ISO/IEC JTC 1/SC 22/WG 14 N 1609

Date: 2012-03-15

ISO/IEC TS 17961

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

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Technical Specification
Document subtype:
Document stage: 20 (Preparatory Stage)
Document language: E

Copyright notice

Contents

Page

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 17961 was prepared by ISO/IEC Joint Technical Committee 1, Subcommittee 22, Working Group 14.

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.

The application of static analysis to security has evolved in an ad hoc manner. This is useful from the point of view of exploring a new market to see what works. However, it has resulted in a fragmented market, with different vendors addressing different security issues and no way for a purchaser to specify the minimum requirements of a static analysis tool. Now that the shape of security needs is becoming clearer, there is a need for a specification that says, “For an analysis tool to conform to this specification, it must be able to do at least *this much*,” where *this much* is well specified. By imposing a floor on analysis capabilities rather than circumscribing them completely, a specification can allow for continued improvements while still giving customers a way to know what they are buying.

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

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

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

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

This Technical Specification has two major subdivisions:

— preliminary elements (clauses 1–4) and

— secure coding rules (clause 5).

Annexes provide additional information. A bibliography lists documents that were referred to during the preparation of the standard.

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 analyzer. 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.)

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

Two distinct kinds of examples are provided:

- *noncompliant examples* demonstrating language constructs that have weaknesses with potentially exploitable security implications; such examples are expected to elicit a diagnostic from a conforming analyzer for the affected language construct; and
- *compliant examples* are expected not to elicit a diagnostic.

Examples are not intended to be complete programs. For 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.

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. These rules may be extended in an implementation-dependent manner.

A conforming analyzer shall issue at least one diagnostic for a program containing one or more violations of the rules in this specification.

For each distinct rule in this Technical Specification, a conforming analyzer shall be capable of producing a distinct diagnostic.

NOTE The diagnostic message might be of the form:

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

Conformance is evaluated by testing the ability of analyzers to diagnose all violations of the rules represented by the noncompliant examples and not diagnose compliant examples and exceptions in the whole program. Conforming analyzers shall diagnose transformations of these rules requiring interprocedural analysis.

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

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

2.1 Completeness and soundness

To the greatest extent possible, an analyzer should be both complete and sound with respect to enforceable rules. An analyzer is considered sound (with respect to a specific rule) if it does not give a false-negative result, meaning it is able to find all violations of a rule 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 rule are outlined in Table 1.

Table 1—Completeness and soundness

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

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

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 analyzers that are in the extremes. An analyzer can report all of the lines in the program and have no false negatives at the expense of large numbers of false positives. Conversely, an analyzer can report nothing and have no false positives at the expense of not reporting real defects that could be detected automatically. Analyzers 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. Analyzers 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 conforming analyzers minimize false-positive diagnostics is a quality of implementation issue. In other words, quantitative thresholds for false positives and false negatives are outside the scope of this Technical Specification.

Analyzers are trusted processes, meaning that developers rely on their output. Consequently, developers must ensure that this trust is not misplaced. To earn this trust, the analyzer supplier should, ideally, run appropriate validation tests. Although 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 C implementation. An analyzer need not diagnose a rule violation if the result is documented for the target implementation and does not cause a security flaw.

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 mismatch might not be a problem for all target implementations, but it is a portability problem because not all implementations have the same representation for `int` and `long`.

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 C-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 values from those that do not involve tainted values.

2.4 Taint analysis

2.4.1 Taint and tainted sources

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

Tainted sources include

- parameters to the `main` function,
- the returned values 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 Restricted sinks

Operands and arguments whose domain is a subset of the domain described by their types are called *restricted sinks*. Any pointer arithmetic operation involving an integer operand is a restricted sink for that operand. Certain parameters of certain library functions are restricted sinks because these functions perform address arithmetic with these parameters, or control the allocation of a resource, or pass these parameters on to another taintededness sink. All string input parameters to library functions are restricted sinks because those strings are required to be null-terminated, with the exception of `strncpy` and `strncpy_s`, which explicitly allow the source argument not to be null-terminated. Loop bounds are not restricted sinks.

2.4.3 Propagation

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

2.4.4 Approaches to analysis

By definition, any tainted value flowing into a restricted sink is a security issue, so all such cases must be diagnosed. Diagnosing these violations requires some form of data flow analysis. In its most basic form, such an analysis operates intraprocedurally to determine which local tainted sources flow into local restricted sinks. Intraprocedural analysis is limited and can be extended by interprocedural analysis. Interprocedural analysis can be accomplished by top-down or bottom-up 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 restricted sinks would themselves be treated as restricted 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 description 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, it must be *sanitized* to ensure that it is in the defined domain of any restricted sink into which it flows. Sanitization is performed by *replacement* or *termination*. In replacement, out-of-domain values are replaced by in-domain values, and processing continues using an in-domain value in place of the original. In termination, the program logic terminates the path of execution when an out-of-domain value is detected, often simply by branching around whatever code would have used the value.

