C Secure Coding Guidelines

Review Draft

WG14/N1393

[cover page content]

Document type: Technical Report Type 2**Error! Reference source not found.**

Document subtype: n/a**Error! Reference source not found.**

Document stage: (3) Proposed Draft Technical Report**Error! Reference source not found.**

Document language: E**Error! Reference source not found.Error! Reference source not found.**

Contents

Page

C Secure Coding Guidelines 1

Review Draft 1

1 Scope 1

2 Conformance 1
2.1 Completeness and Soundness 2

3 Normative references 2

4 Terms and definitions 3
4.1 analyzer 3
4.2 asynchronous-safe 3
4.3 exploit 3
4.4 invalid pointer 3
4.5 security flaw 3
4.6 security policy 3
4.7 valid pointer 3
4.8 vulnerability 4

5 Symbols (and abbreviated terms) 4

6 Known Security Problems 4
6.1 Do not call system if you do not need a command processor [ENV004] 4
6.2 Do not use user input in format strings [FIO030] 5
6.3 Detect and handle input and output errors [FIO004] 5
6.4 Canonicalize file names originating from untrusted sources [FIO002] 7
6.5 Only use valid format strings [FIO000] 8
6.6 Allocate sufficient memory for an object [MEM035] 9
6.7 Do not access freed memory [MEM030] 10
6.8 Overwrite sensitive information from the heap before returning it [MEM003] 11
6.9 Do not use deprecated or obsolescent functions [MSC034] 12
6.10 Do not truncate a null-terminated byte string [STR003] 15
6.11 Do not use pointer arithmetic that results in an invalid pointer [ARR038] 16
6.12 Do not loop beyond the end of an array [ARR035] 17
6.13 Do not copy data into an object of insufficient storage size [ARR033] 17
6.14 Use array indices that are within the range of the array bounds [ARR030] 18
6.15 Do not apply the sizeof operator to a pointer when taking the size of an array [ARR001] 18
6.16 Evaluate integer expressions in a larger size before using with that size [INT035] 19
6.17 Do not let operations on signed integers result in overflow [INT032] 20
6.18 Do not let integer conversions result in lost data or sign [INT031] 21
6.19 Verify that all integer values are in range [INT008] 22
6.20 Enforce limits on integer values originating from untrusted sources [INT004] 23

7 Direct Result 24
7.1 Do not create a universal character name through concatenation [PRE030] 24
7.2 Do not shift a negative number of bits or more bits than exist in the operand [INT034] 24
7.3 Use the correct syntax for flexible array members [MEM033] 25
7.4 Do not call remove on an open file [FIO008] 26
7.5 Do not use non-standard mode parameters when calling fopen [FIO011] 26
7.6 Do not push back anything other than one read character [FIO013] 27
7.7 Do not simultaneously open the same file multiple times [FIO031] 28
7.8 Use feof and ferror to detect file conditions when sizeof(int) == sizeof(char) [FIO035] 28
7.9 Do not use a copy of a FILE object for input and output [FIO038] 29
7.10 Do not input and output from a stream without a flush or positioning call [FIO039] 29
7.11 Only use values for fsetpos that are returned from fgetpos [FIO044] 30
7.12 Use consistent array notation across all source files [ARR031] 31
7.13 Do not manipulate time_t typed values directly [MSC005] 32

