. If such a comparison is intentional, this intention can be made explicit by explicitly casting 0 or NULL to the appropriate pointer type before using it in the comparison (which may require making the comparison itself explicit, too).

EXAMPLE 1 In this example, a diagnostic is required because the addresses of the functions `getuid` and `geteuid` are compared to `0`.

```
void f() {
  if (getuid == 0  // diagnostic required
    || geteuid != 0) {  // diagnostic required
    /* ... */
  }

  /* ... */
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because the addresses of the functions `getuid` and `geteuid` are compared to null function pointers of the same type.

```
typedef uid_t (*getuid_t)(void);

void f() {
  if (getuid == (getuid_t)0
    || geteuid == (getuid_t)0) {
    /* ... */
  }

  /* ... */
}
```

EXAMPLE 3 In this example, a diagnostic is required because the address of the function `do_xyz` is compared to `0`.

```
int do_xyz(void);

void g() {
  if (do_xyz) {  // diagnostic required
    /* ... */
  }

  /* ... */
}
```

EXAMPLE 4 In this compliant example, a diagnostic is not required because the address of the function `do_xyz` is compared to a null function pointer of the same type.

```
int do_xyz(void);
typedef int (*do_xyz_t)(void);

void g() {
  if (do_xyz == (do_xyz_t)0) {
    /* ... */
  }

  /* ... */
}
```

Related guidelines

CERT C Secure Coding Standard: EXP16-C. Do not compare function pointers to constant values

ISO/IEC TR 24772 "KOA Likely Incorrect Expressions"

Bibliography

[Hatton 1995] Section 2.7.2, "Errors of omission and addition"

5.16 Comparing or assigning expressions to a larger size objects (INT35-C)

An *assignment-expression* (including initialization) whose right operand has an unsigned integer or floating type smaller than that of the left operand and that is not a primary expression shall be diagnosed because the smaller expression may either wrap (if it is of unsigned integer type) or silently overflow (if it is of floating type).

A *relational-expression* or *equality-expression*, one of whose operands has an unsigned integer or floating type smaller than that of the other operand and that is not a primary expression, shall be diagnosed for the same reason as above.

NOTE Expressions involving signed integer types are not subject to these requirements because signed integer overflow is the subject of Signed integer overflow (INT32-C).

EXAMPLE 1 In this example, a diagnostic is required because the unsigned integer expression `x + 1` is assigned to the larger size variable `y`.

```
void f(unsigned x) {
#if UINT_MAX < ULONG_MAX
  unsigned long y = x + 1;
#elif UINT_MAX < ULLONG_MAX
  unsigned long long y = x + 1;
#else
  uintmax_t y = x + 1;   // diagnostic required
#endif

  printf("x + 1 = %jd\n", (uintmax_t)y);
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because the unsigned integer expressions `x + 1UL`, `x + 1ULL`, and `x + (uintmax_t)1` have the same size as the corresponding variables they are assigned to.

```
void f(unsigned x) {
#if UINT_MAX < ULONG_MAX
  unsigned long y = x + 1UL;
#elif UINT_MAX < ULLONG_MAX
  unsigned long long y = x + 1ULL;
#else
  assert(x < UINTMAX_MAX);
  uintmax_t y = x + (uintmax_t)1;
#endif

  printf("x + 1 = %jd\n", (uintmax_t)y);
}
```

EXAMPLE 3 In this example, a diagnostic is required because the unsigned integer expression `length + BLOCK_HEADER_SIZE` is compared to the larger size integer expression `(unsigned long long)SIZE_MAX`.

```
enum { BLOCK_HEADER_SIZE = 16 };

void *AllocateBlock(unsigned long length) {
  struct memBlock *mBlock;

  if (length + BLOCK_HEADER_SIZE > (unsigned long long)SIZE_MAX) {   //
diagnostic required
    return NULL;
  }
  mBlock = (struct memBlock *)malloc(
    length + BLOCK_HEADER_SIZE
  );
  if (!mBlock) {
    return NULL;
  }

  /* ... */

  return mBlock;
}
```

EXAMPLE 4 In this compliant example, a diagnostic is not required because the unsigned integer expression `(unsigned long long)length + BLOCK_HEADER_SIZE` is compared to the same size integer expression `(unsigned long long)SIZE_MAX`.

```
enum { BLOCK_HEADER_SIZE = 16 };

void *AllocateBlock(unsigned long length) {
  struct memBlock *mBlock;

  assert((unsigned long long)length <= ULLONG_MAX - BLOCK_HEADER_SIZE);
  if ((unsigned long long)length + BLOCK_HEADER_SIZE > (unsigned long
long)SIZE_MAX) {
    return NULL;
  }
  mBlock = (struct memBlock *)malloc(
    length + BLOCK_HEADER_SIZE
  );
  if (!mBlock) {
    return NULL;
  }

  /* ... */

  return mBlock;
}
```

EXAMPLE 5 In this example, a diagnostic is required because the unsigned integer expression `cBlocks * 16` is assigned to the larger size variable `alloc`.

```
void *AllocBlocks(unsigned long cBlocks) {
  if (cBlocks == 0){
    return NULL;
  }

  unsigned long long alloc = cBlocks * 16;   // diagnostic required
  if (alloc < UINT_MAX) {
    return malloc(cBlocks * 16);
  } else {
    return NULL;
  }
}
```

EXAMPLE 6 In this compliant example, a diagnostic is not required because the unsigned integer expression `cBlocks * 16ULL` is assigned to the same size variable `alloc`.

```
void *AllocBlocks(unsigned long cBlocks) {
  if (cBlocks == 0) {
    return NULL;
  }

  assert((unsigned long long)cBlocks <= ULLONG_MAX / 16);
  unsigned long long alloc = cBlocks * 16ULL;

  if (alloc < UINT_MAX) {
    return malloc(cBlocks * 16);
  } else {
    return NULL;
  }
}
```

EXAMPLE 7 In this example, a diagnostic is required because the unsigned integer expression `x * 1024` is implicitly converted to the larger type `double`, and the expression may wrap.

```
double g(unsigned int x) {
  return x * 1024;  // diagnostic required
}
```

EXAMPLE 8 In this compliant example, a diagnostic is not required because the type of the expression `x * 1024.0` is the same as the return type of the function.

```
double g(unsigned int x) {
  return x * 1024.0;
}
```

Related guidelines

CERT C Secure Coding Standard: INT35-C. Evaluate integer expressions in a larger size before comparing or assigning to that size

ISO/IEC TR 24772 "FLC Numeric Conversion Errors"

MITRE CWE:

— CWE-190: Integer Overflow or Wraparound

— CWE-681: Incorrect Conversion between Numeric Types

Bibliography

[Dowd 2006] Chapter 6, "C Language Issues"

[Seacord 2005] Chapter 5, "Integer Security"

5.17 Comparison of padding data (EXP04-C)

Comparison of padding data shall be diagnosed because the value of padding bits is unspecified and may contain data initially provided by an attacker.

EXAMPLE In this 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;
}
```

Related guidelines

CERT C Secure Coding Standard: EXP04-C. Do not perform byte-by-byte comparisons involving a structure

Bibliography

[Dowd 2006] Chapter 6, "C Language Issues" (Structure Padding 284-287)

[Kernighan 1988] Chapter 6, "Structures" (Structures and Functions 129)

[Summit 1995] Question 2.8, Question 2.12

5.18 Converting a pointer to integer or integer to pointer (INT11-C)

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.

EXAMPLE 1 In this example, a diagnostic is required because the pointer `ptr` is converted to an integer and the integer `number` is converted to a pointer.

```
void f() {
  char *ptr;
  unsigned int flag;
  /* ... */
  unsigned int number = (unsigned int)ptr;  // diagnostic required
  number = (number & 0x7fffff) | (flag << 23);
  ptr = (char *)number;  // diagnostic required
}
```

EXAMPLE 2 In this example, a diagnostic is required because the integer literal `0xdeadbeef` is converted to a pointer.

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

Exceptions

— INT11-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.

— INT11-EX2: Any valid pointer 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`.)

```
void h() {
  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);
}
```

Related guidelines

CERT C Secure Coding Standard: INT11-C. Take care when converting from pointer to integer or integer to pointer

ISO/IEC TR 24772 "HFC Pointer casting and pointer type changes"

MITRE CWE:

— CWE-466: Return of Pointer Value Outside of Expected Range

— CWE-587: Assignment of a Fixed Address to a Pointer

5.19 Converting floating point values to types that cannot represent their value (FLP34-C)

Floating point conversions that are not within range of the new type shall be diagnosed because this results in an unspecified value.

If the value being converted is outside the range of values that can be represented, the behavior is undefined (C99 Section 6.3.1.5).

This is overridden by Annex F.

EXAMPLE 1 In this example, a diagnostic is required because the integral part of `f` may not be within the range of the new type, `int`.

```
int f(float f) {
  int i = f;  // diagnostic required

  /* ... */
  return i;
}
```

EXAMPLE 2 In this example, a diagnostic is required because the conversions of the longer floating-point types may not be within the range of the smaller types.

```
void g(long double ld, double d1) {
  float f1 = (float)d1;  // diagnostic required
  float f2 = (float)ld;  // diagnostic required
  double d2 = (double)ld;  // diagnostic required

  /* ... */
  printf("%f %f %f\n", f1, f2, d2);
}
```

Related guidelines

CERT C Secure Coding Standard: FLP34-C. Ensure that floating point conversions are within range of the new type

ISO/IEC TR 24772 "FLC Numeric Conversion Errors"

MITRE CWE: CWE-681: Incorrect Conversion between Numeric Types

Bibliography

[IEEE 754: 2006] IEEE 754-1985 Standard for Binary Floating-Point Arithmetic

5.20 Converting integer to a type that is unable to represent its value (INT31-C)

Conversion of a tainted, potentially mutilated, or out-of-domain integer value to a signed integer type, when the possible range of values cannot be represented in the destination type, shall be diagnosed.

EXAMPLE 1 In this example, a diagnostic is required because the value of the variable `z` may not be representable when it is returned from the function.

```
int square() {
#if INT_MAX < LONG_MAX
   typedef long wider_type;
#else
   typedef long long wider_type;
#endif
```

```
  int x;
  GET_TAINTED_INT(x);

  wider_type z = (wider_type)x * x;
  return z;  // diagnostic required
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because the value of the variable z is always representable when it is returned from the function.

```
int square() {
#if INT_MAX < LONG_MAX
   typedef long wider_type;
#else
   typedef long long wider_type;
#endif

  int x;
  GET_TAINTED_INT(x);

  wider_type z = (wider_type)x * x;

  if (INT_MIN <= z && z <= INT_MAX) {
    return z;
  } else {
    return 0;
  }
}
```

Related guidelines

CERT C Secure Coding Standard: INT15-C. Use intmax_t or uintmax_t for formatted IO on programmer-defined integer types

CERT C Secure Coding Standard: INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data

ISO/IEC TR 24772 "XYY Wrap-around Error"

MITRE CWE: CWE-190: Integer Overflow or Wraparound

Bibliography

[Dowd 2006] Chapter 6, "C Language Issues" (Arithmetic Boundary Conditions, pp. 211-223)

[Seacord 2005] Chapter 5, "Integers"

[Viega 2005] Section 5.2.7, "Integer overflow"

[VU#551436] Mozilla Firefox SVG viewer vulnerable to buffer overflow

[Warren 2002] Chapter 2, "Basics"

5.21 Converting pointer values to more strictly aligned pointer types (EXP36-C)

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

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

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

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

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

```
void f() {
  char *char_ptr;
  int *int_ptr;
  int i;

  char_ptr = (char *)&i;
  int_ptr = (int *)char_ptr;
  /* ... */
}
```

Related guidelines

CERT C Secure Coding Standard: EXP36-C. Do not convert pointers into more strictly aligned pointer types

ISO/IEC TR 24772 "HFC Pointer casting and pointer type changes"

MISRA-C 2004, Rules 11.2 and 11.3

Bibliography

[Bryant 2003] Computer Systems: A Programmer's Perspective.

5.22 Copying a FILE object (FIO38-C)

Copying a `FILE` object shall be diagnosed because the copy does not need to be safe to be used as an argument to any I/O function.

According to C99, Section 7.19.3, 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 In this 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 0;
}
```

Related guidelines

CERT C Secure Coding Standard: FIO38-C. Do not use a copy of a FILE object for input and output

5.23 Declaring an identifier with conflicting linkage classifications (DCL36-C)

An identifier with conflicting linkage classifications shall be diagnosed because referencing it results in undefined behavior.

EXAMPLE 1 In this example, a diagnostic is required because the identifiers `i2` and `i5` are defined as having both internal and external linkage.

```
static int i1 = 10;
static int i2;

int i1;  // diagnostic required
int i2;  // diagnostic required

int main(void) {
  /* ... */
  return 0;
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because no identifiers are defined as having both internal and external linkage.

```
int i1 = 10;
int i2;
extern int i3 = 30;

int i1;
int i2;
int i3;

int main(void) {
  /* ... */
  return 0;
}
```

Related guidelines

CERT C Secure Coding Standard: DCL36-C. Do not declare an identifier with conflicting linkage classifications

MISRA-C 2004, Rule 8.1

Bibliography

[Banahan 2003] Section 8.2, "Declarations, Definitions and Accessibility"

[Kirch-Prinz 2002] C Pocket Reference.