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

2.4.6 Tainted source macros

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

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

#define GET_TAINTED_INTEGER(type, val)           \
  do {                                           \
    const char *taint = getenv("TAINT");         \
    if (taint == 0) {                            \
      exit(1);                                   \
    }                                            \
                                                 \
    errno = 0;                                   \
    long tmp = strtol(taint, 0, 10);             \
    if ((tmp == LONG_MIN || tmp == LONG_MAX) &&  \
        errno == ERANGE)                         \
      ; // retain LONG_MIN or LONG_MAX           \
    val = tmp & ~(type)0;                        \
  } 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 9899:2011] Programming Languages—C.

[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).

[IEEE 60559:2011] 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 9899:2011, 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
mechanism that diagnoses coding flaws in software programs

NOTE Analyzers may include static analysis tools, tools within a compiler suite, and code reviewers.

4.2
asynchronous-safe function
asynchronous-signal safe
function that can be called safely and without side effects from within a signal-handler context

NOTE The function must be interruptible at any point to run linearly out of sequence without causing an inconsistent state and must also function properly when global data might itself be in an inconsistent state.

4.3
data flow analysis
tracking of values along specific paths through the code

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

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

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

NOTE 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.

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

- *if E is a subtraction expression (`-`) whose operands both are of pointer type, then T is the type pointed to by the operands of the expression;*
- 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

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

4.7
mutilated

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

EXAMPLE:

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

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

NOTE 2 Mutilated values cannot be sanitized.

4.8
out-of-domain value

one of a set of values that is not in the domain of a particular operator or function

4.9
persistent signal handler

signal handler running on an implementation that reinstalls it each time it is called (for example, by calling `signal`)

NOTE A persistent signal handler need only be installed once; a signal handler is non-persistent when run on a platform where it is not automatically reinstalled, meaning the programmer has to reinstall the handler to catch each signal.

4.10
restricted sink

argument or operand into which a tainted value flows, requiring a diagnostic

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

NOTE 2 Different restricted sinks may impose different validity constraints for the same value; a given value can be tainted with respect to one restricted sink but sanitized (and consequently no longer tainted) with respect to a different restricted sink.

NOTE 3 Specific restricted sinks and requirements for sanitizing tainted values are described in specific rules dealing with taint analysis (see 5.9, 5.16, 5.17, 5.26, 5.31, and 5.46).

4.11
sanitize

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

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

EXAMPLE A string must be null-terminated if it is passed as an argument to the `strlen` function.

4.12
security flaw
software defect that poses a potential security risk

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

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

4.15
tainted value
value derived from untrusted tainted data that has not been sanitized

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

4.17
UB
undefined behavior

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

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

4.20
untrusted data
data originating from an untrusted source; for analysis purposes, any input external to the program

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

NOTE 1 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 C, sec. 6.5.8, paragraph 4.)

NOTE 2 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 C, sec. 6.3.2.3, paragraph 7.)

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

5 Rules

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

C, section 6.5, paragraph 7, states

> *An object shall have its stored value accessed only by an lvalue expression that has one of the following types:*
>
> — *a type compatible with the effective type of the object,*
>
> — *a qualified version of a type compatible with the effective type of the object,*
>
> — *a type that is the signed or unsigned type corresponding to the effective type of the object,*
>
> — *a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,*
>
> — *an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or*
>
> — *a character type.*

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

According to section 6.2.6.1 of C,

> *Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined.*

Accessing an object through a pointer to an incompatible type (other than `unsigned char`) is undefined behavior (see undefined behavior 37 in Annex B) and shall be diagnosed.

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

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

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

    (*ip)++;  // diagnostic required

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

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 [accfree]

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.

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

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

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

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

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

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

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

  char *return_val = 0;

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

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

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

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

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 [accsig]

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 accessing such objects results in undefined behavior.

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

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

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

```
#define MAX_MSG_SIZE 24
char *err_msg;

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

int main(void) {

  signal(SIGINT, handler);

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

  /* Main code loop */

  return EXIT_SUCCESS;
}
```

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 Adding or subtracting a byte count to an element pointer [cntradd]

Adding to or subtracting from a pointer an integer value whose derived type is incompatible with the type pointed to by the pointer shall be diagnosed.

Similarly, dividing the difference of two pointers by an integer whose derived type is incompatible with that of the pointers shall be diagnosed.

NOTE No diagnostic is required if the pointer is a pointer to a character type.

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

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

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

  return 0;
}

void collect_longs() {
  long buf[BUFSIZ];
  long *buf_ptr = buf;
  while (buf_ptr < (buf + sizeof(buf))) {  // diagnostic required
    long data;
    if (get_long(&data) != 0) {
      break;
    }

    *buf_ptr++ = data;
  }

  /* ... */
}
```

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the difference of two element pointers `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 noncompliant 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 noncompliant 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: ";
  wchar_t *return_val = L’\0’;

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

    /* ... */
  }
}
```

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] *Secure Coding in C and C++*

5.5 No assignment in conditional expressions [boolasgn]

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

- `if` (controlling expression)
- `while` (controlling expression)
- `do ... while` (controlling expression)
- `for` (second operand)
- `?:` (first operand)
- `&&` (either operand)
- `||` (either operand)
- `?:` (second or third operands) or comma operator (second operand), where that operand is used in any of these contexts

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

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

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

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

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

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

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

Exceptions

— EX1: Assignment is permitted where the result of the assignment is itself a parameter to a comparison expression or relational expression and need not be diagnosed.