7.14 Do not invoke an unsafe macro with arguments containing side effects [PRE031] 32
7.15 Do not refer to an object outside its lifetime [DCL030] 33
7.16 Do not use restrict-qualified pointers that overlap as parameters [DCL033] 33
7.17 Do not declare an identifier with conflicting linkage classifications [DCL036] 34
7.18 Do not cast away const qualification [EXP005] 35
7.19 Do not apply operators expecting one type to data of an incompatible type [EXP011] 35
7.20 Do not depend on order of evaluation between sequence points [EXP030] 36
7.21 Do not cast away a volatile qualification [EXP032] 37
7.22 Do not reference uninitialized memory [EXP033] 37
7.23 Do not call functions with incorrect arguments [EXP037] 39
7.24 Do not dereference a null pointer [EXP034] 39
7.25 Do not access an array in the result of a function call after a sequence point [EXP035] 40
7.26 Do not convert pointers into more strictly aligned pointer types [EXP036] 40
7.27 Do not call offsetof on bit-field members or invalid types [EXP038] 41
7.28 Do not access a variable through a pointer of an incompatible type [EXP039] 42
7.29 Do not convert a pointer to integer or integer to pointer [INT011] 43
7.30 Do not let division and modulo operations result in divide-by-zero errors [INT033] 43
7.31 Do not call functions expecting real values with complex values [FLP031] 44
7.32 Do not compare floating point values [FLP035] 45
7.33 Do not modify string literals [STR030] 45
7.34 Pass only unsigned char to character handling functions [STR037] 46
7.35 Do not perform zero length allocations [MEM004] 46
7.36 Only pass arguments to calloc that, when multiplied, fit in size_t [MEM007] 47
7.37 Use realloc only to resize dynamically allocated arrays [MEM008] 48
7.38 Only free memory allocated dynamically [MEM034] 49
7.39 Do not perform operations on devices that are only appropriate for files [FIO032] 49
7.40 Only register atexit handlers that return normally [ENV032] 50
7.41 Do not call non-asynchronous-safe functions from signal handlers [SIG030] 51
7.42 Do not call raise from a signal handler [SIG033] 52
7.43 Do not access or modify shared objects in signal handlers [SIG031] 53
7.44 Do not redefine errno [ERR031] 54
7.45 Invoke functions using the correct type [DCL035] 54
7.46 Do not use incompatible array types in an expression [ARR034] 55
7.47 Do not subtract or compare two pointers that do not refer to the same array [ARR036] 55
7.48 Do not modify the string returned by getenv [ENV030] 56
7.49 Free dynamically allocated memory exactly once [MEM031] 57

8 Indirect Result 58
8.1 Do not let unsigned integer operations wrap [INT030] 58
8.2 Do not use a plain int bit-field [INT012] 59
8.3 Do not hard code the size of a type [EXP009] 59
8.4 Do not shift signed types [INT013] 60
8.5 Use intmax_t or uintmax_t for formatted IO on defined integer types [INT015] 60
8.6 Do not use floating point variables as loop counters [FLP030] 61
8.7 Do not reuse variable names in subscopes [DCL001] 61
8.8 Include the appropriate type information in function declarators [DCL007] 62
8.9 Operands to the sizeof operator should not contain side effects [EXP006] 63
8.10 Use rsize_t or size_t for all integer values representing the size of an object [INT001] 63
8.11 Use only explicitly signed or unsigned char type for numeric values [INT007] 65
8.12 Do not use enumeration constants that map to nonunique values [INT009] 65
8.13 Do not place a semicolon on the same line as an if, for, or while statement [EXP017] 65
8.14 Do not compare function pointers to constant values [EXP018] 66
8.15 Do not perform bitwise operations in conditional expressions [EXP016] 66
8.16 Do not perform assignments in conditional expressions [EXP015] 67
8.17 Check all possible data paths in an if-chain or switch statement [MSC001] 67
8.18 Do not use trigraphs [PRE007] 68
8.19 Declare identifiers before using them [DCL031] 69
8.20 Use only unique mutually visible identifiers [DCL032] 70
8.21 Do not perform byte-by-byte comparisons between structures [EXP004] 70
8.22 Avoid domain and range errors in math functions [FLP032] 71

8.23 Convert integers to floating point for floating point operations [FLP033] 73
8.24 Do not declare a variable length array with an untrusted value [ARR032] 73
8.25 Cast characters to unsigned types before converting to larger integer sizes [STR034] 74
8.26 Cast the result of memory allocation into a pointer to the allocated type [MEM002] 75
8.27 Use a variable of type int to capture the return value of character IO functions [FIO034] 75
8.28 Do not call getc or putc with stream arguments that have side effects [FIO040] 76
8.29 Do not store the pointer to the string returned by getenv [ENV000] 77
8.30 Do not call signal from interruptible signal handlers [SIG034] 77
8.31 Do not call longjmp from inside a signal handler [SIG032] 78
8.32 Do not use abort or assert when atexit handlers are registered [ERR006] 79
8.33 Set and check errno correctly [ERR030] 80
8.34 Finish case labels with a break statement [MSC017] 83
8.35 Do not assume a positive remainder when using the % operator [INT010] 83
8.36 Ensure that floating point conversions are within range of the new type [FLP034] 84
8.37 Null-terminate strings that are not null-terminated [STR032] 85
8.38 Do not assume character data does not contain a null byte [FIO037] 86
8.39 Close files when they are no longer needed [FIO042] 86
8.40 Sanitize the environment when invoking external programs [ENV003] 87
8.41 Do not take the size of a pointer to determine the size of the pointed-to type [EXP001] 88
8.42 Do not add or subtract a scaled integer to a pointer [EXP008] 88
8.43 Do not ignore values returned by functions [EXP012] 89
8.44 Detect and handle memory allocation errors [MEM032] 90