5.24 Declaring the same function or object in incompatible ways (ARR31)

Two or more incompatible declarations of the same function or object that appear in the same program shall be diagnosed because they result in undefined behavior.

C99 identifies three distinct situations in which undefined behavior (UB) may arise as a result of incompatible declarations of the same function or object:

UB	Description
14	*Two declarations of the same object or function specify types that are not compatible (6.2.7).*

UB **Description**

34 *An object has its stored value accessed other than by an lvalue of an allowable type (6.5).*

39 *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 effect 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 1 In this 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 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 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) {
  return h(10);
}

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

EXAMPLE 4 In this example, a diagnostic is required on implementations where limitations cause the external identifiers `bash_groupname_completion_function` and `bash_groupname_completion_func` to be identical, because this results in incompatible declarations.

```
/* in bash/bashline.h */
extern char* bash_groupname_completion_function(const char*, int);  // diagnostic
required
```

```
/* 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_func;  // diagnostic required
```

NOTE 1 The identifier `bash_groupname_completion_function` referenced above was taken from GNU Bash version 3.2.

NOTE 2 Rule Using non-unique identifiers (DCL32-C) applies to multiple identifiers of the same type with non-unique names.

Exception: ARR31-EX1

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 possibly additional declarations.

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

/* b.c: */
extern char x; /* incompatible declaration */
/* but no other references to 'x' */
```

Related guidelines

CERT C Secure Coding Standard: ARR31-C. Use consistent array notation across all source files

Bibliography

[Hatton 1995] Section 2.8.3

5.25 Dereferencing a null pointer (EXP34-C)

Dereferencing a tainted or out-of-domain pointer shall be diagnosed because, if such a pointer is null, doing so results in undefined behavior.

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

```
void f(const char *input_str) {
  size_t size = strlen(input_str) + 1;
  char *str = (char *)malloc(size);
  memcpy(str, input_str, size);  // diagnostic required

  /* ... */
  free(str);
  str = NULL;
}
```

Related guidelines

CERT C Secure Coding Standard: EXP34-C. Do not dereference null pointers

ISO/IEC TR 24772 "HFC Pointer casting and pointer type changes" and "XYH Null Pointer Dereference"

MITRE CWE: CWE-476: NULL Pointer Dereference

Bibliography

[Jack 2007] Vector Rewrite Attack.

[van Sprundel 2006] Unusualbugs.

[Viega 2005] Section 5.2.18, "Null-pointer dereference"

5.26 Dividing by zero (INT33-C)

Tainted, potentially mutilated, and out-of-domain values that are used as the second operand to the `/` operator or the `%` operator shall be diagnosed because they may result in divide-by-zero errors and undefined behavior.

EXAMPLE 1 In this example, a diagnostic is required because the expression `x / y` can result in a divide-by-zero error.

```
int divide(int x) {
  int y;
  GET_TAINTED_INT(y);

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

EXAMPLE 2 In this example, a diagnostic is required because the expression `x % y` can result in a divide-by-zero error.

```
int modulus(int x) {
  int y;
  GET_TAINTED_INT(y);

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

Related guidelines

CERT C Secure Coding Standard: INT33-C. Ensure that division and modulo operations do not result in divide-by-zero errors

MITRE CWE: CWE-369: Divide By Zero

Bibliography

[Seacord 2005] Chapter 5, "Integers"

[Warren 2002] Chapter 2, "Basics"

5.27 Escaping of the address of an automatic object (DCL30-C)

The address of an object with automatic storage duration shall not be 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.

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

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

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

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

  return 0;
}
```

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

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

EXAMPLE 3 In this 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 0;
}
```

Related guidelines

CERT C Secure Coding Standard: DCL30-C. Declare objects with appropriate storage durations

ISO/IEC TR 24772 "DCM Dangling references to stack frames"

MISRA-C 2004, Rule 8.6

Bibliography

[Coverity 2007] Coverity Prevent User’s Manual (3.3.0)

5.28 Conversion of signed characters to wider integer types (STR34-C)

Converting character data of type `char` or `signed char` to a larger integer type without having first cast the variable to `unsigned char` shall be diagnosed because this can result in unexpected behavior.

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

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

  if (string && *string) {
    c = *string++;  // diagnostic required
  }

  return c;
}
```

Related guidelines

CERT C Secure Coding Standard: STR34-C. Cast characters to unsigned char before converting to larger integer sizes

MISRA-C 2004, Rule 6.1, "The plain char type shall be used only for the storage and use of character values."

MITRE CWE: CWE-704: Incorrect Type Conversion or Cast

5.29 Use of an implied default in a switch statement (MSC01-C)

Not checking all possible data paths in a `switch` statement shall be diagnosed because doing so can result in unexpected behavior.

EXAMPLE In this 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;
  }
}
```

Related guidelines

CERT C Secure Coding Standard: MSC01-C. Strive for logical completeness

ISO/IEC TR 24772 "CLL Switch statements and static analysis"

Bibliography

[Hatton 1995] Section 2.7.2, "Errors of omission and addition"

[Viega 2005] Section 5.2.17, "Failure to account for default case in switch"

5.30 Failing to close files or free dynamic memory when they are no longer needed (FIO42-C)

Failing to close files or to free dynamic memory allocations when they are no longer needed shall be diagnosed because doing so can result in the exhaustion and manipulation of system resources.

EXAMPLE 1 In this example, a diagnostic is required because the file `f` is not closed before the call to `system`, which may spawn a child process and give it access to the opened file.

```
void f(const char *filename) {
  FILE *f = NULL;
  const char *editor = NULL;

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

  /* ... */
  editor = getenv("EDITOR");
  if (editor == NULL) {
    /* ... */
  }

  if (system(editor) == -1) {  // diagnostic required
    /* ... */
  }

  /* ... */
}
```

EXAMPLE 2 In this example, a diagnostic is required because sensitive information is stored in dynamic memory and the pointer to that memory is lost on the second call to `malloc`.

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

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

  char *text_buffer = (char *)malloc(BUFSIZ);
  fgets(text_buffer, BUFSIZ, f);

  /* ... */
  text_buffer = (char *)malloc(BUFSIZ);  // diagnostic required

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

Related guidelines

CERT C Secure Coding Standard: FIO42-C. Ensure files are properly closed when they are no longer needed

MITRE CWE:

— CWE-403: Exposure of File Descriptor to Unintended Control Sphere

— CWE-404: Improper Resource Shutdown or Release

Bibliography

[Dowd 2006] Chapter 10, "UNIX Processes" (File Descriptor Leaks 582-587)

[IEEE Std 1003.1: 2008]

[MSDN] Inheritance (Windows)

[NAI 1998]

5.31 Failing to detect and handle standard library errors (FIO04-C)

Failure to branch conditionally on detection or absence of a standard library error condition shall be diagnosed because this can result in undefined or unexpected behavior.

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, alternatively, 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	non-zero
`at_quick_exit`	zero	non-zero
`atexit`	zero	non-zero
`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	non-zero
`fclose`	zero	`EOF` (negative)
`fflush`	zero	`EOF` (negative)
`fgetc`	character read	`EOF`[b]
`fgetpos`	zero	non-zero
`fgets`	pointer to string	`NULL`
`fgetwc`	wide character read	`WEOF`[b]

Function	**Successful return**	**Error return**
fopen	pointer to stream	NULL
fopen_s	zero	non-zero
fprintf	number of characters (non-negative)	negative
fprintf_s	number of characters (non-negative)	negative
fputc	character written	EOF[a]
fputs	non-negative	EOF (negative)
fputws	non-negative	EOF (negative)
fread	elements read	elements read
freopen	pointer to stream	NULL
freopen_s	zero	non-zero
fscanf	number of conversions (non-negative)	EOF (negative)
fscanf_s	number of conversions (non-negative)	EOF (negative)
fseek	zero	non-zero
fsetpos	zero	non-zero
ftell	file position	-1L
fwprintf	number of wide characters (non-negative)	negative
fwprintf_s	number of wide characters (non-negative)	negative
fwrite	elements written	elements written
fwscanf	number of conversions (non-negative)	EOF (negative)
fwscanf_s	number of conversions (non-negative)	EOF (negative)
getc	character read	EOF[b]
getchar	character read	EOF[b]
getenv	pointer to string	NULL
getenv_s	pointer to string	NULL
gets_s	pointer to string	NULL
getwc	wide character read	WEOF[b]
getwchar	wide character read	WEOF[b]
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

Function	Successful return	Error return
`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	non-zero
`mbstowcs`	number of non-null elements	`(size_t)(-1)`
`mbstowcs_s`	zero	non-zero
`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 (non-negative)	negative
`putc`	character written	`EOF`[a]
`putwc`	wide character written	`WEOF`
`raise`	zero	non-zero
`realloc`	pointer to space	`NULL`
`remove`	zero	non-zero
`rename`	zero	non-zero
`setlocale`	pointer to string	`NULL`
`setvbuf`	zero	non-zero
`scanf`	number of conversions (non-negative)	`EOF` (negative)
`scanf_s`	number of conversions (non-negative)	`EOF` (negative)
`signal`	pointer to previous function	`SIG_ERR`, `errno > 0`
`snprintf`	number of characters that would be written (non-negative)	negative
`snprintf_s`	number of characters that would be written (non-negative)	negative

Function	**Successful return**	**Error return**
sprintf	number of non-null characters written	negative
sprintf_s	number of non-null characters written	negative
sscanf	number of conversions (non-negative)	EOF (negative)
sscanf_s	number of conversions (non-negative)	EOF (negative)
strchr	pointer to located character	NULL
strerror_s	zero	non-zero
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 (non-negative)	EOF (negative)
swscanf_s	number of conversions (non-negative)	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

Function	**Successful return**	**Error return**
`time`	calendar time	`(time_t)(-1)`
`timespec_get`	base	zero
`tmpfile`	pointer to stream	`NULL`
`tmpfile_s`	zero	non-zero
`tmpnam`	non-null pointer	`NULL`
`tmpnam_s`	zero	non-zero
`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 (non-negative)	negative
`vfprintf_s`	number of characters (non-negative)	negative
`vfscanf`	number of conversions (non-negative)	`EOF` (negative)
`vfscanf_s`	number of conversions (non-negative)	`EOF` (negative)
`vfwprintf`	number of wide characters (non-negative)	negative
`vfwprintf_s`	number of wide characters (non-negative)	negative
`vfwscanf`	number of conversions (non-negative)	`EOF` (negative)
`vfwscanf_s`	number of conversions (non-negative)	`EOF` (negative)
`vprintf_s`	number of characters (non-negative)	negative
`vscanf`	number of conversions (non-negative)	`EOF` (negative)
`vscanf_s`	number of conversions (non-negative)	`EOF` (negative)
`vsnprintf`	number of characters that would be written (non-negative)	negative
`vsnprintf_s`	number of characters that would be written (non-negative)	negative
`vsprintf`	number of non-null characters (non-negative)	negative
`vsprintf_s`	number of non-null characters (non-negative)	negative
`vsscanf`	number of conversions (non-negative)	`EOF` (negative)
`vsscanf_s`	number of conversions (non-negative)	`EOF` (negative)
`vswprintf`	number of non-null wide characters	negative
`vswprintf_s`	number of non-null wide characters	negative
`vswscanf`	number of conversions (non-negative)	`EOF` (negative)

Function	Successful return	Error return
vswscanf_s	number of conversions (non-negative)	EOF (negative)
vwprintf_s	number of wide characters (non-negative)	negative
vwscanf	number of conversions (non-negative)	EOF (negative)
vwscanf_s	number of conversions (non-negative)	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
wcsrtombs_s	zero	non-zero
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	non-zero
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

Function	Successful return	Error return
`wctype`	valid argument to `iswctype`	zero
`wmemchr`	pointer to located wide character	NULL
`wprintf_s`	number of wide characters (non-negative)	negative
`wscanf`	number of conversions (non-negative)	EOF (negative)
`wscanf_s`	number of conversions (non-negative)	EOF (negative)
[a] Use `ferror`.		
[b] Use `feof` and `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. C99 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.

EXAMPLE In this 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 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 (non-negative)	negative
`putchar`	character written	EOF [1]
`puts`	non-negative	EOF (negative)
`putwchar`	wide character written	WEOF
`vprintf`	number of characters (non-negative)	negative
`vwprintf`	number of wide characters (non-negative)	negative
`wprintf`	number of wide characters (non-negative)	negative

Exceptions

— EXP12-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);
  /* ... */
}
```

— EXP12-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.

Related guidelines

CERT C Secure Coding Standard: FIO04-C. Detect and handle input and output errors

MITRE CWE: CWE-391: Unchecked Error Condition

Bibliography

[Kettelwell 2002] Section 6, "I/O Error Checking"

[Seacord 2005] Chapter 7, "File I/O"

5.32 Failing to prevent or detect domain and range errors in math functions (FLP32-C)

Lack of prevention or detection of domain and range errors in math functions shall be diagnosed because the value returned is not the correct result of the computation. A *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. A *range error* occurs if the mathematical result of the function cannot be represented in an object of the specified type because of extreme magnitude.

The following table shows standard math functions and their domains and ranges. The standard math functions not in this table, such as `atan`, have no domain restrictions and do not produce range errors.