EXAMPLE This example shows an acceptable use of this exception.

```
if ( ( x = y )  != 0  ) { }
```

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

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

```
if ( ( x = y ) ) { }
```

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

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

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

EXAMPLE This example shows an acceptable use of this exception.

```
if ( foo( x = y ) ) { }
```

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-481: Assigning instead of Comparing

Bibliography

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

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

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.

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

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

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

```
#define MAXLINE 1024

char info[MAXLINE];

void log_message() {
  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 EXIT_SUCCESS;
}
```

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

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

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

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

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

  return EXIT_SUCCESS;
}
```

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

```
#define MAXLINE 1024

static jmp_buf env;

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

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

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

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

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

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

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

  char *info1;
  char *info2;

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

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

  return EXIT_SUCCESS;
}
```

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”

5.7 Calling functions with incorrect arguments [argcomp]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
/* in somefile.c */

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

/* in otherfile.c */

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

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”

5.8 Calling signal from interruptible signal handlers [sigcall]

Calling `signal` from within a signal handler whose execution can be interrupted by receipt of a signal on platforms where `signal` handlers are non-persistent shall be diagnosed because doing so presents a race window.

EXAMPLE In this noncompliant 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

5.9 Calling system [syscall]

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

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

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

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

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

```
void g() {
  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"

5.10 Comparing function addresses to zero [funcaddr]

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 noncompliant example, a diagnostic is required because the address of the function `thrd_current` is compared to `0`.

```
void f() {
  if (thrd_current == 0) {  // diagnostic required
    /* ... */
  }

  /* ... */
}
```

EXAMPLE 2 In this compliant example, a diagnostic is not required because the address of the function `thrd_current` is compared to null function pointer of the same type.

```
typedef thrd_t (*thrd_current_t)(void);

void f() {
  if (thrd_current == (thrd_current_t)0) {
    /* ... */
  }

  /* ... */
}
```

EXAMPLE 3 In this noncompliant 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”

5.11 Comparison of padding data [padcomp]

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

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

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

  return 0;
}
```

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

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

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 noncompliant 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 noncompliant 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

- EX1: A null pointer can be converted to an integer; it takes on the value 0. Likewise, a 0 integer can be converted to a pointer; it becomes the null pointer.
- EX2: Any valid pointer to `void` can be converted to `intptr_t` or `uintptr_t` and back with no change in value. (This includes the underlying types if `intptr_t` and `uintptr_t` are `typedefs`, and any `typedefs` that denote the same types as `intptr_t` and `uintptr_t`.)

EXAMPLE

```
void h() {
  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

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

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 it results in undefined behavior if the actual value is unaligned with respect to the destination type.

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

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

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

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

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

  c_ptr = (char *)&i;
  i_ptr = (int *)c_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*

5.14 Copying a FILE object [filecpy]

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 C, section 7.21.3, paragraph 6,

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

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

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

Related guidelines

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
/* in a.c */
```

```
extern int h(int a);  // diagnostic required

int main(void) {
  return h(10);
}

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

EXAMPLE 4 In this noncompliant example, a diagnostic is required on implementations where the external identifiers `bash_groupname_completion_function` and `bash_groupname_completion_func` are identical, because it 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 The identifier `bash_groupname_completion_function` referenced here was taken from GNU Bash version 3.2.

Exception

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

EXAMPLE

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

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

Related guidelines

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

Bibliography

[Hatton 1995] Section 2.8.3

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

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

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

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

5.17 Dividing by zero [divzero]

Tainted 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 noncompliant 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_INTEGER(int, y);

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

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

```
int modulus(int x) {
  int y;
  GET_TAINTED_INTEGER(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”

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

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

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

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

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

  return EXIT_SUCCESS;
}
```

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

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

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

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

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

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

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

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)

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

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

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

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

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

/* ... */

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

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

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

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

A `switch` statement that causes control to jump past the last statement in the `switch` body shall be diagnosed because this indicates logical incompleteness.