9 Good Practices 91
9.1 Avoid performing bitwise and arithmetic operations on the same data [INT014] 91
9.2 Use typedefs to define types [PRE003] 91
9.3 Use parentheses within macros around replacement lists [PRE002] 92
9.4 Use parentheses within macros around parameter names [PRE001] 92
9.5 Do not conclude a single-statement macro definition with a semicolon [PRE011] 93
9.6 Wrap multi-statement macros in a do-while loop [PRE010] 94
9.7 Use visually distinct identifiers [DCL002] 94
9.8 Use a static assertion to test the value of a constant expression [DCL003] 95
9.9 Use typedefs to improve code readability [DCL005] 96
9.10 Do not declare more than one variable per declaration [DCL004] 96
9.11 Use 'L', not 'l', to indicate a long value [DCL016] 97
9.12 Use parentheses for precedence of operation [EXP000] 97
9.13 Do not use relational operators on boolean values [EXP013] 97
9.14 Do not include unused values [MSC013] 98
9.15 Do not use code that has no effect [MSC012] 98
9.16 Do not use dead code [MSC007] 99
9.17 Use comments consistently and in a readable fashion [MSC004] 100
9.18 Use unique header file names [PRE008] 101
9.19 Enclose header files in an inclusion guard [PRE006] 101
9.20 Use plain char for characters in the basic character set [STR004] 102
9.21 Reset strings after fgets failure [FIO040] 102
9.22 Do not store or use naked function pointers [MSC016] 103
9.23 Const-qualify immutable variables [DCL000] 103
9.24 Declare functions that return an errno error code with a return type of errno_t [DCL009] 104
9.25 Declare file-scope objects or functions without external linkage as static [DCL015] 104
9.26 Declare parameter pointers whose pointed-to values do not change as const [DCL013] 105
9.27 Explicitly specify array bounds [ARR002] 105
9.28 Store a new value in pointers immediately after free [MEM001] 106

Annex A (normative) Tool-generated and Tool-maintained Code 108
A.1 General 108
A.2 Clause 108

Foreword

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

Technical Reports are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft Technical Reports adopted by the joint technical committee are circulated to national bodies for voting. Publication as a Technical Report requires approval by at least 75% of the member bodies casting a vote.

The main task of technical committees is to prepare International Standards, but in exceptional circumstances a technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when a technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

ISO/IEC TR NNNNN, which is a Technical Report of type 2, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Introduction

An essential element of secure coding in the C programming language is a set of well-documented and enforceable coding guidelines. The guidelines specified in this Technical Report applies to *analyzers*, including static analysis tools and C language compilers vendors that wish to diagnose insecure code beyond the requirements of the language standard.

This Technical Report is divided into two major subdivisions:

— preliminary elements (clauses 1–5);

— secure coding guidelines (clauses 6-18);

The guidelines documented in this Technical Report 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 guidelines, as specified, are reasonably simple, although complications can exist in indentifying exceptions. Additionally, there are significant differences in guidelines that are intended primarily for evaluating new code versus legacy code. Because security is a primary concern, these guidelines are intended first and foremost for evaluating new code and legacy code secondarily. Consequently, appilcation 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 to eliminate the need to research each diagnostic on every invocation of the tool. The implementation of such a mechanism is encouraged, but not required by this standard.

1 Scope

This Technical Report specifies

— guidelines for secure coding in the C programming language;

— non-compliant code examples.