Table 4 — Standard math functions and their domains and ranges

Function	**Domain**	**Range error**
`acos(x), asin(x)`	`-1 <= x && x <= 1`	no
`atan2(y, x)`	`x != 0 || y != 0`	no
`acosh(x)`	`x >= 1`	no
`atanh(x)`	`-1 < x && x < 1`	no
`cosh(x) , sinh(x)`	none	yes
`exp(x) , exp2(x) , expm1(x)`	none	yes
`ldexp(x, exp)`	none	yes
`log(x) , log10(x) , log2(x)`	`x > 0`	no
`log1p(x)`	`x > -1`	no
`ilogb(x) , logb(x)`	`x != 0`	yes
`scalbn(x, n), scalbln(x, n)`	none	yes
`hypot(x, y)`	none	yes
`pow(x, y)`	`x > 0 || (x == 0 && y > 0) || (x < 0 &&` y is an integer)	yes
`sqrt(x)`	`x >= 0`	no
`erfc(x)`	none	yes
`lgamma(x) , tgamma(x)`	`x != 0 && !(x < 0 &&` x is an integer)	yes
`lrint(x) , lround(x)`	none	yes
`fmod(x, y)`	y != 0	no

Function	Domain	Range error
`nextafter(x, y)`, `nexttoward(x, y)`	none	yes
`fdim(x, y)`	none	yes
`fma(x, y, z)`	none	yes

EXAMPLE 1 In this example, a diagnostic is required because parameter `x` may not be in the domain of the C standard library function `sqrt`.

```
double f(double x) {
  double result = sqrt(x);  // diagnostic required
  return result * 2.0;
}
```

EXAMPLE 2 In this example, a diagnostic is required because the result of the call to the C standard library function `cosh` is not checked for a range error.

```
double g(double x) {
  double result = cosh(x);  // diagnostic required
  return result * 2.0;
}
```

EXAMPLE 3 In this example, a diagnostic is required because the parameters `x` and `y` may not be in the domain of the C standard library function `pow`, and the result of the call to `pow` is not checked for a range error.

```
double h(double x, double y) {
  double result = pow(x, y);  // diagnostic required
  return result * 2.0;
}
```

Related guidelines

CERT C Secure Coding Standard: FLP32-C. Prevent or detect domain and range errors in math functions

MITRE CWE: CWE-682: Incorrect Calculation

Bibliography

[Plum 1985] Rule 2-2

[Plum 1989] Topic 2.10, "conv - conversions and overflow"

5.33 Failing to sanitize the environment when invoking external programs (ENV03-C)

Invoking external programs that depend on the environment without first sanitizing the environment shall be diagnosed because the invoked program can be influenced by an attacker.

EXAMPLE In this example, a diagnostic is required on POSIX systems because the environment variable `IFS` is not sanitized before the call to `system`. If `IFS` is set to "." then the intended directory will not be found.

```
void f() {
  if (system("/bin/ls dir.`date +%Y%m%d`") == -1) {  // diagnostic required
    /* ... */
  }
}
```

Related guidelines

CERT C Secure Coding Standard: ENV03-C. Sanitize the environment when invoking external programs

ISO/IEC TR 24772 "XYS Executing or Loading Untrusted Code"

MITRE CWE:

— CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

— CWE-88: Argument Injection or Modification

— CWE-426: Untrusted Search Path

Bibliography

[CA-1995-14] "Telnetd Environment Vulnerability"

[Dowd 2006] Chapter 10, "UNIX II: Processes"

[IEEE Std 1003.1: 2008] The Open Group Base Specifications Issue 7

[Open Group 2004] Chapter 8, "Environment Variables," and confstr

[Viega 2003] Section 1.1, "Sanitizing the Environment"

[Wheeler 2003] Section 5.2, "Environment Variables"

5.34 Forming invalid pointers by library function

Invoking a C library function with a pair of 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. In addition, invocations of functions that do not directly lead to the formation of such pointers but that are likely to have unintended effects given the types of the arguments or the computations used to derive their values shall be diagnosed according to the remainder of this rule.

The motivation for this is that 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.

When the value of the integer argument passed to such a function would cause the function to form a pointer that does not point into or just past the end of the object pointed into by the first argument, the behavior is undefined (see item 103 in Annex B (informative) Undefined Behavior).

For a function `f` taking the pair of not necessarily consecutive arguments (p, n), where `p` is a non-`const` qualified (possibly `void *`) pointer and `n` is an integer that specifies the effective size of an object, a call to `f` where `n` is greater than the effective size of `*p` shall be diagnosed.

EXAMPLE 1 In the following function definition, the effective type of `*p` is `char` and the derived type of the expression `n` is a compatible character type. However, the effective size of `*p` is equal to `nchars`, which is less than `n` (that is, `nchars + 1`). Consequently, the call to `memset` is diagnosed.

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

For a function `f` taking the pair of not necessarily consecutive arguments (p, n), where p is a non-`const` qualified (possibly `void *`) pointer and n is an integer that specifies the effective size of an object, a call to `f` where where the effective type of `*p` is not compatible with the derived type of the expression n or `unsigned char` shall be diagnosed.

EXAMPLE 2 In the following function definition, assume `sizeof(int) == sizeof(float)` holds. The effective size of `*p` is equal to `4 * sizeof(float)` which is equal to `4 * sizeof(int)`. However, because the effective type of `*p` is `float` and the derived type of the expression `n` is `int`, the call to `memset` is diagnosed because `float` is incompatible with `int`.

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

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

  /* ... */
}
```

For a function g taking the triple of not necessarily consecutive arguments (p, q, n), where p is a non-`const` qualified (possibly `void`) pointer, q is a `const`-qualified (possibly `void`) pointer, and n is an integer that specifies the effective size of an object, a call to g where n is greater than the minimum of the effective size of `*p` and the effective size of `*q` shall be diagnosed.

For a function g taking the triple of not necessarily consecutive arguments (p, q, n), where p is a non-`const` qualified (possibly `void`) pointer, q is a `const`-qualified (possibly `void`) pointer, and n an integer that specifies the effective size of an object, a call to g where the effective type of `*p` is incompatible with either the effective type of `*q` or `unsigned char` shall be diagnosed.

EXAMPLE 3 In the following function definition, assume `(sizeof(int) < sizeof(double))` holds. The effective size of `*p` is equal to `sizeof(int)`, the effective size of `*p` is equal to `sizeof(double)`, and `n` is equal to `sizeof(int)`. Consequently, `n` is less than or equal to the minimum of the effective size of `*p` and the effective size of `*q`. Furthermore, the effective type of `*p` (that is, `int`) is compatible with the derived type of the expression `n` (also `int`). However, the effective type of `*p` (`int`) is not compatible with the effective type of `*q` (`double`), so the call to `memcpy` is diagnosed.

```
void f4(int *a) {
  double b = 3.14;
  const size_t n = sizeof(*a);
  void *p = a;
  void *q = &b;

  memcpy(p, q, n);  // diagnostic required

  /* ... */
}
```

For a function g taking the triple of not necessarily consecutive arguments (p, q, n), where p is a non-`const` qualified (possibly `void`) pointer, q is a `const`-qualified (possibly `void`) pointer, and n is an integer that specifies the effective size of an object, a call to g where the effective type of `*p` is not compatible with the derived type of the expression n shall be diagnosed.

EXAMPLE 4 In the following function definition, assume that the effective size of `*p` and the effective size of `*q` are not determinable. Furthermore, the effective type of `*p` (that is, `char`) is compatible with the effective type of `*q` (also `char`). However, the effective type of `*p` (`char`) is not compatible with the derived type of the expression `n` (pointer to `char`), so the call to `memcpy` is diagnosed.

```
void f5(char p[], const char *q) {
  const size_t n = sizeof(p);  // diagnostic required
  memcpy(p, q, n);

  /* ... */
}
```

For an assignment expression `E` whose right operand is a call to a memory allocation function taking an integer argument `n`, and whose left operand is of type `T*` or the equivalent initialization expression, the expression E where `(n < sizeof (T))` shall be diagnosed.

For an assignment expression `E` whose right operand is a call to a memory allocation function taking an integer argument `n`, and whose left operand is of tyep `T*` or the equivalent initialization expression, the expression `E` where `T` is compatible with neither the derived type of the expression `n` nor `unsigned char` shall be diagnosed.

EXAMPLE 5 In the following function definition, assume `(sizeof(wchar_t) == sizeof(wchar_t *))` holds (that is, the size of the `wchar_t` type is the same as that of an object pointer). The initialization expression of `q` with type `T *`, where `T` is `wchar_t`, is a memory allocation function called with the size argument `n` whose value is `(sizeof(wchar_t *) * 14)`, which is greater than `sizeof(T)` (that is, `sizeof(wchar_t)`). However, because `n` is derived from an expression involving `sizeof(wchar_t *)`, the derived type of the expression `n` is `wchar_t *`, which is compatible with neither `wchar_t` nor `unsigned char`. Consequently, the expression is diagnosed.

```
wchar_t *f7() {
  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;
}
```

5.35 Forming or using out-of-bounds pointers or array subscripts (ARR30-C)

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

Likewise, using an array subscript so that the resulting reference does not refer to an element in the array also results in potentially exploitable undefined behavior and shall be diagnosed.

C99 identifies four distinct situations in which undefined behavior (UB) may arise as a result of invalid pointer operations:

UB	**Description**
43	*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.*
44	*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.*
46	*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]`).*

UB	Description
59	*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.*
103	*The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid.*

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

```
enum { 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.

```
enum { 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.

```
enum { TABLESIZE = 100 };

static int table[TABLESIZE];

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

  return NULL;
}
```

EXAMPLE 4 In this 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.

```
enum { 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`.

```
enum { 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;
  }

  while (path != machine_name_end) {
    *machine_name++ = *path++;
  }

  *machine_name = '\0';

  return machine_name;
}
```

EXAMPLE 6 In this 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 example, a diagnostic is required because a value is stored beyond the end of the arrays `matrix[0..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 `matrix[0..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 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() {
  struct S *s = (struct S *)malloc(sizeof(struct S));
  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() {
  struct S *s = (struct S *)malloc(sizeof(struct S));
  s->len = 0;
  /* ... */
  char *where = find(s, '.');
  if (where == NULL) {
    return;
  }

  /* ... */
}
```

EXAMPLE 12 In this example, a diagnostic is required because the parameters passed to the standard library function `fread` are calculated incorrectly, which may result in values being assigned to beyond the end of the array `wbuf`.

```
void h(FILE *file) {
  wchar_t wbuf[BUFSIZ];

  const size_t size = sizeof(wbuf[0]);
  const size_t nitems = sizeof(wbuf);
  size_t nread = fread(wbuf, size, nitems, file);  // diagnostic required
  if (nread != nitems) {
    return;
  }

  /* ... */
}
```

EXAMPLE 13 In this compliant example, a diagnostic is not required because the parameters passed to the standard library function `fread` are calculated correctly, which will not result in values being assigned to beyond the end of the array `wbuf`.

```
void h(FILE *file) {
  wchar_t wbuf[BUFSIZ];

  const size_t size = sizeof(wbuf[0]);
  const size_t nitems = sizeof(wbuf) / size;

  size_t nread = fread(wbuf, size, nitems, file);
  if (nread != nitems) {
    return;
  }

  /* ... */
}
```

EXAMPLE 14 In this example, a diagnostic is required because the string `dest` may not be not be null-terminated in the call to `printf`.

```
void f(char *src, size_t src_size) {
  char dest[BUFSIZ];

  dest[sizeof(dest) - 1] = '\0';
  src[src_size - 1] = '\0';

  strncpy(dest, src, sizeof(dest));
  printf("%s\n", dest);  // diagnostic required for 'dest' as an argument to
printf
}
```

EXAMPLE 15 In this example, a diagnostic is required because the string `cur_msg` is not null-terminated when passed to `strlen`.

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

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

  /* ... */

  if (cur_msg != NULL) {
    temp_size = cur_msg_size / 2 + 1;
    temp = realloc(cur_msg, temp_size);
    if (temp == NULL) {
      /* ... */
    }

    cur_msg = temp;
    cur_msg_size = temp_size;
    cur_msg_len = strlen(cur_msg);  // diagnostic required for 'cur_msg' as an
argument to strlen
  }
}
```

5.35.1 Related Vulnerabilities

CVE-2008-1517 results from a violation of this rule. Before Mac OSX version 10.5.7, the xnu kernel accessed an array at an unverified, user-input index, allowing an attacker to execute arbitrary code by passing an index greater than the length of the array, thereby accessing outside memory [xorl 2009].

Related guidelines

CERT C Secure Coding Standard: ARR30-C. Do not form or use out of bounds pointers or array subscripts

ISO/IEC TR 24772 "XYX Boundary Beginning Violation," "XYY Wrap-around Error," and "XYZ Unchecked Array Indexing"

MITRE CWE:

— CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

— CWE-121: Stack-based Buffer Overflow

— CWE-122: Heap-based Buffer Overflow

— CWE-129: Improper Validation of Array Index

— CWE-788: Access of Memory Location After End of Buffer

— CWE-805: Buffer Access with Incorrect Length Value

Bibliography

[CERT/CC 2003]

[Microsoft 2003]

[Pethia 2003]

[Seacord 2005] Chapter 1, "Running with Scissors"

[Viega 2005] Section 5.2.13, "Unchecked array indexing"

[xorl 2009] "CVE-2008-1517: Apple Mac OS X (XNU) Missing Array Index Validation"

5.36 Freeing memory multiple times (MEM31-C)

Freeing memory multiple times shall be diagnosed (subject to exceptions below) because this results in "double-free" vulnerabilities [Seacord 2005].