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

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

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

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”

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

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

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

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

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

  FILE *f = fopen(filename, "r");  // diagnostic required
  if (f == NULL) {
    /* ... */
  }
  /* ... */
  return 0;
}
```

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

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

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

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]

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

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 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[a]
fgetpos	zero	non-zero
fgets	pointer to string	NULL
fgetwc	wide character read	WEOF[a]
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[b]
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[a]
getchar	character read	EOF[a]
getenv	pointer to string	NULL
getenv_s	pointer to string	NULL
gets_s	pointer to string	NULL
getwc	wide character read	WEOF
getwchar	wide character read	WEOF
gmtime	pointer to broken-down time	NULL
gmtime_s	pointer to broken-down time	NULL
localtime	pointer to broken-down time	NULL
localtime_s	pointer to broken-down time	NULL
malloc	pointer to space	NULL
mblen, s != NULL	number of bytes	−1
mbrlen, s != NULL	number of bytes or status	(size_t)(-1)
mbrtoc16	number of bytes or status	(size_t)(-1), errno == EILSEQ
mbrtoc32	number of bytes or status	(size_t)(-1), errno == EILSEQ
mbrtowc, s != NULL	number of bytes or status	(size_t)(-1), errno == EILSEQ
mbsrtowcs	number of non-null elements	(size_t)(-1), errno == EILSEQ
mbsrtowcs_s	zero	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[b]
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
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
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)
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
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 feof and ferror.

[b] Use ferror.

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

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

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

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

Table 3—Example library functions and returns

Function	Successful return	Error return
printf	number of characters (non-negative)	negative
putchar	character written	EOF
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

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

EXAMPLE This example shows an acceptable use of this exception.

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

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

Related guidelines

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

MITRE CWE, CWE-391: Unchecked Error Condition

Bibliography

[Kettlewell 2002] Section 6, “I/O error checking”

[Seacord 2005] Chapter 7, “File I/O”

5.23 Forming invalid pointers by library function [libptr]

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.

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 109 in Annex B).

5.23.1 Library functions that take a pointer and an integer

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 size of the object referenced by p, a call to `f` where the effective type of `*p` is not compatible with the derived type of the expression n or `unsigned char` shall be diagnosed.

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

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

5.23.2 Library functions that take two pointers and an integer

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

For a function g taking the triple of not necessarily consecutive arguments (p, q, n), where p is a (possibly `void`) pointer, q is a `const`-qualified (possibly `void`) pointer, and n is an integer that specifies the 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.

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

The following standard library functions take two pointer arguments and a size argument, with the constraint that both pointers must point to valid memory objects of at least size bytes.

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

5.23.3 Library functions that take a pointer and two integers

For a function `g` taking the triple of not necessarily consecutive arguments (`p`, `m`, `n`), where `p` is a (possibly `void`) pointer, and `m` and `n` are integers that specify the size of an object, a call to `g` where the product of `m * n` is greater than the minimum of the number of remaining bytes of `*p` shall be diagnosed.

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

`bsearch`

`qsort`

`bsearch_s`

`qsort_s`

5.23.4 Standard memory allocation functions

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

- In the right operand of an assignment to an object of type `T *`, or
- In an initializer for an object of type `T *`, or
- In an expression that is passed as an argument of type `T *`, or

- In the expression of a return statement for a function returning type `T *`.

A call to a standard memory allocation function taking a size integer argument `n` and presumed to be intended for type `T *` shall be diagnosed when at least one of the following is true.

- `n < sizeof(T)`, or
- `T` is compatible with neither the derived type of the expression `n` nor `unsigned char`.

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

`aligned_alloc`

`calloc`

`malloc`

`realloc`

NOTE For purpose of this rule, the term *size* refers, for a declared object, to the size of the object, and for an allocated object, to the amount of the allocated storage. 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 size of the object referenced by `p`, a call to `f` where `n` is greater than the number of remaining bytes in the object referenced by `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 number of remaining bytes 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;
  if (p) {
    memset(p, 0, n); // diagnostic required
    /* ... */
  }
}
```

EXAMPLE 2 The size of `*p` in the following function definition is equal to `4 * sizeof(float)`. Assuming an implementation where the `sizeof(int) == sizeof(float)`, this value is also 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

  /* ... */
}
```

EXAMPLE 3 In the following function definition, assume `(sizeof(int) < sizeof(double))` holds. The size of `*p` is equal to `sizeof(int)`, the size of `*q` is equal to `sizeof(double)`, and `n` is equal to `sizeof(int)`. Consequently, `n` is less than or equal to the minimum of the size of `*p` and the 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 f3(int *a) {
  double b = 3.14;
  const size_t n = sizeof(*a);
  void *p = a;
  void *q = &b;

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

EXAMPLE 4 In the following function definition, assume that the size of `*p` and the 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 f4(char p[], const char *q) {
  const size_t n = sizeof(p);  // diagnostic required
  if ((memcpy(p, q, n)) == p) {
    /* ... */
  }

  /* ... */
}
```

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 initializer 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 incompatible with both `wchar_t` and `unsigned char`. Consequently, the expression is diagnosed.

```
wchar_t *f5() {
  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;
}
```

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

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

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

UB	Description
46	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).
47	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that

	points just beyond the array object and is used as the operand of a unary `*` operator that is evaluated (6.5.6).
49	An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression *`a[1][7]`* given the declaration *`int a[4][5]`*) (6.5.6).
62	An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).
109	The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

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

```
#define TABLESIZE 100

static int table[TABLESIZE];

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

  return NULL;
}
```