This Technical Report does not specify

— the mechanism by which these guidelines are enforced;

— any particular coding style to be enforced [1];

2 Conformance

In this Technical Report, ‘‘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 guidelines specified in this Technical Report. The guideilnes may be extended in an implementation-dependent manner.

Secure coding guidance varies depending on how code is generated and maintained. Categories of code include the following:

— Tool-generated, tool-maintained code that is specified and maintained in a higher level format, from which language-specific source code is generated. The source code is generated from this higher level description and then provided as input to the language compiler. The generated source code is never viewed or modified by the programmer.

— Tool-generated, hand-maintained code that is specified and maintained in a higher level format, from which language-specific source code is generated. It is expected or anticipated, however, that at some point in the development cycle the tool will cease to be used and the generated source code will be visually inspected and/or manually modified and maintained.

— Hand-developed and maintained code is manually written by a programmer using a text editor or interactive development environment; the programmer maintains source code directly in the source code format provided to the compiler.

Source code that is written and maintained by hand must be concerned with readability and program comprehension.

Certain guidelines (listed in Annex A) are not applicable for source code that is never directly handled by a programmer, although requirements for correct behavior still apply. Reading and comprehension requirements apply to code that is tool generated and hand maintained but does not apply to code that is tool generated and tool maintained.

[1] Coding style issues as it has proven impossible to develop a consensus on appropriate style guidelines. Programmers should defines 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 (IDEs) provide such capabilities.

2.1 Completeness and Soundness

To the greatest extent possible, an analyzer should be both complete and sound with respect to enforceable guidelines.

An analyzer is considered complete (with respect to a specific guideline) if it does not give a false-negative result, meaning it is able to find all violations of a guideline.

An analyzer is considered sound if it does not issue false-positive results, or false alarms. Too many false alarms can lead users to ignore the results of a tool, potentially missing serious issues.

The possibilities for a given guideline are outlined in Figure 1.

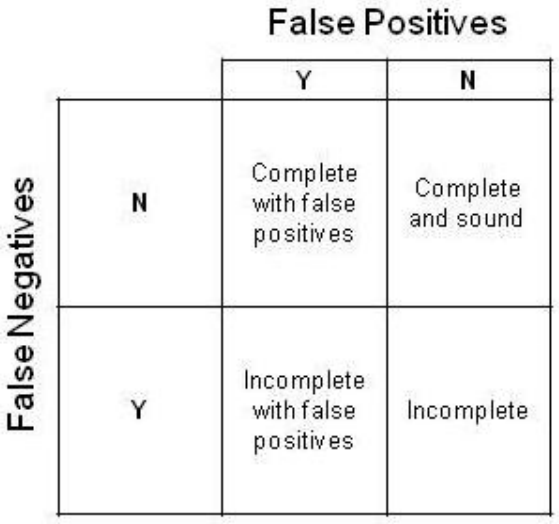

Figure 1. Completeness and soundness.

Analyzers are trusted processes, meaning that reliance is placed on the output of the tools. Consequently, developers must ensure that this trust is not misplaced. Ideally, this should be achieved by the tool supplier running appropriate validation tests. While it is possible to use a validation suite to test an analyzer, no formal validation scheme exists at this time.

The degree to which tools minimize false positive diagnostics is a quality of implementation issue.

3 Normative references

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

ISO/IEC 9899:1999, *Programming Languages — C*.

ISO/IEC 9899:1999/Cor 1:2001, *Programming Languages — C — Technical Corrigendum 1*.

ISO/IEC 9899:1999/Cor 2:2004, *Programming Languages — C — Technical Corrigendum 2*.

ISO/IEC 9899:1999/Cor 3:2007, *Programming Languages — C — Technical Corrigendum 3*.

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

ISO/IEC PDTR 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, *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 (all parts), Information technology — *Universal Multiple-Octet Coded Character Set (UCS)*.

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems (previously designated IEC 559:1989)*.

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

4 Terms and definitions

4.1 **analyzer**

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

4.2 **asynchronous-safe**

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

4.3 **exploit**

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

4.4 **invalid pointer**

A pointer that is not a **valid pointer**.

4.5 **security flaw**

A software defect that poses a potential security risk.