EXAMPLE 1 In this 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 example, a diagnostic is required because `realloc` may free `str1` when it returns `NULL`, resulting in `str1` being freed twice.

```
void g(char *str1, size_t size) {
  char *str2 = (char *)realloc(str1, size);
  if (str2 == NULL) {
    free(str1);  // diagnostic required
    return;
  }
}
```

According to C99 (7.20.3),

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 7.20.3.4,

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 non-conforming 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: MEM31-EX1

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.

Related guidelines

CERT C Secure Coding Standard: MEM31-C. Free dynamically allocated memory exactly once

ISO/IEC TR 24772 "XYK Dangling Reference to Heap" and "XYL Memory Leak"

MITRE CWE: CWE-415: Double Free

Bibliography

[MIT 2005]

[OWASP] Double Free

[Seacord 2005]

[Viega 2005] "Doubly freeing memory"

[VU#623332]

5.37 Including tainted or out-of-domain input in a format string (FIO30-C)

Invoking any of the formatted input/output functions identified in C99 Section 7.20.6 ("Formatted input/output functions"[ISO/IEC 9899:1999]), where the format argument references string data that is tainted or out-of-domain with respect to character content, shall be diagnosed because this can result in undefined or unexpected behavior. Any comparison of a character in the string to a value other than the null character sanitizes the string. Additionally, an empty string is not considered to be tainted.

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 [Seacord 2005].

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 1 In this example, a diagnostic is required because a format string is read from an external catalog 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 an external catalog 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 example, a diagnostic is required because the tainted string `user` may contain untrusted data.

```
void incorrect_password() {
  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) {
    /* ... */
  }

  ret = snprintf(msg, size, MSG_FORMAT, user);
  free(user);
  if (ret < 0) {
    /* ... */
  } else if (ret >= size) {
    /* ... */
  }

  fprintf(stderr, msg); // diagnostic required
  free(msg);
}
```

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, which is not controlled by the user.

```
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;
}
```

Related guidelines

CERT C Secure Coding Standard: FIO30-C. Exclude user input from format strings

ISO/IEC TR 24772 "RST Injection"

MITRE CWE: CWE-134: Uncontrolled Format String

Bibliography

[Seacord 2005] Chapter 6, "Formatted Output"

[Viega 2005] Section 5.2.23, "Format string problem"

5.38 Incorrectly setting and using errno (ERR30-C)

Incorrectly setting and using `errno` shall be diagnosed because doing so can result in undefined or unexpected behavior. The correct way to set and check `errno` is defined in the following cases.

5.38.1 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 following functions set `errno` and return an in-band error indicator.

Table 5 — 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 C99 standard, if the result of `strtod`, `strtof`, or `strtold` (and the related wide-character

Function name	Return value	errno value
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."		

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

A program that uses errno for error checking need not set errno to zero before calling one of these library functions. Then, if and only if the function returned an error indicator, the program shall inspect errno before a subsequent library function call.

The following functions set errno and return an out-of-band error indicator.

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

Function name	Return value	errno value
fgetpos	non-zero	positive
fgetwc	WEOF	EILSEQ
fputwc	WEOF	EILSEQ
fsetpos	non-zero	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.

5.38.3 Library functions that may or may not set errno

Programs shall not rely on errno after calling a function that could or could not 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 non-zero, then errno is set to EDOM.

— According to the C99 standard, "If a floating result overflows and default rounding is in effect, or if the mathematical result is an exact infinity (for example log(0.0)), 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" [ISO/IEC 9899:1999].

— Similarly, according to the C99 standard, "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" [ISO/IEC 9899:1999].

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

5.38.4 Library functions that do not explicitly set `errno`

Programs shall not rely on `errno` to determine whether an error occurred after calling a 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`.

EXAMPLE 1 In this 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 *string) {
  char *endptr = NULL;
  unsigned long number = strtoul(string, &endptr, 0);

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

  /* ... */
}
```

EXAMPLE 2 In this 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() {
  signal(SIGINT, SIG_DFL);
  if (errno != 0) {  // diagnostic required
    /* ... */
  }
}
```

EXAMPLE 3 In this 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`.

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

Related guidelines

CERT C Secure Coding Standard: ERR30-C. Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure

MITRE CWE: CWE-456: Missing Initialization

Bibliography

[Brainbell.com] Macros and Miscellaneous Pitfalls

[Horton 1990] Section 11 p. 168 and Section 14 p. 254

[Koenig 1989] Section 5.4 p. 73

5.39 Interleaving stream inputs and outputs without a flush or positioning call (FIO39-C)

The following scenarios shall be diagnosed because either can result in undefined behavior:

— 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

According to section 7.19.5.3 of C99,

When a file is opened with update mode both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the `fflush` *function or to a file positioning function (*`fseek`*,* `fsetpos`*, or* `rewind`*), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.*

(See also undefined behavior 143, Annex B.)

EXAMPLE In this 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, BUFSIZ, 1, file) != BUFSIZ) {
    /* ... */
  }

  if (fread(data, BUFSIZ, 1, file) != 0) {  // diagnostic required
    /* ... */
  }

  fclose(file);
}
```

Related guidelines

CERT C Secure Coding Standard: FIO39-C. Do not alternately input and output from a stream without an intervening flush or positioning call

5.40 Invoking an unsafe macro with arguments containing side effects (PRE31-C)

An unsafe function-like macro is unsafe with respect to a parameter if the macro evaluates that parameter more than once in the code expansion or never evaluates the parameter at all. Call such a parameter an *unsafe parameter*. Invoking an unsafe function-like macro with an argument for an unsafe parameter containing assignment, increment, decrement, volatile access, or other side effects, or a function call that performs side effects, shall be diagnosed because this may result in undefined or unexpected behavior.

EXAMPLE 1 In this example, a diagnostic is required because an expression containing increment `++n` is passed to the unsafe function-like macro `CUBE`.

```
#define CUBE(x) ((x) * (x) * (x))

int f(int n) {
  int m = CUBE(++n);  // diagnostic required

  /* ... */
  return m;
}
```

EXAMPLE 2 In this example, a diagnostic is required because an expression containing decrement `--n` is passed to the unsafe function-like macro ABS.

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

int g(int n) {
  int m = ABS(--n);  // diagnostic required

  /* ... */
  return m;
}
```

EXAMPLE 3 In this example, a diagnostic is required when `getc` is implemented as an unsafe function-like macro because an expression containing assignment, `fptr = fopen(filename, "r")`, is passed to `getc`.

```
char getch(const char *filename) {
  FILE *fptr = NULL;

  int c = getc(fptr = fopen(filename, "r"));  // diagnostic required
  if (c == EOF) {
    /* ... */
  }

  return c;
}
```

EXAMPLE 4 In this example, a diagnostic is required when `putc` is implemented as an unsafe function-like macro because an expression containing side effects `fptr ? fptr : (fptr = fopen(filename, "w"))` is passed to `put`.

```
void putalpha(const char *filename) {
  FILE *fptr = NULL;

  int c = 'a';
  while (c <= 'z') {
    if (putc(c++, fptr ? fptr : (fptr = fopen(filename, "w"))) == EOF) {  //
diagnostic required
      /* ... */
    }
  }
}
```

Related guidelines

CERT C Secure Coding Standard: PRE31-C. Avoid side-effects in arguments to unsafe macros

ISO/IEC TR 24772 "NMP Pre-processor Directions"

MISRA-C 2004, Rule 19.6

Bibliography

[Plum 1985] Rule 1-11

5.41 Modifying constant values (EXP40-C)

C99 Section 6.7.3, "Type qualifiers," Paragraph 5 [ISO/IEC 9899:1999], states,

If an attempt is made to modify an object defined with a `const`-qualified type through use of an lvalue with non-`const`-qualified type, the behavior is undefined.

(See also undefined behavior 61 in Annex B.)

Attempting to modify a `const`-qualified object through the use of an lvalue of a non-`const`-qualified type shall be diagnosed.

There are existing compiler implementations that allow `const`-qualified values to be modified without generating a warning message.

EXAMPLE 1 In this example, a diagnostic is required because the constant value in the variable `c` is modified.

```
void f() {
  char const **cpp;
  char *cp;
  char const c = 'A';

  cpp = &cp;  // diagnostic required
  *cpp = &c;
  *cp = 'B';
}
```

EXAMPLE 2 In this example, a diagnostic is required because a constant value in the array `s` is modified.

```
const char s[] = "bar";

int main(void) {
  *(char *)s = '\0';  // diagnostic required
  /* ... */
  return 0;
}
```

5.42 Modifying string literals (STR30-C)

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

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

EXAMPLE 2 In this example, a diagnostic is required because the string literal `"/tmp/edXXXXXX"` is modified by the POSIX function `mkstemp`.

```
void f2() {
  mkstemp("/tmp/edXXXXXX");  // diagnostic required
  /* ... */
}
```

EXAMPLE 3 In this 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() {
  char *last_slash = strrchr("/tmp/filename", '/');
  *last_slash = '\0';  // diagnostic required
  /* ... */
}
```

EXAMPLE 4 In this 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() {
  *strrchr("/tmp/filename", '/') = '\0';  // diagnostic required
  /* ... */
}
```

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

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

Exception: STR030-EX1

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.

```
int main(void) {
  char *p = "abc";
  printf("%s\n", p);
  return 0;
}
```

Related guidelines

CERT C Secure Coding Standard: STR30-C. Do not attempt to modify string literals

Bibliography

[Plum 1991] Topic 1.26, "strings - string literals"

[Summit 1995] comp.lang.c FAQ list - Question 1.32

5.43 Modifying the string returned by getenv, localeconv, setlocale, and strerror (ENV30-C)

Modifying the objects or strings returned by the library functions listed in the table below shall be diagnosed because doing so results in undefined behavior.

C99 identifies the following three instances of undefined behavior (UB), which arise as a result of modifying the data structures or strings returned from `getenv`, `localeconv`, `setlocale`, and `strerror`:

UB	Description
114	*The program modifies the string pointed to by the value returned by the setlocale function (7.11.1.1).*
115	*The program modifies the structure pointed to by the value returned by the localeconv function (7.11.2.1).*
174	*The string set up by the getenv or strerror function is modified by the program (7.20.4.5, 7.21.6.2).*

EXAMPLE 1 In this example, a diagnostic is required because the string returned from the C standard library function `setlocale` is modified.

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

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

  /* ... */
}
```

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

```
void f2() {
  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 example, a diagnostic is required because the string returned from the C standard library function `getenv` is modified.

```
void f3() {
  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 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[16];
  sprintf(buf, "(errno = %d)", error);

  char *error_str = strerror(error);

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

Related guidelines

CERT C Secure Coding Standard: ENV30-C. Do not modify the object referenced by the return value of certain functions

Bibliography

[Open Group 2004] `getenv`

5.44 Not finishing case labels with a break statement (MSC17-C)

A set of statements associated with a `case` label that does not end with a `break` statement shall be diagnosed (subject to exceptions below) because this can result in unexpected behavior.

EXAMPLE In this example, a diagnostic is required because the case where `widget_type` has value `WE_W` lacks a concluding `break` statement.

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

void f(enum WidgetEnum widget_type) {
  switch (widget_type) {
    case WE_W:  // diagnostic required
      /* ... */
    case WE_X:
      /* ... */
      break;
    case WE_Y:
    case WE_Z:
      /* ... */
      break;
    default:
      /* ... */
      break;
  }
}
```

Exception: MSC17-EX1

The last label in a `switch` statement, if it does not conclude with a `break` statement, need not be diagnosed.

Related guidelines

CERT C Secure Coding Standard: MSC17-C. Finish every set of statements associated with a case label with a break statement

5.45 Overflowing signed integers (INT32-C)

An exceptional condition that occurs during the evaluation of an expression is undefined behavior (see undefined behavior 33 in Annex B). The common cause of such an exception is signed integer overflow. Because signed integer overflow is undefined behavior, implementations are allowed to silently wrap (the most common behavior) or trap. Because signed integer overflow produces a silent wraparound in most existing C implementations, some programmers assume that this is a well-defined behavior.

Whenever at least one operand is a tainted, potentially mutilated, or out-of-domain value, signed integer operations that can overflow shall be diagnosed.

The following table indicates whether an expression may result in overflow (denoted by the ✓ icon) or not (denoted by the ✗ icon).