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

```
#define TABLESIZE 100

static int table[TABLESIZE];

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

  return NULL;
}
```

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

```
#define TABLESIZE 100

static int table[TABLESIZE];

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

  return NULL;
}
```

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

```
#define MAX_MACHINE_NAME_LENGTH 64

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

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

  *machine_name = '\0';

  return machine_name;
}
```

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

```
#define MAX_MACHINE_NAME_LENGTH 64

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

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

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

  *machine_name = '\0';

  return machine_name;
}
```

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

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

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

    size = pos + 1;
    table = tmp;
  }

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

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

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

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

    size  = pos + 1;
    table = tmp;
  }

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

EXAMPLE 8 In this noncompliant example, a diagnostic is required because a value is stored beyond the end of the arrays `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 noncompliant example, a diagnostic is required because the expression `first++` results in a pointer beyond the end of the array `buf` when `buf` contains no elements.

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

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

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

  return NULL;
}

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

  /* ... */
}
```

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

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

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

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”

5.25 Freeing memory multiple times [dblfree]

Freeing memory multiple times shall be diagnosed (subject to the following exception) because this results in “double-free” vulnerabilities [Seacord 2005].

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

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

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

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

```
void g(char *c_str1, size_t size) {
  char *c_str2 = (char *)realloc(c_str1, size);
  if (c_str2 == NULL) {
    free(str1);  // diagnostic required
    return;
  }
}
```

According to C, section 7.22.3, paragraph 1,

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

And according to section 7.22.3.5, paragraph 3,

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

If `realloc` is called with `size` equal to `0`, then if a `NULL` pointer is returned, the old value should be unchanged. However, there are some common but 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

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]

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

Invoking any of the formatted input/output functions identified in C, section 7.21.6, 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 noncompliant 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 noncompliant example, a diagnostic is required because the string `user` may contain a tainted value.

```
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);
  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”

5.27 Incorrectly setting and using errno [inverrno]

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.

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

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

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

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

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

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

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

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

Function name	Return value	errno value
fgetpos	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.

5.27.3 Library functions that may or may not set errno

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

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

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

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

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

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

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

5.27.4 Library functions that do not explicitly set errno

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

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

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

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

  /* ... */
}
```

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

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

EXAMPLE 3 In this noncompliant example, a diagnostic is required because `errno` is used for error checking and `errno` is checked after the call to the C Standard Library function `setlocale`.

```
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

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

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 C, section 7.21.5.3, paragraph 7,

> *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 151, Annex B.)

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

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

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

  if (fwrite(append_data, 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

5.29 Modifying string literals [strmod]

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 noncompliant example, a diagnostic is required because the string literal "string literal" is modified through the pointer p.

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

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

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

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

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

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

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

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

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

Exception

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

EXAMPLE

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

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

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

Modifying the objects or strings returned by the library functions listed in the following table shall be diagnosed because such modification results in undefined behavior.

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

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

EXAMPLE 1 In this noncompliant 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;

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

  /* ... */
}
```

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

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

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

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

  /* ... */
}
```

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

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

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

    /* use shell_dir */
  }
}
```

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

```
const char *f4(int error) {
  char buf[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

5.31 Overflowing signed integers [intoflow]

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

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

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

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

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

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

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

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,” 211–223)

[Seacord 2005] Chapter 5, “Integers”

[Viega 2005] Section 5.2.7, “Integer overflow”

[VU#551436]

[Warren 2002] Chapter 2, “Basics”

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

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 noncompliant example, a diagnostic is required because the parameter to `isspace`, `*t`, may not be representable as an `unsigned char`.

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

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

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

[Kettlewell 2002] Section 1.1, “<ctype.h> and characters types”

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

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

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

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

EXAMPLE 2 In this noncompliant example, a diagnostic is required because the pointer `src` is twice a `restrict`-qualified pointer parameter to `dual_memcpy`, referencing overlapping objects.

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

  return s1;
}

void f() {
  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”

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

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

```
#define BUFSIZE 256

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

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

```
#define MAX_ALLOCATION 1000

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

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

Exception

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

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 9899:2011] Section 7.22.3.3, “The free function,” Section 7.22.3.5, “The realloc function”

5.35 Referencing uninitialized memory [uninitref]

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

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

  if (number > 0) {
    *sign = 1;
  } else if (number < 0) {
    *sign = -1;
  }
}

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

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

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

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

  /* ... */
  return result;
}

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

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

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

  return EXIT_SUCCESS;
}
```

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

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

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

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

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

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

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]

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

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 the following exceptions) because this results in undefined behavior. The relational operators are >, <, >=, and <=.

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

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

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

```
#define SIZE 256

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

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

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

Exceptions

- EX1: Comparing two pointers within the same object does not need to be diagnosed.
- EX2: Subtracting two pointers to `char` within the same object does not need to be diagnosed.

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”

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

Tainted strings, wide or narrow, that are passed as the source argument to the `strcpy`, `strcat`, `wcscpy`, or `wcscat` function, and which exceed the size of the destination array, shall be diagnosed because doing so can result in writing to memory that is outside the bounds of existing objects.