4.6 **security policy**

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

4.7 **valid pointer**

A pointer that refers to an element within an array or one past the last element of an array. For the purposes of this definition, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type. (Cf 6.5.8p3)

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. (Cf 6.3.2.3p7)

4.8 vulnerability

A set of conditions that allows an attacker to violate an explicit or implicit security policy.

5 Symbols (and abbreviated terms)

6 Known Security Problems

6.1 Do not call system if you do not need a command processor [ENV004]

Using the `system` function when a command processor is not needed shall be diagnosed because doing so complicates command-string sanitization.

Noncompliant Code Example

In this noncompliant code example, `system` is used to execute `any_cmd`.

```
char *input = NULL;

/* input gets initialized by user */

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) {
  perror("Error executing input");
}
```

Noncompliant Code Example

In this noncompliant code example, `system` is used to remove the `.config` file in the user's home directory. If this program has superuser privileges on a POSIX system, an attacker can manipulate the value of `HOME` so that this program can remove any file named `.config` anywhere on the system.

```
system("rm ~/.config");
```

Bibliography

[ISO/IEC 9899:1999] Sections 7.20.4.6, "The `system` function"

[ISO/IEC PDTR 24772] "XZQ Unquoted Search Path or Element"

[MITRE 07] CWE ID 88, "Argument Injection or Modification," and CWE ID 78, "Failure to Sanitize Data into an OS Command (aka 'OS Command Injection')"

[Open Group 04] `environ`, `execl`, `execv`, `execle`, `execve`, `execlp`, `execvp` - execute a file, `popen`, `unlink`, XCU Section 2.8.2, "Exit Status for Commands"

[Seacord 09] "ENV04-C. Do not call system if you do not need a command processor"

[Wheeler 04]

6.2 Do not use user input in format strings [FIO030]

Using a format string containing user input as an argument to a formatted IO function shall be diagnosed because this can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the string `msg` contains user input and is used as the format string in the call to `fprintf`.

```
#define MSG_FORMAT "%s cannot be authenticated.\n"

void incorrect_password(const char *user) {
  /* user names are restricted to 256 characters or less */
  static const char *msg_format = MSG_FORMAT;
  size_t len = strlen(user) + sizeof(MSG_FORMAT);
  char *msg = (char *)malloc(len);
  if (!msg) {
    /* handle error condition */
  }
  int ret = snprintf(msg, len, msg_format, user);
  if (ret < 0 || ret >= len) {\
    /* Handle error */
  }
  fprintf(stderr, msg);
  free(msg);
  msg = NULL;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.6, "Formatted input/output functions"

[ISO/IEC PDTR 24772] "RST Injection"

[MITRE 07] CWE ID 134, "Uncontrolled Format String"

[Open Group 04] syslog

[Seacord 05] Chapter 6, "Formatted Output"

[Seacord 09] "FIO30-C. Exclude user input from format strings"

[Viega 05] Section 5.2.23, "Format string problem"

6.3 Detect and handle input and output errors [FIO004]

Not detecting and handling input and output errors shall be diagnosed because this can result in undefined or unexpected behavior.

The return values of the standard file IO functions in the following table shall be checked for errors.

Function	Successful Return	Error Return

fclose	zero	EOF
fflush	zero	EOF
fgetc	character read	use ferror and feof
fgetpos	zero	nonzero
fgets	string	NULL
fprintf	number of characters	negative
fputc	character written	use ferror
fputs	nonnegative	EOF
fread	elements read	elements read
freopen	pointer to stream	NULL
fscanf	number of conversions	EOF
fseek	zero	nonzero
fsetpos	zero	nonzero
ftell	file position	-1L
fwrite	elements written	elements written
getc	character read	use ferror and feof
getchar	character read	use ferror and feof
printf	number of characters	negative
putc	character written	use ferror
putchar	character written	use ferror
puts	nonnegative	EOF
remove	zero	nonzero
rename	zero	nonzero
setbuf	zero	nonzero

scanf	number of conversions	EOF
snprintf	number of characters that would be written	negative
sscanf	number of conversions	EOF
tmpfile	pointer to stream	NULL
tmpnam	non-NULL pointer	NULL
ungetc	character pushed back	EOF
vfprintf	number of characters (non-negative)	negative
vfscanf	number of conversions (non-negative)	EOF
vprintf	number of characters (non-negative)	negative
vscanf	number of conversions (non-negative)	EOF

Noncompliant Code Example

In this noncompliant code example, the return value of `fseek` is not checked for an error condition.