Table 7 — Expressions and overflow

Operator	Overflow	Operator	Overflow	Operator	Overflow	Operator	Overflow
`+`	✓	`-=`	✓	`<<`	✓	`<`	✗
`-`	✓	`*=`	✓	`>>`	✗	`>`	✗
`*`	✓	`/=`	✓	`&`	✗	`>=`	✗
`/`	✓	`%=`	✓	`|`	✗	`<=`	✗
`%`	✓	`<<=`	✓	`^`	✗	`==`	✗
`++`	✓	`>>=`	✗	`~`	✗	`!=`	✗
`--`	✓	`&=`	✗	`!`	✗	`&&`	✗
`=`	✗	`|=`	✗	unary +	✗	`||`	✗
`+=`	✓	`^=`	✗	unary -	✓	`?:`	✗

EXAMPLE 1 In this 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_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_INT(x);

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

Related guidelines

CERT C Secure Coding Standard: INT32-C. Ensure that operations on signed integers do not result in overflow

ISO/IEC TR 24772 "XYY Wrap-around Error"

MITRE CWE: CWE-190: Integer Overflow or Wraparound

Bibliography

[Dowd 2006] Chapter 6, "C Language Issues" (Arithmetic Boundary Conditions, pp. 211-223)

[Seacord 2005] Chapter 5, "Integers"

[Viega 2005] Section 5.2.7, "Integer overflow"

[VU#551436]

[Warren 2002] Chapter 2, "Basics"

5.46 Passing arguments to character handling functions that are not representable as unsigned char (STR37-C)

Arguments to the character handling functions in `<ctype.h>` that are not representable as `unsigned char` shall be diagnosed because these functions are defined only for values representable as `unsigned char` and the macro `EOF`.

The following character classification functions are affected:

```
isalnum  isalpha  isascii  isblank

iscntrl  isdigit  isgraph  islower

isprint  ispunct  isspace  isupper

isxdigit toascii  toupper  tolower
```

EXAMPLE In this example, a diagnostic is required because the parameter to `isspace`, `*t`, may 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;
}
```

Related guidelines

CERT C Secure Coding Standard: STR37-C. Arguments to character handling functions must be representable as an unsigned char

MITRE CWE:

— CWE-686: Function Call With Incorrect Argument Type

— CWE-704: Incorrect Type Conversion or Cast

Bibliography

[Kettelwell 2002] Section 1.1, "<ctype.h> And Characters Types"

5.47 Passing pointers into the same object as arguments to different restrict-qualified parameters (DCL33-C)

Function arguments that are `restrict`-qualified pointers and reference overlapping objects shall be diagnosed because accessing the object pointed to by a `restrict`-qualified pointer via another pointer results in undefined behavior.

This corresponds to undefined behavior 65 as defined in Annex B:

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.

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

```
void abcabc() {
  char str[]= "abc123";
  char *ptr1 = str;
  char *ptr2 = str + strlen("abc");

  memcpy(ptr2, ptr1, strlen("123"));  // diagnostic required
  puts(str);
}
```

EXAMPLE 2 In this example, a diagnostic is required because the pointer `src` is twice a `restrict`-qualifed 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() {
  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);
}
```

Related guidelines

CERT C Secure Coding Standard: DCL33-C. Ensure that restrict-qualified source and destination pointers in function arguments do not reference overlapping objects

ISO/IEC TR 24772 "CSJ Passing parameters and return values"

5.48 Performing bitwise operations on Boolean operands (EXP16-C)

Applying `bitwise` AND (&, ampersand), `bitwise` OR (|, pipe), XOR (^, caret), or COMPLIMENT (~, tilde) on a Boolean operand, where an operand is considered to be Boolean if its value is the result of a comparison, a logical AND (&&, double-ampersand), a logical OR (||, double-pipe), or a NEGATION (!, exclamation point) operator, shall be diagnosed unless both operands of a `bitwise` binary operator are Boolean. Note that the error in these cases is often the choice of operator in the subexpression rather than the outermost operator.

EXAMPLE 1 In this example, a diagnostic is required because the bitwise AND (&) operator is applied to the Boolean expression `geteuid() == 0`.

```
void f() {
  if (!(getuid() & geteuid() == 0)) {  // diagnostic required
    /* ... */
  }
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because a bitwise operator is not applied to the Boolean expression `geteuid() == 0`.

```
void f() {
  if (!(getuid() && geteuid() == 0)) {
    /* ... */
  }
}
```

Related guidelines

CERT C Secure Coding Standard: EXP17-C. Do not perform bitwise operations in conditional expressions

ISO/IEC TR 24772 "KOA Likely Incorrect Expressions"

Bibliography

[Hatton 1995] Section 2.7.2, "Errors of omission and addition"

5.49 Reallocating or freeing memory that was not dynamically allocated (MEM34-C)

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 because this results in undefined behavior.

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

```
enum { BUFSIZE = 256 };

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

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

```
enum { MAX_ALLOCATION = 1000 };

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

  if (argc == 2) {
    len = strlen(argv[1]) + 1;
    if (len > MAX_ALLOCATION) {
      /* Handle error */
    }
    str = (char *)malloc(len);
    if (str == NULL) {
      /* Handle allocation error */
    }
    strcpy(str, argv[1]);
  }
  else {
    str = "usage: $>a.exe[string]";
    printf("%s\n", str);
  }
  /* ... */
  free(str);  // diagnostic required
  return 0;
}
```

Exception: MEM34-EX1

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.

Related guidelines

CERT C Secure Coding Standard: MEM34-C. Only free memory allocated dynamically

ISO/IEC TR 24772 "AMV Type-breaking reinterpretation of data"

MITRE CWE:

— CWE-590: Free of Memory not on the Heap

— CWE-628: Function Call with Incorrectly Specified Arguments

Bibliography

[ISO/IEC 9899:1999] Section 7.20.3.4, "The realloc function," and Section 7.20.3.3, "The free function"

5.50 Referencing uninitialized memory (EXP33-C)

There are two main sources of uninitialized memory:

— uninitialized automatic variables and

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

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` shall be diagnosed because doing so has undefined behavior. Typical consequences of accessing uninitialized memory relevant to security range from denial of service lead to information exposure as a result of leaking sensitive data previously stored in a memory region.

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 1 In this 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;
  }
}

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

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

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

```
int do_auth() {
  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 0;
}
```

EXAMPLE 3 In this 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));
  for (size_t i = 0; i != n; ++i) {
    a[i] = a[i] ^ a[i];  // diagnostic required
  }

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

EXAMPLE 4 In this 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));
  for (size_t i = 0; i != n * 2 + 1; ++i) {
    if (a[i] < 0) {
      a[i] = abs(a[i]);  // diagnostic required
    }
  }

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

Related guidelines

CERT C Secure Coding Standard:

— EXP33-C. Do not reference uninitialized memory

— MEM09-C. Do not assume memory allocation routines initialize memory

ISO/IEC TR 24772 "LAV Initialization of Variables"

Bibliography

[Flake 2006]

[mercy 2006]

5.51 Shifting signed types (INT13-C)

Right-shifting a value of a signed type shall be diagnosed because the result of a right shift expression whose operand has a negative value may lead to unexpected results.

EXAMPLE In this example, a diagnostic is required because the value of `i` may be negative in the right shift subexpressions in the `return` statement.

```
int32_t bswap32(int32_t i) {
  return ((i >> 24)  // diagnostic required
    | ((i >> 8) & 0xff00)  // diagnostic required
    | ((i << 8) & 0xff0000)
    | (i << 24));
}
```

Related guidelines

CERT C Secure Coding Standard: INT13-C. Use bitwise operators only on unsigned operands

ISO/IEC 2003 Section 6.5.7, "Bitwise shift operators"

ISO/IEC TR 24772 "STR Bit Representations," "XYY Wrap-around Error," and "XZI Sign Extension Error"

MITRE CWE: CWE-682: Incorrect Calculation

Bibliography

[Dowd 2006] Chapter 6, "C Language Issues"

5.52 Subtracting or comparing two pointers that do not refer to the same array (ARR36-C)

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 (subject to exceptions below) because this results in undefined behavior. The relational operators are >, <, >=, and <=.

C99 identifies two distinct situations in which undefined behavior (UB) may arise as a result of using pointers that do not point to the same object:

UB	**Description**
45	*Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).*
50	*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 In this example, a diagnostic is required because the pointers `string` and `(char **)next_num_ptr` are subtracted and do not refer to the same array.

```
enum { SIZE = 256 };

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

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

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

Exceptions

— ARR36-EX1: Comparing two pointers within the same object does not need to be diagnosed.

— ARR36-EX2: Subtracting two pointers to `char` within the same object does not need to be diagnosed.

Related guidelines

CERT C Secure Coding Standard: ARR36-C. Do not subtract or compare two pointers that do not refer to the same array

MITRE CWE: CWE-469: Use of Pointer Subtraction to Determine Size

Bibliography

[Banahan 2003] Section 5.3, "Pointers," and Section 5.7, "Expressions involving pointers"

5.53 Taking the size of a pointer to determine the size of the pointed-to type (EXP01-C)

Using the `sizeof` operator on a pointer type shall be diagnosed because this is often a sign of programmer error and can result in undefined or unexpected behavior.

EXAMPLE In this example, a diagnostic is required because the `sizeof` operator is applied to the pointer variable `d_array`.

```
double *allocate_array(size_t num_elems) {
  double *d_array;

  if (num_elems > SIZE_MAX / sizeof(d_array)) {  // diagnostic required
    /* ... */
  }

  d_array = (double *)malloc(sizeof(d_array) * num_elems);  // diagnostic 
required
  if (d_array == NULL) {
    /* ... */
  }

  return d_array;
}
```

Related guidelines

CERT C Secure Coding Standard: EXP01-C. Do not take the size of a pointer to determine the size of the pointed-to type

MITRE CWE: CWE-467: Use of sizeof() on a Pointer Type

Bibliography

[Drepper 200] Section 2.1.1, "Respecting Memory Bounds"

[Viega 2005] Section 5.6.8, "Use of `sizeof` on a pointer type"

5.54 Using a value for fsetpos that is returned from fgetpos (FIO44-C)

Using an offset value for `fsetpos`, other than the value returned from `fgetpos`, shall be diagnosed because this results in undefined behavior.

EXAMPLE In this example, a diagnostic is required because an offset value other than the 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;
}
```

Related guidelines

CERT C Secure Coding Standard: FIO44-C. Only use values for fsetpos() that are returned from fgetpos()

5.55 Using abort or assert when atexit handlers are registered (ERR06-C)

Using `assert` or `abort` in a program where `atexit` handlers are registered shall be diagnosed because these functions terminate the program and do not execute `atexit` handlers.

EXAMPLE In this example, a diagnostic is required because the C standard library function-like macro `assert` is called while the `atexit` handler `cleanup` is registered.

```
void cleanup() {
  /* ... */
}

int main(int argc, char *argv[]) {
  if (atexit(cleanup) != 0) {
    /* ... */
  }

  assert(argc > 1);  // diagnostic required

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

Related guidelines

CERT C Secure Coding Standard: ERR06-C. Understand the termination behavior of assert() and abort()

ISO/IEC TR 24772 "REU Termination Strategy"

5.56 Using an object overwritten by getenv, localeconv, setlocale, and strerror

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 because the object may be overwritten by the subsequent call to the function.

EXAMPLE 1 In this 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() {
  char *tmpvar = getenv("TMP");
  char *tempvar = getenv("TEMP");

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

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

EXAMPLE 2 In this 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 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;

  fprintf(stderr, "parsing results: %s, %s",
          strerror(e1), strerror(e2));  // diagnostic required

}
```

Related guidelines

CERT C Secure Coding Standard: ENV00-C. Do not store the pointer to the string returned by getenv()

ISO/IEC TR 24731-2

Bibliography

[MSDN] `_dupenv_s and _wdupenv_s, getenv_s, _wgetenv_s`

[Open Group 2004] Chapter 8, and "Environment Variables", `strdup`

[Viega 2003] Section 3.6, "Using Environment Variables Securely"

5.57 Using character values that are indistinguishable from EOF (FIO34-C)

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 because a character type cannot represent all character values plus the value of `EOF`.

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 because a wide-character type cannot represent all character values plus the value of `WEOF`.

EXAMPLE 1 In this 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() {
  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 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() {
  char 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';
  printf("%s\n", buf);
}
```

Related guidelines

CERT C Secure Coding Standard: FIO34-C. Use int to capture the return value of character IO functions

ISO/IEC TR 24731-1:2007 Section 6.5.4.1, "The `gets_s` function"

Bibliography

[NIST 2006] SAMATE Reference Dataset Test Case ID 000-000-088

5.58 Using identifiers that are reserved for the implementation (DCL37-C)

According to ISO/IEC 9899:TC3 Section 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 future library directions) 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 one of its associated headers is included.

No other identifiers are reserved.[1] 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. See also undefined behavior 100 of Annex J of C99. 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.

NOTE The POSIX ® standard extends the set of identifiers reserved by C99 to include an open-ended set of its own. See section 2.2 Compilation Environment in \[IEEE Std 1003.1: 2008\].

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

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

EXAMPLE 2 In this 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 example, a diagnostic is required because the file scope identifiers `_max_limit` and `_limit` are reserved because they begin with an underscore.