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

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

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

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

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

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

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

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”

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

Passing a value that is tainted in any way as an argument to a call through a pointer to a function that was declared without a prototype shall be diagnosed because such a pointer may refer to a function whose parameters ultimately flow into a restricted sink. When a prototype is available, an analyzer might be able to determine if such a pointer points to such a function.

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

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

void (*pf)() = restricted_sink;

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

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

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

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

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

The standard C formatted input functions declared in `<stdio.h>` can corrupt memory if a tainted value is provided to the `%s` format specifier. Calls to the `fscanf`, `scanf`, `vfscanf`, and `vscanf` functions that pass tainted values as arguments and that can result in writes outside the bounds of the specified object shall be diagnosed. Calls to the `sscanf` and `vsscanf` functions that can result in writes outside the bounds of the specified object shall also be diagnosed when the input string is tainted.

The standard C `sprintf` function can also corrupt memory if supplied with tainted values. Calls to the `sprintf` function that can result in writes outside the bounds of the destination array shall be diagnosed when any of its variadic arguments are tainted.

A tainted value is one that originates from a tainted source, as defined in Section 2.4.1.

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

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

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

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

Related guidelines

CERT C Secure Coding Standard, STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

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

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

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

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

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

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

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

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

  return file;
}
```

Related guidelines

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

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

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

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

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

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

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

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

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

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

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

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

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

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”

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

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

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

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

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

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

```
void g() {
  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

Bibliography

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

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

According to C, 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 the future library directions) and `errno` are always reserved for use as identifiers with external linkage.
- Each identifier with file scope listed in any of the above subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

No other identifiers are reserved. The behavior of a program that declares or defines an identifier in a context in which it is reserved or defines a reserved identifier as a macro name is undefined. See also undefined behavior 106 of Annex B. 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 C to include an open-ended set of its own [IEEE Std 1003.1: 2008].

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

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

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

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

/* contents of <my_header.h> */

#endif /* _MY_HEADER_H_ */
```

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

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

/* contents of <my_header.h> */

#endif /* MY_HEADER_H */
```

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

```
static const 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.

```
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 noncompliant 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.

```
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.

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

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

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

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

```
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 9899:2011] Section 7.1.3, “Reserved identifiers”

[IEEE Std 1003.1: 2008] Section 2.2, “The compilation environment”

5.45 Using invalid format strings [invfmtstr]

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 noncompliant 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

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

Values that are tainted, potentially mutilated, or out-of-domain integers and are used in an integer **Error! Reference source not found.** shall be diagnosed because doing so can result in accessing memory that is outside the bounds of existing objects.

Restricted sinks for integers are

- any pointer arithmetic, including array indexing;
- a length or size of an object (for example, the size of a variable-length array);
- 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 noncompliant example, a diagnostic is required because the tainted integer `size` is used to declare the size of the variable length array `vla`.