```
FILE *file;
long offset;

/* initialize file and offset */

fseek(file, offset, SEEK_SET);
/* Process file */
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.3, "Files," Section 7.19.4, "Operations on Files," and "File Positioning Functions"

[Kettlewell 02] Section 6, "I/O Error Checking"

[MITRE 07] CWE ID 391, "Unchecked Error Condition"

[Seacord 05a] Chapter 7, "File I/O"

[Seacord 09] "FIO04-C. Detect and handle input and output errors"

6.4 Canonicalize file names originating from untrusted sources [FIO002]

Using an untrusted file name that has not been canonicalized shall be diagnosed because it is difficult to validate non-canonical path names. A canonical file name is one that compares equal to any other canonical file name for the same file.

Noncompliant Code Example

In this noncompliant code example, the file name referred to by `argv[1]` is not canonicalized. If `verify_file` does not correctly handle special characters or symbolic links, for example, then the call to `fopen` may result in an unintended file being accessed.

```
/* Verify argv[1] is supplied */

if (!verify_file(argv[1]) {
  /* Handle error */
}

if (fopen(argv[1], "w") == NULL) {
  /* Handle error */
}

/* ... */
```

Bibliography

[Austin Group 08] `realpath`

[Drepper 06] Section 2.1.2, "Implicit Memory Allocation"

[Howard 02] Chapter 11, "Canonical Representation Issues"

[ISO/IEC 9899:1999] Section 7.19.3, "Files"

[ISO/IEC PDTR 24772] "EWR Path Traversal"

[Linux 07] `realpath`(3), `pathconf`(3)

[MITRE 07] CWE ID 73, "External Control of File Name or Path," CWE ID 41, "Failure to Resolve Path Equivalence," CWE ID 59, "Failure to Resolve Links Before File Access (aka 'Link Following')," and CWE ID 22, "Path Traversal"

[MSDN] "`GetFullPathName` Function"

[Open Group 04] Section 4.11, "Pathname Resolution", and `realpath`

[Seacord 05a] Chapter 7, "File I/O"

[Seacord 09] "FIO02-C. Canonicalize path names originating from untrusted sources"

[VU#743092]

6.5 Only use valid format strings [FIO000]

Using a format string whose conversion specifiers do not match the number and type of arguments to a formatted IO function shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the conversion specifiers in the format string supplied to `printf` do not match the arguments.

```
const char *error_msg = "Resource not available to user.";
int error_type = 3;
```

```
/* ... */
printf("Error (type %s): %d\n", error_type, error_msg);
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.6.1, "The `fprintf` function"

[MITRE 07] CWE ID 686, "Function Call With Incorrect Argument Type"

[Seacord 09] "FIO00-C. Take care when creating format strings"

6.6 Allocate sufficient memory for an object [MEM035]

Allocating insufficient memory for an object shall be diagnosed because this can result in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the multiplication of `num_blocks` and `BLOCKSIZE` could result in an integer overflow before being stored in `alloc`. If this happens, `malloc` will return insufficient memory for the intended object.

```
enum { BLOCKSIZE = 16 };
/* ... */
void *alloc_blocks(size_t num_blocks) {
  if (num_blocks == 0) {
    return NULL;
  }
  unsigned long long alloc = num_blocks * BLOCKSIZE ;
  return (alloc < UINT_MAX)
     ? malloc(num_blocks * BLOCKSIZE )
     : NULL;
}
```

Noncompliant Code Example

In this noncompliant code example, if `len` is a negative number then `len` will be converted to a `size_t` in the call to `memcpy`, likely resulting in a large positive number of bytes to be copied to `buf`.

```
int len;
char *str;
char buf[BUFF_SIZE];

/* ... */
if (len < BUFF_SIZE){
  memcpy(buf, str, len);
}
/* ... */
```

Noncompliant Code Example

In this noncompliant code example, if `sizeof(int) != sizeof(long)` then the size of the allocation will be calculated incorrectly.