```
#include <stddef.h>   /* for size_t */

static const size_t _max_limit = 1024;  // diagnostic required
size_t _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.

```
#include <stddef.h>   /* for size_t */

static const size_t max_limit = 1024;
size_t limit = 100;

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

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

```
#include <inttypes.h>   /* for int_fast16_t and PRIdFAST16 */

static const int_fast16_t INTFAST16_LIMIT_MAX = 12000;  // diagnostic required

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

    if (INTFAST16_LIMIT_MAX < val) {
      sprintf(buf, "The value is too large");
    } else {
      snprintf(buf, MAX_SIZE, "The value is %" PRIdFAST16, val);
    }

    /* ... */
}
```

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

```
#include <inttypes.h>   /* for int_fast16_t and PRIdFAST16 */

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 {
      snprintf(buf, BUFSIZE, "The value is %" PRIdFAST16, val);
    }

    /* ... */
}
```

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

```
#include <stddef.h>

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.

```
#include <stddef.h>

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

void my_free(void *ptr) {
  /* ... */
  /* return storage to own pool */
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.1.3, "Reserved Identifiers"

[IEEE Std 1003.1: 2008] Section 2.2 "The Compilation Environment"

5.59 Using integer arithmetic to calculate a value for assignment to a floating-point variable (FLP33-C)

Using integer arithmetic to calculate a value for assignment to a floating-point variable shall be diagnosed because this may lead to a loss of information.

EXAMPLE In this example, a diagnostic is required because the values assigned to the floating-point variables `d`, `e`, and `f` are calculated using integer arithmetic.

```
void f() {
  short a = 533;
  int b = 6789;
  long c = 466438237;

  float d = a / 7;  // diagnostic required
  double e = b / 30;  // diagnostic required
  double f = c * 789;  // diagnostic required

  printf("%f %f %f\n", d, e, f);
}
```

Related guidelines

CERT C Secure Coding Standard: FLP33-C. Convert integers to floating point for floating point operations

MITRE CWE:

— CWE-681: Incorrect Conversion between Numeric Types

— CWE-682: Incorrect Calculation

Bibliography

[Hatton 1995] Section 2.7.3, "Floating-point misbehavior"

5.60 Using invalid format strings (FIO00-C)

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 because it results in undefined behavior.

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

```
void f() {
  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
}
```

Related guidelines

CERT C Secure Coding Standard: FIO00-C. Take care when creating format strings

MITRE CWE: CWE-686: Function Call With Incorrect Argument Type

5.61 Using non-unique identifiers (DCL32-C)

Using non-unique external identifiers at file scope or internal identifiers in the same scope shall be diagnosed because this can result in unexpected behavior.

Section 5.2.4.1 of the C Standard defines the following minimum requirements for uniqueness:

— 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character) and

— 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any).

Rule Declaring the same function or object in incompatible ways (ARR31) prohibits excessively long identifiers from resulting in incompatible declarations.

EXAMPLE 1 In this example, a diagnostic is required because the external identifiers `global_symbol_definition_lookup_table_a` and `global_symbol_definition_lookup_table_b` are not unique because the first 31 characters are identical.

```
extern int *global_symbol_definition_lookup_table_a;
extern int *global_symbol_definition_lookup_table_b;  // diagnostic required
```

EXAMPLE 2 In this example, a diagnostic is required because the external identifiers `\U00010401\U00010401\U00010401\U00010401` and `\U00010401\U00010401\U00010401\U00010402` are not unique because the first three universal character names are identical.

```
extern int *\U00010401\U00010401\U00010401\U00010401;
extern int *\U00010401\U00010401\U00010401\U00010402;  // diagnostic required
```

Exception: DCL32-EX1

Code written for implementations that support longer restrictions need not be diagnosed.

Related guidelines

CERT C Secure Coding Standard: DCL32-C. Guarantee that mutually visible identifiers are unique

ISO/IEC TR 24772 "AJN Choice of Filenames and Other External Identifiers" and "YOW Identifier name reuse"

MISRA-C 2004, Rules 5.1 and 8.9

5.62 Tainted, potentially mutilated, or out-of-domain integer values are used in a taintedness sink (INT04-C)

Values that are tainted, potentially mutilated, or out-of-domain integers and are used in an integer taintedness sink shall be diagnosed because doing so can result in undefined or unexpected behavior.

Taintedness sinks for integers are

— in any pointer arithmetic, including array indexing;

— as a length or size of an object (for example, the size of a variable-length array);

— as the bound of access to an array (for example, a loop counter); and

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

EXAMPLE 1 In this 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 *str) {
  size_t size;
  GET_TAINTED_INT(size);
  char vla[size];  // diagnostic required

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

  /* ... */
}
```

EXAMPLE 2 In this 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() {
  int color_index;
  GET_TAINTED_INT(color_index);

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

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

Related guidelines

CERT C Secure Coding Standard: ARR32-C. Ensure size arguments for variable length arrays are in a valid range

CERT C Secure Coding Standard: INT04-C. Enforce limits on integer values originating from untrusted sources

ISO/IEC TR 24772 "XYX Boundary Beginning Violation" and "XYZ Unchecked Array Indexing"

Bibliography

[Griffiths 2006]

[Seacord 2005] Chapter 5, "Integer Security"

5.63 Using the sizeof operator on an expression that contains side effects (EXP06-C)

Using the `sizeof` operator on an expression that contains side effects shall be diagnosed because doing so is often a sign of programmer error. The `sizeof` operator does not evaluate its operand if the operand's type is not a variable-length array.

EXAMPLE In this example, a diagnostic is required because the `sizeof` operator is applied to an expression containing side effects, `a++`.

```
int f(int a) {
  return sizeof(a++);  // diagnostic required
}
```

Related guidelines

CERT C Secure Coding Standard: EXP06-C. Operands to the sizeof operator should not contain side effects

5.64 Using trigraphs (PRE07-C)

Trigraphs shall be diagnosed because their use can result in unexpected behavior.

EXAMPLE 1 In this example, a diagnostic is required because the trigraph sequence `??/` is used.

```
void f(int a) {
  // what is the value of a now??/
  a++;
  // diagnostic required
  /* ... */
}
```

EXAMPLE 2 In this example, a diagnostic is required because the trigraph sequence `??!` is used.

```
void g(size_t i) {
  if (i > 9000) {
    if (puts("Over 9000!??!") == EOF) {  // diagnostic required
      /* ... */
    }
  }
}
```

Related guidelines

CERT C Secure Coding Standard: PRE07-C. Avoid using repeated question marks

MISRA-C 2004, Rule 4.2

5.65 Wrapping unsigned integers (INT30-C)

Unsigned integer wrapping must be diagnosed when the resulting tainted, potentially mutilated value, or out-of-domain value, is used in a *taintedness sink*. Taintedness sinks include function arguments of type `size_t` or `rsize_t` or the size of a variable-length array.

A *taintedness sink* is an operation that can misbehave in a security-relevant manner, if supplied with certain operand values, so all operands to these operations must be implicitly or explicitly sanitized in some manner to ensure they do not have an unexpected value. Examples of *taintedness sinks* are memory allocation operations, such as `malloc` or VLA bounds, pointer or integer operands to pointer arithmetic (either `p+i` or `p[i]`), and buffer size arguments to library or system calls, such as `strncpy`, `read`, or `write`.

If an operation is performed on an untainted value that can produce either an undefined result, such as the result of signed integer overflow, or a defined but unexpected result, such as unsigned integer overflow, we say that the result of such an operation is *mutilated*. A mutilated value can be just as dangerous as a tainted value because it also can differ either in sign or dramatically in magnitude from what the programmer expects.

Using tainted or mutilated values as arguments to or operands of taintedness sinks shall be diagnosed because this can result in unexpected behavior.

The following table indicates which operators can result in wrapping.

Table 8 — Expressions and wrapping

Operator	Wrap	Operator	Wrap	Operator	Wrap	Operator	Wrap
`+`	✓	`-=`	✓	`<<`	✓	`<`	✗
`-`	✓	`*=`	✓	`>>`	✗	`>`	✗
`*`	✓	`/=`	✗	`&`	✗	`>=`	✗
`/`	✗	`%=`	✗	`|`	✗	`<=`	✗
`%`	✗	`<<=`	✓	`^`	✗	`==`	✗
`++`	✓	`>>=`	✗	`~`	✗	`!=`	✗
`--`	✓	`&=`	✗	`!`	✗	`&&`	✗
`=`	✗	`|=`	✗	unary +	✗	`||`	✗
`+=`	✓	`^=`	✗	unary -	✓	`?:`	✗

EXAMPLE In this example, a diagnostic is required because all of the arithmetic operations may result in unsigned integer wrapping. The results of the operations are used in taintedness sinks.

```
void f() {
  unsigned int ui1, ui2, result;
  char array[BUFSIZ];
  char *p, *q;

  GET_TAINTED_INT(ui1);
  GET_TAINTED_INT(ui2);

  result = ui1 + ui2;
  array[result] = 0;  // diagnostic required

  p = (char *)malloc(ui1 - ui2);  // diagnostic required

  result = ui1 * ui2;
  ++result;

  q = array + result;  // diagnostic required

  /* ... */
}
```

Related guidelines

CERT C Secure Coding Standard: INT30-C. Ensure that unsigned integer operations do not wrap

ISO/IEC TR 24772 "XYY Wrap-around Error"

MITRE CWE: CWE-190: Integer Overflow or Wraparound

Bibliography

[Dowd 2006] Chapter 6, "C Language Issues" (Arithmetic Boundary Conditions, pp. 211-223)

[Seacord 2005] Chapter 5, "Integers"

[Viega 2005] Section 5.2.7, "Integer overflow"

[VU#551436]

[Warren 2002] Chapter 2, "Basics"

[Wojtczuk 2008]

Annex A (normative) Intra- to Interprocedural Transformations

Rather than giving interprocedural examples of each relevant rule, the basic examples in many cases can be intraprocedural, and a set of interprocedural examples can be *derived* from those by applying various transformations to source code.

— Function arguments and return values

— Indirection

— Transformation involving standard library functions

— Example

A.1 Function arguments and return values

The simplest case is a rule involving only one value, such as Detect and Handle Input and Output Errors. The following is an intraprocedural example:

```
int result = write(fd, buf, length);
 if (result == length) /* checking for success */
     ...
```

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

```
void check_it(int length, int result)
 {
      if (result == length) /* checking for success */
          ...
 }


 ...
 check_it(length, write(fd, buf, length));
int xwrite(int fd, void *buf, int length)
 {
      return write(fd, buf, length); /* return for checking elsewhere */
 }


 ...
 int result = xwrite(fd, buf, length);
 if (result == length) /* checking for success */
     ...
```

A.2 Indirection

The next transformation is to add indirection:

```
void check_indirect(int length, int *result)
 {
     if (*result == length) /* checking for success */
         ...
 }
 ...
 int result = write(fd, buf, length);
 check_indirect(length, &result);
void return_result_thru_param(int fd, void *buf, int length, int *result)
 {
     *result = write(fd, buf, length);
 }


 ...
 int result;
 return_result_thru_param(fd, buf, length, &result);
 if (result == length) /* checking for success */
     ...
```

Indirection can also involve fields of structs or unions. Theoretically, indirection can be applied recursively, but modeling this causes scaling issues for many analysis frameworks.

When a rule involves multiple values, such as Do Not Use Invalid Array Indexing (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;
 array[index] = 0; /* violation */
```

Applying some of the interprocedural transformations yields

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


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

or

```
static int array[2];
 int *get_array()
 {
     return array;
 }


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

or

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


 void indexer(struct array_params *ap)
 {
     ap->array[ap->index] = 0;
 }
 ...
 int array[2];
 struct array_params params;
 params.array = array;
 params.index = 2;
 indexer(&params); /* violation */
```

One could argue that these violations actually involve four steps: the array, the index, the address arithmetic, and the dereference. In theory, each of these elements could occur in different functions:

```
int *add(int *base, int offset)
 {
   return base + offset;
 }


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

However, it is not clear whether we want to treat array indexing as an "atomic" operation or simply as the composition of address arithmetic and dereferencing.

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, in fact, modifies an element of the `const` array `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;
}
```

A.4 Example

Just for fun, let's put these all together and see just how non-obvious such a seemingly simple bug can be to diagnose:

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


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

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

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

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

 ...
 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); /* part of violation */
 store(&t, 0); /* violation */
```

Annex B (informative) Undefined Behavior

According to C99 (as summarized in Section 2 of Annex J therein), the behavior of a program is undefined in the circumstances outlined in the table below. The "Class" column in the table identifies the nature of the undefined behavior as outlined in Annex L of N1494, the Working Draft of the International Standard, Programming languages — C, from which the descriptions in the table below come. The parenthesized section numbers refer to the section of C99 that identifies the undefined behavior.