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

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

  /* ... */
}
```

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

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

const char *set_background_color() {
  int color_index;
  GET_TAINTED_INTEGER(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
- 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”

Annex A
(informative)
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 C (as summarized in Section 2 of Annex J therein), the behavior of a program is undefined in the circumstances outlined in Table B.1. The parenthesized section numbers refer to the section of C that identifies the undefined behavior.

Table B.1—Undefined behaviors

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

22	An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).
23	An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to `void`) is applied to a void expression (6.3.2.2).
24	Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).
25	Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).
26	A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).
27	An unmatched `'` or `"` character is encountered on a logical source line during tokenization (6.4).
28	A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).
29	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).
30	The initial character of an identifier is a universal character name designating a digit (6.4.2.1).
31	Two identifiers differ only in nonsignificant characters (6.4.2.1).
32	The identifier `__func__` is explicitly declared (6.4.2.2).
33	The program attempts to modify a string literal (6.4.5).
34	The characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the `<` and `>` delimiters, or the characters `'`, `\`, `//`, or `/*` occur in the sequence between the `"` delimiters, in a header name preprocessing token (6.4.7).
35	A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5).
36	An exceptional condition occurs during the evaluation of an expression (6.5).
37	An object has its stored value accessed other than by an lvalue of an allowable type (6.5).
38	For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).
39	For 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).
40	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).
41	A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).
42	A member of an atomic structure or union is accessed (6.5.2.3).
43	The operand of the unary `*` operator has an invalid value (6.5.3.2).
44	A pointer is converted to other than an integer or pointer type (6.5.4).
45	The value of the second operand of the `/` or `%` operator is zero (6.5.5).
46	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).
47	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary `*` operator that is evaluated (6.5.6).
48	Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).
49	An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`) (6.5.6).
50	The result of subtracting two pointers is not representable in an object of type `ptrdiff_t` (6.5.6).

51	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).
52	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).
53	Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).
54	An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).
55	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, _Alignof expressions, or immediately-cast floating constants; or contains casts (outside operands to sizeof and _Alignof operators) other than conversions of arithmetic types to integer types (6.6).
56	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 a complete object type plus or minus an integer constant expression (6.6).
57	An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, or _Alignof expressions; or contains casts (outside operands to sizeof or _Alignof operators) other than conversions of arithmetic types to arithmetic types (6.6).
58	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).
59	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).
60	A function is declared at block scope with an explicit storage-class specifier other than extern (6.7.1).
61	A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1).
62	An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).
63	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).
64	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).
65	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).
66	The specification of a function type includes any type qualifiers (6.7.3).
67	Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).
68	An object which has been modified is accessed through a restrict-qualified pointer to a const-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).
69	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).
70	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).
71	A function declared with a _Noreturn function specifier returns to its caller (6.7.4).
72	The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5).
73	Declarations of an object in different translation units have different alignment specifiers (6.7.5).

74	Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.6.1).
75	The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.6.2).
76	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 (6.7.6.2).
77	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 (6.7.6.3).
78	A storage-class specifier or type qualifier modifies the keyword `void` as a function parameter type list (6.7.6.3).
79	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) (6.7.6.3).
80	The value of an unnamed member of a structure or union is used (6.7.9).
81	The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.9).
82	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 (6.7.9).
83	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 (6.7.9).
84	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).
85	A function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list (6.9.1).
86	An adjusted parameter type in a function definition is not a complete object type (6.9.1).
87	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).
88	The `}` that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).
89	An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).
90	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).
91	The `#include` preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).
92	The character sequence in an `#include` preprocessing directive does not start with a letter (6.10.2).
93	There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.3).
94	The result of the preprocessing operator # is not a valid character string literal (6.10.3.2).
95	The result of the preprocessing operator ## is not a valid preprocessing token (6.10.3.3).
96	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).
97	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).
98	A `#pragma STDC` preprocessing directive does not match one of the well-defined forms (6.10.6).
99	The name of a predefined macro, or the identifier `defined`, is the subject of a `#define` or `#undef`

	preprocessing directive (6.10.8).
100	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).
101	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).
102	A header is included within an external declaration or definition (7.1.2).
103	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).
104	A standard header is included while a macro is defined with the same name as a keyword (7.1.2).
105	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).
106	The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).
107	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).
108	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).
109	The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).
110	The macro definition of `assert` is suppressed in order to access an actual function (7.2).
111	The argument to the `assert` macro does not have a scalar type (7.2).
112	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).
113	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).
114	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).
115	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).
116	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).
117	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).
118	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).
119	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, 7.22.6.1, 7.22.6.2, 7.22.1).
120	The program modifies the string pointed to by the value returned by the `setlocale` function (7.11.1.1).
121	The program modifies the structure pointed to by the value returned by the `localeconv` function (7.11.2.1).
122	A macro definition of `math_errhandling` is suppressed or the program defines an identifier with the name `math_errhandling` (7.12).
123	An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.14).
124	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).
125	An invocation of the `setjmp` macro occurs other than in an allowed context (7.13.2.1).

126	The `longjmp` function is invoked to restore a nonexistent environment (7.13.2.1).
127	After a `longjmp`, there is an attempt to access the value of an object of automatic storage duration that does not have 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).