```
void function(size_t len) {
   long *p;
   if (len == 0 || len > SIZE_MAX / sizeof(long)) {
      /* Handle overflow */
   }
```



```
   p = (long *)malloc(len * sizeof(int));
   if (p == NULL) {
      /* Handle error */
   }
   /* ... */
   free(p);
}
```

Bibliography

[Coverity 07]

[ISO/IEC 9899:1999] Section 7.20.3, "Memory Management Functions"

[ISO/IEC PDTR 24772] "XYB Buffer Overflow in Heap"

[MITRE 07] CWE ID 190, "Integer Overflow (Wrap or Wraparound)," and CWE ID 131, "Incorrect Calculation of Buffer Size"

[Seacord 05] Chapter 4, "Dynamic Memory Management," and Chapter 5, "Integer Security"

[Seacord 09] "MEM35-C. Allocate sufficient memory for an object"

6.7 Do not access freed memory [MEM030]

Accessing freed memory shall be diagnosed because doing so results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, `p->next` is accessed after `p` has been freed.

```
for (p = head; p != NULL; p = p->next)
  free(p);
```

Noncompliant Code Example

In this noncompliant code example, `buff` is written to after it has been freed.

```
int main(int argc, const char *argv[]) {
  char *buff;

  buff = (char *)malloc(BUFSIZE);
  if (!buff) {
     /* Handle error condition */
  }
  /* ... */
  free(buff);
  /* ... */
  strncpy(buff, argv[1], BUFSIZE-1);
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.20.3.2, "The `free` function"

[ISO/IEC PDTR 24772] "DCM Dangling references to stack frames" and "XYK Dangling Reference to Heap"

[Kernighan 88] Section 7.8.5, "Storage Management"

[MISRA 04] Rule 17.6

[MITRE 07] CWE ID 416, "Use After Free"

[OWASP Freed Memory]

[Seacord 05a] Chapter 4, "Dynamic Memory Management"

[Seacord 09] "MEM30-C. Do not access freed memory"

[Viega 05] Section 5.2.19, "Using freed memory"

6.8 Overwrite sensitive information from the heap before returning it [MEM003]

Returning sensitive memory to the heap without first overwriting the memory shall be diagnosed because the sensitive data may be viewable by other programs.

Consequently, passing a pointer that references sensitive data to `free` without first overwriting the memory shall be diagnosed.

Additionally, passing a pointer that references sensitive data to `realloc` can result in an implicit call to `free`, leaving a copy of the data on the heap. Because there is no way to tell if this has happened, there is no safe way to use `realloc` with sensitive data. Therefore, using `realloc` to reallocate space for sensitive data shall be diagnosed.

Noncompliant Code Example

In this noncompliant code example, the sensitive memory referenced by `new_secret` is deallocated using `free` without having been overwritten first.

```
char *secret;

/* initialize secret */

char *new_secret;
size_t size = strlen(secret);
if (size == SIZE_MAX) {
  /* Handle error */
}

new_secret = (char *)malloc(size+1);
if (!new_secret) {
  /* Handle error */
}
strcpy(new_secret, secret);

/* Process new_secret... */

free(new_secret);
new_secret = NULL;
```

Noncompliant Code Example

In this noncompliant code example, the sensitive memory referenced by secret is reallocated using `realloc`.

```
char *secret;

/* initialize secret */
```

```
size_t secret_size = strlen(secret);
/* ... */
if (secret_size > SIZE_MAX/2) {
   /* handle error condition */
}
else {
  secret = (char *)realloc(secret, secret_size * 2);
}
```

Bibliography

[Black 07]

[Fortify 06]

[Graff 03]

[ISO/IEC 9899:1999] Section 7.20.3, "Memory management functions"

[ISO/IEC PDTR 24772] "XZK Sensitive Information Uncleared Before Use"

[MITRE 07] CWE ID 226, "Sensitive Information Uncleared Before Release," CWE ID 244, and "Failure to Clear Heap Memory Before Release"

[Seacord 09] "MEM03-C. Clear sensitive information stored in reusable resources returned for reuse"

6.9 Do not use deprecated or obsolescent functions [MSC034]

Using deprecated or obsolsecent functions shall be diagnosed because there exist equivalent functions that are more secure.

Deprecated functions are defined by the C99 standard and Technical Corrigenda. Obsolescent functions are defined by this guideline.