Symbol	**Classification**
❌	Critical Undefined Behavior
⚠	Bounded Undefined Behavior
ℹ	Undefined Behavior (information/confirmation needed)

Table B.1 — Undefined behaviors

UB	**Class**	**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 that is not immediately preceded by a backslash character or that 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	⚠	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).
6	⚠	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).
7	⚠	The same identifier has both internal and external linkage in the same translation unit (6.2.2).
8	❌	An object is referred to outside of its lifetime (6.2.4).
9	❌	The value of a pointer to an object whose lifetime has ended is used (6.2.4).
10	⚠	The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.8, 6.8).
11	⚠	A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).
12	⚠	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).
13	⚠	The arguments to certain operators could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).
14	⚠	Two declarations of the same object or function specify types that are not compatible (6.2.7).
15	⚠	Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

UB	Class	Description
16	⚠	Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).
17	✖	An lvalue does not designate an object when evaluated (6.3.2.1).
18	⚠	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).
19	⚠	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).
20	⚠	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).
21	⚠	Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).
22	⚠	Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).
23	✖	A pointer is used to call a function whose type is not compatible with the pointed-to type (6.3.2.3).
24	⚠	An unmatched `'` or `"` character is encountered on a logical source line during tokenization (6.4).
25	⚠	A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).
26	⚠	A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).
27	⚠	The initial character of an identifier is a universal character name designating a digit (6.4.2.1).
28	⚠	Two identifiers differ only in nonsignificant characters (6.4.2.1).
29	⚠	The identifier `__func__` is explicitly declared (6.4.2.2).
30	ℹ	The program attempts to modify a string literal (6.4.5).
31	⚠	The characters `'`, back-slash, `"`, `/`, or `/*` occur in the sequence between the `<` and `>` delimiters, or the characters `'`, back-slash, `//`, or `/*` occur in the sequence between the `"` delimiters, in a header name preprocessing token (6.4.7).
32	⚠	Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored (6.5).
33	⚠	An exceptional condition occurs during the evaluation of an expression (6.5).
34	⚠	An object has its stored value accessed other than by an lvalue of an allowable type (6.5).
35	ℹ	An attempt is made to modify the result of a function call, a conditional operator, an assignment operator, or a comma operator, or to access it after the next sequence point (6.5.2.2, 6.5.15, 6.5.16, 6.5.17).
36	⚠	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).
37	⚠	For 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).

UB	Class	Description
38	⚠	For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2).
39	⚠	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).
40	❌	The operand of the unary `*` operator has an invalid value (6.5.3.2).
41	⚠	A pointer is converted to other than an integer or pointer type (6.5.4).
42	⚠	The value of the second operand of the `/` or `%` operator is zero (6.5.5).
43	⚠	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 is just beyond, the same array object (6.5.6).
44	❌	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).
45	⚠	Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).
46	⚠	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).
47	⚠	The result of subtracting two pointers is not representable in an object of type `ptrdiff_t` (6.5.6).
48	⚠	An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).
49	⚠	An expression having signed promoted type is left-shifted, and either the value of the expression is negative or the result of shifting would be not be representable in the promoted type (6.5.7).
50	⚠	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).
51	⚠	An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).
52	⚠	An expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, `sizeof` expressions whose results are integer constants, or immediately cast floating constants; or contains casts (outside operands to `sizeof` operators) other than conversions of arithmetic types to integer types (6.6).
53	⚠	A constant expression in an initializer is not, or does not evaluate to, one of the following: an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for an object type plus or minus an integer constant expression (6.6).
54	⚠	An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, or `sizeof` expressions; or contains casts (outside operands to `sizeof` operators) other than conversions of arithmetic types to arithmetic types (6.6).
55	⚠	The value of an object is accessed by an array-subscript`[]`, member-access `.` or `->`, address `&`, or indirection `*` operator or a pointer cast in creating an address constant (6.6).
56	⚠	An identifier for an object is declared with no linkage, and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.7).
57	⚠	A function is declared at block scope with an explicit storage-class specifier other than `extern` (6.7.1).

UB	Class	Description
58	⚠	A structure or union is defined as containing no named members (6.7.2.1).
59	⚠	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).
60	⚠	When the complete type is needed, an incomplete structure or union type is not completed in the same scope by another declaration of the tag that defines the content (6.7.2.3).
61	ℹ	An attempt is made to modify an object defined with a `const`-qualified type through use of an lvalue with non-`const`-qualified type (6.7.3).
62	⚠	An attempt is made to refer to an object defined with a `volatile`-qualified type through use of an lvalue with non-`volatile`-qualified type (6.7.3).
63	⚠	The specification of a function type includes any type qualifiers (6.7.3).
64	⚠	Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).
65	⚠	An object that 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).
66	⚠	A `restrict`-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).
67	⚠	A function with external linkage is declared with an `inline` function specifier, but is not also defined in the same translation unit (6.7.4).
68	⚠	Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.5.1).
69	⚠	The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.5.2).
70	⚠	In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.5.2).
71	⚠	A declaration of an array parameter includes the keyword `static` within the[and `]` and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.5.3).
72	⚠	A storage-class specifier or type qualifier modifies the keyword `void` as a function parameter type list (6.7.5.3).
73	⚠	In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipsis terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list or when one type is specified by a function definition with an identifier list) (6.7.5.3).
74	⚠	The value of an unnamed member of a structure or union is used (6.7.8).
75	⚠	The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.8).
76	⚠	The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.8).
77	⚠	The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.8).

UB	Class	Description
78	⚠	An identifier with external linkage is used, but, in the program, there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9).
79	⚠	A function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list (6.9.1).
80	⚠	An adjusted parameter type in a function definition is not an object type (6.9.1).
81	⚠	A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.9.1).
82	⚠	The `}` that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).
83	⚠	An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).
84	⚠	The token defined is generated during the expansion of a `#if` or `#elif` preprocessing directive, or the use of the defined unary operator does not match one of the two specified forms prior to macro replacement (6.10.1).
85	⚠	The `#include` preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).
86	⚠	The character sequence in an `#include` preprocessing directive does not start with a letter (6.10.2).
87	⚠	There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.3).
88	⚠	The result of the preprocessing operator # is not a valid character string literal (6.10.3.2).
89	⚠	The result of the preprocessing operator ## is not a valid preprocessing token (6.10.3.3).
90	⚠	The `#line` preprocessing directive that results after expansion does not match one of the two well-defined forms, or its digit sequence specifies zero or a number greater than 2147483647 (6.10.4).
91	⚠	A non-`STDC` `#pragma` preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.6).
92	⚠	A `#pragma STDC` preprocessing directive does not match one of the well-defined forms (6.10.6).
93	⚠	The name of a predefined macro, or the identifier defined, is the subject of a `#define` or `#undef` preprocessing directive (6.10.8).
94	⚠	An attempt is made to copy an object to an overlapping object by use of a library function, other than as explicitly allowed (e.g., `memmove`) (clause 7).
95	⚠	A file with the same name as one of the standard headers, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files (7.1.2).
96	⚠	A header is included within an external declaration or definition (7.1.2).
97	⚠	A function, object, type, or macro that is specified as being declared or defined by some standard header is used before any header that declares or defines it is included (7.1.2).
98	⚠	A standard header is included while a macro is defined with the same name as a keyword (7.1.2).
99	⚠	The program attempts to declare a library function itself, rather than via a standard header, but the declaration does not have external linkage (7.1.2).

UB	Class	Description
100	⚠	The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).
101	⚠	The program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3).
102	✖	An argument to a library function has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).
103	ℹ	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).
104	⚠	The macro definition of `assert` is suppressed in order to access an actual function (7.2).
105	⚠	The argument to the `assert` macro does not have a scalar type (7.2).
106	⚠	The `CX_LIMITED_RANGE`, `FENV_ACCESS`, or `FP_CONTRACT` pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.3.4, 7.6.1, 7.12.2).
107	⚠	The value of an argument to a character handling function is neither equal to the value of `EOF` nor representable as an `unsigned char` (7.4).
108	⚠	A macro definition of `errno` is suppressed in order to access an actual object, or the program defines an identifier with the name `errno` (7.5).
109	⚠	Part of the program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the `FENV_ACCESS` pragma `"off"` (7.6.1).
110	⚠	The exception-mask argument for one of the functions that provide access to the floating-point status flags has a nonzero value not obtained by bitwise OR of the floating-point exception macros (7.6.2).
111	⚠	The `fesetexceptflag` function is used to set floating-point status flags that were not specified in the call to the `fegetexceptflag` function that provided the value of the corresponding `fexcept_t` object (7.6.2.4).
112	⚠	The argument to `fesetenv` or `feupdateenv` is neither an object set by a call to `fegetenv` or `feholdexcept`, nor is it an environment macro (7.6.4.3, 7.6.4.4).
113	⚠	The value of the result of an integer arithmetic or conversion function cannot be represented (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.20.6.1, 7.20.6.2, 7.20.1).
114	ℹ	The program modifies the string pointed to by the value returned by the `setlocale` function (7.11.1.1).
115	ℹ	The program modifies the structure pointed to by the value returned by the `localeconv` function (7.11.2.1).
116	⚠	A macro definition of `math_errhandling` is suppressed or the program defines an identifier with the name `math_errhandling` (7.12).
117	⚠	An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.14).
118	⚠	A macro definition of `setjmp` is suppressed in order to access an actual function, or the program defines an external identifier with the name `setjmp` (7.13).
119	⚠	An invocation of the `setjmp` macro occurs other than in an allowed context (7.13.2.1).
120	ℹ	The `longjmp` function is invoked to restore a nonexistent environment (7.13.2.1).

UB	Class	Description
121	⚠	After a `longjmp`, there is an attempt to access the value of an object of automatic storage class with non-`volatile`-qualified type, local to the function containing the invocation of the corresponding `setjmp` macro, that was changed between the `setjmp` invocation and `longjmp` call (7.13.2.1).
122	ℹ	The program specifies an invalid pointer to a signal handler function (7.14.1.1).
123	⚠	A signal handler returns when the signal corresponded to a computational exception (7.14.1.1).
124	ℹ	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).
125	⚠	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).
126	⚠	The value of `errno` is referred to after a signal occurred other than as the result of calling the `abort` or `raise` function and the corresponding signal handler obtained a `SIG_ERR` return from a call to the `signal` function (7.14.1.1).
127	ℹ	A signal is generated by an asynchronous signal handler (7.14.1.1).
128	⚠	A function with a variable number of arguments attempts to access its varying arguments other than through a properly declared and initialized `va_list` object, or before the `va_start` macro is invoked (7.15, 7.15.1.1, 7.15.1.4).
129	⚠	The macro `va_arg` is invoked using the parameter `ap` that was passed to a function that invoked the macro `va_arg` with the same parameter (7.15).
130	⚠	A macro definition of `va_start`, `va_arg`, `va_copy`, or `va_end` is suppressed in order to access an actual function, or the program defines an external identifier with the name `va_copy` or `va_end` (7.15.1).
131	⚠	The `va_start` or `va_copy` macro is invoked without a corresponding invocation of the `va_end` macro in the same function, or vice versa (7.15.1, 7.15.1.2, 7.15.1.3, 7.15.1.4).
132	⚠	The type parameter to the `va_arg` macro is not such that a pointer to an object of that type can be obtained simply by postfixing a `*` (7.15.1.1).
133	⚠	The `va_arg` macro is invoked when there is no actual next argument, or with a specified type that is not compatible with the promoted type of the actual next argument, with certain exceptions (7.15.1.1).
134	⚠	The `va_copy` or `va_start` macro is called to initialize a `va_list` that was previously initialized by either macro without an intervening invocation of the `va_end` macro for the same `va_list` (7.15.1.2, 7.15.1.4).
135	⚠	The parameter *parmN* of a `va_start` macro is declared with the register storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions (7.15.1.4).
136	⚠	The member designator parameter of an `offsetof` macro is an invalid right operand of the `.` operator for the type parameter, or designates a bit-field (7.17).
137	⚠	The argument in an instance of one of the integer-constant macros is not a decimal, octal, or hexadecimal constant, or it has a value that exceeds the limits for the corresponding type (7.18.4).
138	ℹ	A byte input/output function is applied to a wide-oriented stream, or a wide character input/output function is applied to a byte-oriented stream (7.19.2).

UB	Class	Description
139		Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.19.2).
140	ℹ	The value of a pointer to a `FILE` object is used after the associated file is closed (7.19.3).
141	ℹ	The stream for the `fflush` function points to an input stream or to an update stream in which the most recent operation was input (7.19.5.2).
142	⚠	The string pointed to by the mode argument in a call to the `fopen` function does not exactly match one of the specified character sequences (7.19.5.3).
143	ℹ	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 (7.19.5.3).
144	⚠	An attempt is made to use the contents of the array that was supplied in a call to the `setvbuf` function (7.19.5.6).
145	ℹ	There are insufficient arguments for the format in a call to one of the formatted input/output functions, or an argument does not have an appropriate type (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).
146	ℹ	The format in a call to one of the formatted input/output functions or to the `strftime` or `wcsftime` function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.19.6.1, 7.19.6.2, 7.23.3.5, 7.24.2.1, 7.24.2.2, 7.24.5.1).
147	⚠	In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.19.6.1, 7.24.2.1).
148	ℹ	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 (7.19.6.1, 7.24.2.1).
149	⚠	A conversion specification for a formatted output function uses a `#` or `0` flag with a conversion specifier other than those described (7.19.6.1, 7.24.2.1).
150	ℹ	A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).
151	ℹ	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) (7.19.6.1, 7.24.2.1).
152	⚠	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 (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).
153	⚠	A % conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly %% (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).
154	⚠	An invalid conversion specification is found in the format for one of the formatted input/output functions, or the `strftime` or `wcsftime` function (7.19.6.1, 7.19.6.2, 7.23.3.5, 7.24.2.1, 7.24.2.2, 7.24.5.1).
155	⚠	The number of characters transmitted by a formatted output function is greater than `INT_MAX` (7.19.6.1, 7.19.6.3, 7.19.6.8, 7.19.6.10).
156	ℹ	The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.19.6.2, 7.24.2.2).
157	ℹ	A `c`, `s`, or[conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is `s` or[]) (7.19.6.2, 7.24.2.2).