128	The program specifies an invalid pointer to a signal handler function (7.14.1.1).
129	A signal handler returns when the signal corresponded to a computational exception (7.14.1.1).
130	A signal handler called in response to `SIGFPE`, `SIGILL`, `SIGSEGV`, or any other implementation-defined value corresponding to a computational exception returns (7.14.1.1).
131	A signal occurs as the result of calling the `abort` or `raise` function, and the signal handler calls the `raise` function (7.14.1.1).
132	A signal occurs other than as the result of calling the `abort` or `raise` function, and the signal handler refers to an object with static or thread storage duration that is not a lock-free atomic object 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, the `quick_exit` function, or the `signal` function (for the same signal number) (7.14.1.1).
133	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).
134	A signal is generated by an asynchronous signal handler (7.14.1.1).
135	The `signal` function is used in a multi-threaded program (7.14.1.1).
136	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 (7.16, 7.16.1.1, 7.16.1.4).
137	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 (7.16).
138	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` (7.16.1).
139	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 (7.16.1, 7.16.1.2, 7.16.1.3, 7.16.1.4).
140	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 `*` (7.16.1.1).
141	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 (7.16.1.1).
142	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` (7.16.1.2, 7.16.1.4).
143	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 (7.16.1.4).
144	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 (7.19).
145	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 (7.20.4).
146	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 (7.21.2).
147	Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.21.2).
148	The value of a pointer to a `FILE` object is used after the associated file is closed (7.21.3).

149	The stream for the `fflush` function points to an input stream or to an update stream in which the most recent operation was input (7.21.5.2).
150	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 (7.21.5.3).
151	An output operation on an update stream is followed by an input operation without an intervening call to the `fflush` function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.21.5.3).
152	An attempt is made to use the contents of the array that was supplied in a call to the `setvbuf` function (7.21.5.6).
153	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 (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).
154	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 (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).
155	In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).
156	A conversion specification for a formatted output function uses an asterisk to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.21.6.1, 7.29.2.1).
157	A conversion specification for a formatted output function uses a `#` or `0` flag with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).
158	A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).
159	An `s` conversion specifier is encountered by one of the formatted output functions, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.21.6.1, 7.29.2.1).
160	An `n` conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).
161	A `%` conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly `%%` (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).
162	An invalid conversion specification is found in the format for one of the formatted input/output functions, or the `strftime` or `wcsftime` function (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).
163	The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than `INT_MAX` (7.21.6.1, 7.29.2.1).
164	The number of input items assigned by a formatted input function is greater than `INT_MAX` (7.21.6.2, 7.29.2.2).
165	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 (7.21.6.2, 7.29.2.2).
166	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 `[`) (7.21.6.2, 7.29.2.2).
167	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 (7.21.6.2, 7.29.2.2).
168	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 (7.21.6.2, 7.29.2.2).
169	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 (7.21.6.8, 7.21.6.9, 7.21.6.10, 7.21.6.11, 7.21.6.12, 7.21.6.13, 7.21.6.14, 7.29.2.5, 7.29.2.6, 7.29.2.7, 7.29.2.8, 7.29.2.9, 7.29.2.10).

170	The contents of the array supplied in a call to the `fgets` or `fgetws` function are used after a read error occurred (7.21.7.2, 7.29.3.2).
171	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 (7.21.7.10).
172	The file position indicator for a stream is used after an error occurred during a call to the `fread` or `fwrite` function (7.21.8.1, 7.21.8.2).
173	A partial element read by a call to the `fread` function is used (7.21.8.1).
174	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` (7.21.9.2).
175	The `fsetpos` function is called to set a position that was not returned by a previous successful call to the `fgetpos` function on a stream associated with the same file (7.21.9.3).
176	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 (7.22.3).
177	The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is used (7.22.3).
178	The alignment requested of the `aligned_alloc` function is not valid or not supported by the implementation, or the size requested is not an integral multiple of the alignment (7.22.3.1).
179	The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to `free` or `realloc` (7.22.3.3, 7.22.3.5).
180	The value of the object allocated by the `malloc` function is used (7.22.3.4).
181	The value of any bytes in a new object allocated by the `realloc` function beyond the size of the old object are used (7.22.3.5).
182	The program calls the `exit` or `quick_exit` function more than once, or calls both functions (7.22.4.4, 7.22.4.7).
183	During the call to a function registered with the `atexit` or `at_quick_exit` function, a call is made to the `longjmp` function that would terminate the call to the registered function (7.22.4.4, 7.22.4.7).
184	The string set up by the `getenv` or `strerror` function is modified by the program (7.22.4.6, 7.24.6.2).
185	A signal is raised while the `quick_exit` function is executing (7.22.4.7).
186	A command is executed through the `system` function in a way that is documented as causing termination or some other form of undefined behavior (7.22.4.8).
187	A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.22.5).
188	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 (7.22.5).
189	The array being searched by the `bsearch` function does not have its elements in proper order (7.22.5.1).
190	The current conversion state is used by a multibyte/wide character conversion function after changing the `LC_CTYPE` category (7.22.7).
191	A string or wide string utility function is instructed to access an array beyond the end of an object (7.24.1, 7.29.4).
192	A string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.24.1, 7.29.4).
193	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 (7.24.4.5, 7.27.3.5, 7.29.4.4.4, 7.29.5.1).
194	The first argument in the very first call to the `strtok` or `wcstok` is a null pointer (7.24.5.8, 7.29.4.5.7).

195	The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.25).
196	A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.25).
197	At least one member of the broken-down time passed to `asctime` contains a value outside its normal range, or the calculated year exceeds four digits or is less than the year 1000 (7.27.3.1).
198	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 (7.29.2.11).
199	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 (7.29.4.5.7).
200	An `mbstate_t` object is used inappropriately (7.29.6).
201	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` (7.30.1).
202	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 (7.30.2.2.1).
203	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 (7.30.3.2.1).

Bibliography

[Banahan 2003] Banahan, Mike, Brady, Declan, & Doran, Mark. *The C Book, Featuring the ANSI C Standard*. Boston: 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*. Upper Saddle River, NJ: 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 1995] CERT Advisory CA-1995-14 Telnetd Environment Vulnerability. http://www.cert.org/advisories/CA-1995-14.html (rev. October 30, 1997).

[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*. Boston: 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: 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, 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.

[Kettlewell 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, 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. http://msdn.microsoft.com/en-us/library/fw5abdx6%28v=vs.80%29.aspx (2007).

[MISRA 2004] Motor Industry Software Reliability 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: 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*. Boston: Addison-Wesley, 2006.

[van Sprundel 2006] van Sprundel, Ilja. Unusualbugs, 2006.

[Summit 1995] Summit, Steve. *C Programming FAQs: Frequently Asked Questions*. Boston: 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: 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.