UB	Class	Description
158	ℹ	A `c`, `s`, or`[` conversion specifier with an `l` qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.19.6.2, 7.24.2.2).
159	⚠	The input item for a `%p` conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.19.6.2, 7.24.2.2).
160	ℹ	The `vfprintf`, `vfscanf`, `vprintf`, `vscanf`, `vsnprintf`, `vsprintf`, `vsscanf`, `vfwprintf`, `vfwscanf`, `vswprintf`, `vswscanf`, `vwprintf`, or `vwscanf` function is called with an improperly initialized `va_list` argument, or the argument is used (other than in an invocation of `va_end`) after the function returns (7.19.6.8, 7.19.6.9, 7.19.6.10, 7.19.6.11, 7.19.6.12, 7.19.6.13, 7.19.6.14, 7.24.2.5, 7.24.2.6, 7.24.2.7, 7.24.2.8, 7.24.2.9, 7.24.2.10).
161	⚠	The contents of the array supplied in a call to the `fgets`, `gets`, or `fgetws` function are used after a read error occurred (7.19.7.2, 7.19.7.7, 7.24.3.2).
162	⚠	The file position indicator for a binary stream is used after a call to the `ungetc` function where its value was zero before the call (7.19.7.11).
163	⚠	The file position indicator for a stream is used after an error occurred during a call to the `fread` or `fwrite` function (7.19.8.1, 7.19.8.2).
164	⚠	A partial element read by a call to the `fread` function is used (7.19.8.1).
165	⚠	The `fseek` function is called for a text stream with a nonzero offset, and either the offset was not returned by a previous successful call to the `ftell` function on a stream associated with the same file or `whence` is not SEEK_SET (7.19.9.2).
166	⚠	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 (7.19.9.3).
167	⚠	A non-null pointer returned by a call to the `calloc`, `malloc`, or `realloc` function with a zero requested size is used to access an object (7.20.3).
168	✖	The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is used (7.20.3).
169	ℹ	The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by `calloc`, `malloc`, or `realloc`, or the space has been deallocated by a call to `free` or `realloc` (7.20.3.2, 7.20.3.4).
170	⚠	The value of the object allocated by the `malloc` function is used (7.20.3.3).
171	⚠	The value of any bytes in a new object allocated by the `realloc` function beyond the size of the old object are used (7.20.3.4).
172	⚠	The program executes more than one call to the `exit` function (7.20.4.3).
173	⚠	During the call to a function registered with the `atexit` function, a call is made to the `longjmp` function that would terminate the call to the registered function (7.20.4.3).
174	ℹ	The string set up by the `getenv` or `strerror` function is modified by the program (7.20.4.5, 7.21.6.2).
175	ℹ	A command is executed through the `system` function in a way that is documented as causing termination or some other form of undefined behavior (7.20.4.6).
176	ℹ	A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.20.5).

UB	Class	Description
177	⚠	The comparison function called by a searching or sorting utility function alters the contents of the array being searched or sorted, or returns ordering values inconsistently (7.20.5).
178	⚠	The array being searched by the `bsearch` function does not have its elements in proper order (7.20.5.1).
179	⚠	The current conversion state is used by a multibyte/wide character conversion function after changing the `LC_CTYPE` category (7.20.7).
180	✖	A string or wide-string utility function is instructed to access an array beyond the end of an object (7.21.1, 7.24.4).
181	ℹ	A string or wide-string utility function is called with an invalid pointer argument, even if the length is zero (7.21.1, 7.24.4).
182	⚠	The contents of the destination array are used after a call to the `strxfrm`, `strftime`, `wcsxfrm`, or `wcsftime` function in which the specified length was too small to hold the entire null-terminated result (7.21.4.5, 7.23.3.5, 7.24.4.4.4, 7.24.5.1).
183	⚠	The first argument in the very first call to the `strtok` or `wcstok` is a null pointer (7.21.5.8, 7.24.4.5.7).
184	⚠	The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.22).
185	⚠	A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.22).
186	⚠	The argument corresponding to an `s` specifier without an `l` qualifier in a call to the `fwprintf` function does not point to a valid multibyte character sequence that begins in the initial shift state (7.24.2.11).
187	ℹ	In a call to the `wcstok` function, the object pointed to by `ptr` does not have the value stored by the previous call for the same wide string (7.24.4.5.7).
188	ℹ	An `mbstate_t` object is used inappropriately (7.24.6).
189	⚠	The value of an argument of type `wint_t` to a wide-character classification or case-mapping function is neither equal to the value of `WEOF` nor representable as a `wchar_t` (7.25.1).
190	⚠	The `iswctype` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctype` function that returned the description (7.25.2.2.1).
191	⚠	The `towctrans` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctrans` function that returned the description (7.25.3.2.1).

Bibliography

[Banahan 2003] Banahan, Mike; Brady, Declan; & Doran, Mark. *The C Book, Featuring the ANSI C Standard*. Addison-Wesley, 1991.

[Beebe 2005] Beebe, Nelson H. F. Re: Remainder (%) operator and GCC. http://gcc.gnu.org/ml/gcc-help/2005-11/msg00141.html (2005).

[Brainbell.com] Brainbell.com. Advice & Warnings for C Tutorials. http://www.brainbell.com/tutors/c/Advice_and_Warnings_for_C/ (2011)

[Bryant 2003] Bryant, Randal E., & O'Halloran, David. Computer Systems: A Programmer's Perspective. Prentice Hall, 2003 (ISBN 0-13-034074-X).

[CERT 2010] CERT C Secure Coding Standard https://www.securecoding.cert.org/confluence/x/HQE (2010).

[CERT/CC 2003] Finlay, Ian A. CERT Advisory CA-2003-16, Buffer Overflow in Microsoft RPC. http://www.cert.org/advisories/CA-2003-16.html (July 2003).

[Chess 2007] Chess, Brian, & West, Jacob. Secure Programming with Static Analysis. Addison-Wesley 2007.

[Coverity 2007] Coverity Prevent User's Manual (3.3.0), 2007.

[Dowd 2006] Dowd, M., McDonald, J., & Schuh, J. The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Boston, MA: Addison-Wesley, 2006.

[Drepper 2009 Drepper, Ulrich. Defensive Programming for Red Hat Enterprise Linux (and What To Do If Something Goes Wrong). http://web.sunybroome.edu/~antonakos_j/cst203/buffer/defprogramming.pdf (April 8, 2009).

[Flake 2006] Flake, Halvar. "Attacks on uninitialized local variables." http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf (2006).

[Fortify 2006] Fortify Software Inc. Fortify Taxonomy: Software Security Errors. https://www.fortify.com/vulncat/en/vulncat/index.html (2009).

[Griffiths 2006] Griffiths, Andrew. "Clutching at straws: When you can shift the stack pointer." http://arsouyes.org/index.php?id=248 (2006).

[Hatton 1995] Hatton, Les. Safer C: Developing Software for High-Integrity and Safety-Critical Systems. New York: McGraw-Hill Book Company, 1995 (ISBN 0-07-707640-0).

[Horton 1990] Horton, Mark R. Portable C Software. Upper Saddle River, NJ: Prentice-Hall, Inc., 1990 (ISBN:0-13-868050-7).

[IEC 61508-1-7: 2010] International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safety-related systems, Parts 1-7. IEC 61508, Ed. 2.0. International Electrotechnical Commission, 2010.

[IEEE Std 1003.1: 2008] Institute of Electrical and Electronics Engineers. The Open Group Base Specifications Issue 7 IEEE Std 1003.1, 2008 Edition. See also ISO/IEC 9945-2008 and Open Group 08. Institute of Electrical and Electronics Engineers, 2008.

[IEEE 754: 2006] Institute of Electrical and Electronics Engineers. Standard for Binary Floating-Point Arithmetic (IEEE 754-1985). Institute of Electrical and Electronics Engineers, 2006.

[ISO 4217: 2008] International Organization for Standardization. Codes for the representation of currencies and funds. Geneva, Switzerland: International Organization for Standardization, 2008.

[ISO 8601: 2004] International Organization for Standardization. Data elements and interchange formats – Information interchange – Representation of dates and times. Geneva, Switzerland: International Organization for Standardization, 2004.

[ISO/IEC 2003] International Organization for Standardization/International Electrotechnical Commission.. Rationale for International Standard — Programming Languages — C, Revision 5.10. Geneva, Switzerland: International Organization for Standardization, April 2003.

[ISO/IEC TR 24772: 2010] International Organization for Standardization/International Electrotechnical Commission. ISO/IEC TR 24772. Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use. Geneva, Switzerland: International Organization for Standardization, March 2010.

[Jack 2007] Jack, Barnaby. Vector Rewrite Attack. http://cansecwest.com/slides07/Vector-Rewrite-Attack.pdf. (May 2007).

[Kernighan 1988] Kernighan , Brian W., & Ritchie, Dennis M. The C Programming Language, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[Kettelwell 2002] Kettlewell, Richard. C Language Gotchas. http://www.greenend.org.uk/rjk/2001/02/cfu.html (February 2002).

[Kirch-Prinz 2002] Kirch-Prinz, Ulla & Prinz, Peter. C Pocket Reference. Sebastopol, CA: O'Reilly, November 2002 (ISBN: 0-596-00436-2).

[Koenig 1989] Koenig, Andrew. C Traps and Pitfalls. Reading, MA: Addison-Wesley Professional, January 1, 1989.

[Lai 2006] Lai, Ray. "Reading Between the Lines." OpenBSD Journal, October 2006.

[mercy 2006] mercy. Exploiting Uninitialized Data, January 2006.

[Microsoft 2003] Microsoft Security Bulletin MS03-026, "Buffer Overrun In RPC Interface Could Allow Code Execution (823980)," September 2003.

[Microsoft 2007] C Language Reference, 2007.

[MISRA 2004] Motor Industry Software Reliabililty Association. MISRA-C 2004: Guidelines for the Use of the C Language in Critical Systems. MISRA 2004.

[MIT 2005] MIT. "MIT krb5 Security Advisory 2005-003," 2005.

[MITRE 2007] MITRE. Common Weakness Enumeration, Draft 9, April 2008.

[MITRE 2011] Common Vulnerabilities and Exposures List. http://cve.mitre.org/cve/cve.html (2011).

[MSDN 2011] Microsoft Developer Network. http://msdn.microsoft.com/en-us/ms348103 (2011).

[Murenin 2007] Murenin, Constantine A. "cnst: 10-year-old pointer-arithmetic bug in make(1) is now gone, thanks to malloc.conf and some debugging," June 2007.

[NAI 1998] Network Associates Inc. Bugtraq: Network Associates Inc. Advisory (OpenBSD), 1998.

[NIST 2006] NIST. SAMATE Reference Dataset, 2006.

[OpenBSD] Berkley Software Design, Inc. Manual Pages, June 2008.

[Open Group 2004] The Open Group and the IEEE. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, 2004.

[OWASP 2011] Open Web Application Security Project. OWASP Foundation, 2011.

[Pethia 2003] Pethia, Richard D. "Viruses and Worms: What Can We Do About Them?" September 10, 2003.

[Plum 1985] Plum, Thomas. *Reliable Data Structures in C*. Kamuela, HI: Plum Hall, Inc., 1985 (ISBN 0-911537-04-X).

[Plum 1989] Plum, Thomas, & Saks, Dan. *C Programming Guidelines, 2nd ed*. Kamuela, HI: Plum Hall, 1989 (ISBN 0911537074).

[Plum 1991] Plum, Thomas. *C++ Programming*. Kamuela, HI: Plum Hall, 1991 (ISBN 0911537104).

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

[Spinellis 2006] Spinellis, Diomidis. Code Quality: The Open Source Perspective. Addison-Wesley, 2006.

[van Sprundel 2006] van Sprundel, Ilja. Unusualbugs, 2006.

[Summit 1995] Summit, Steve. *C Programming FAQs: Frequently Asked Questions*. Boston, MA: Addison-Wesley, 1995 (ISBN 0201845199).

[Summit 2005] Summit, Steve. comp.lang.c Frequently Asked Questions, 2005.

[Sun 2005] C User's Guide. 819-3688-10. Sun Microsystems, Inc., 2005.

[Viega 2003] Viega, John, & Messier, Matt. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003 (ISBN 0-596-00394-3).

[Viega 2005] Viega, John. CLASP Reference Guide Volume 1.1. Secure Software, 2005.

[VU#551436] Giobbi, Ryan. Vulnerability Note VU#551436, *Mozilla Firefox SVG viewer vulnerable to buffer overflow,* 2007.

[VU#623332] Mead, Robert. Vulnerability Note VU#623332, *MIT Kerberos 5 contains double free vulnerability in "krb5_recvauth()" function,* 2005.

[Warren 2002] Warren, Henry S. Hacker's Delight. Boston, MA: Addison Wesley Professional, 2002 (ISBN 0201914654).

[Wheeler 2003] Wheeler, David. Secure Programming for Linux and Unix HOWTO, v3.010, March 2003.

[Wheeler 2004] Wheeler, David. Secure programmer: Call components safely. December 2004.

[Wojtczuk 2008] Wojtczuk, Rafal. "Analyzing the Linux Kernel vmsplice Exploit." McAfee Avert Labs Blog, February 13, 2008.

[xorl 2009] xorl. xorl %eax, %eax.

[Zalewski 2001] Zalewski, Michal. Delivering Signals for Fun and Profit: Understanding, exploiting and preventing signal-handling related vulnerabilities, May 2001.