Deprecated Functions

The `gets` function was deprecated by Technical Corrigendum 3.

Obsolescent Functions

The following functions are obsolescent.

asctime	atof	atoi	atol	atoll	bsearch	ctime
fopen	fprintf	freopen	fscanf	fwprintf	fwscanf	getenv
gmtime	localtime	mbsrtowcs	mbstowcs	memcpy	memmove	printf
qsort	remove	rename	rewind	setbuf	snprintf	sprintf
sscanf	strcat	strcpy	strerror	strncat	strncpy	strtok
swprintf	swscanf	tmpfile	tmpfile_s	tmpnam	tmpnam_s	vfprintf
vfscanf	vfwprintf	vfwscanf	vprintf	vscanf	vsnprintf	vsprintf
vsscanf	vswprintf	vswscanf	vwprintf	vwscanf	wcrtomb	wcscat

wcscpy	wcsncat	wcsncpy	wcsrtombs	wcstok	wcstombs	wctomb
wmemcpy	wmemmove	wprintf	wscanf			

Many of the above functions have been obsolesced by functions defined by ISO/IEC TR 24731 (Parts 1 and 2).

The `atof`, `atoi`, `atol`, and `atoll` functions are obsolescent because the `strod`, `strtof`, `strtol`, `strtold`, `strtoll`, `strotul`, and `strtoull` functions can emulate their usage and have more robust error handling capabilities.

The `fopen`, `freopen`, `remove`, and `rename` functions are obsolescent because they use filenames to identify files. Filenames can be manipulated by an attacker to target a file unintended by the program.

The `setbuf` function is obsolescent because `setbuf` does not return a value and can be emulated using `setvbuf`.

The `rewind` function is obsolescent because `rewind` does not return a value and can be emulated using `fseek`.

The `tmpfile` and `tmpfile_s` functions are obsolescent because the filenames of files generated by `tmpfile` may be recycled frequently. Additionally, these functions do not allow the programmer to specify a secure directory in which to create the file.

The `tmpnam` and `tmpnam_s` functions are obsolescent because they do not create a file. A race condition is present between the time `tmpnam` or `tmpnam_s` is called and when the file is created from the generated filename.

Noncompliant Code Example

In this noncompliant code example, `strcat` and `strcpy` are used.

```
void complain(const char *msg) {
  static const char prefix[] = "Error: ";
  static const char suffix[] = "\n";
  char buf[BUFSIZE];

  strcpy(buf, prefix);
  strcat(buf, msg);
  strcat(buf, suffix);
  fputs(buf, stderr);
}
```

Noncompliant Code Example

In this noncompliant code example, `setbuf` is used.

```
FILE *file;
/* Setup file */
setbuf(file, NULL);
/* ... */
```

Noncompliant Code Example

In this noncompliant code example, `tmpnam` is used.

```
char file_name[L_tmpnam];
FILE* fp;

if (!tmpnam(file_name)) {
  /* Handle error */
}

/* A TOCTOU race condition exists here */

fp = fopen(file_name, "wb+");
if (fp == NULL) {
   /* Handle error */
}
```

Noncompliant Code Example

In this noncompliant code example, `tmpfile` is used.

```
FILE* fp = tmpfile();
if (fp == NULL) {
  /* Handle error */
}
```

Noncompliant Code Example

In this noncompliant code example, the file identified by `file_name` is opened, processed, closed, and removed. However, it is possible that the file object identified by `file_name` in the call to `remove` is not the same file object identified by `file_name` in the call to `fopen`.

```
char *file_name;
FILE *f_ptr;

/* initialize file_name */

f_ptr = fopen(file_name, "w");
if (f_ptr == NULL) {
  /* Handle error */
}

/*... Process file ...*/

if (fclose(f_ptr) != 0) {
  /* Handle error */
}

if (remove(file_name) != 0) {
  /* Handle error */
}
```

Bibliography

[Apple Secure Coding Guide] "Avoiding Race Conditions and Insecure File Operations"

[Drepper 06] Section 2.2.1 "Identification When Opening"

[ISO/IEC 9899:1999] Section 7.19.3, "Files," and Section 7.19.4, "Operations on Files"

[ISO/IEC 9899:1999] Section 7.19.5.5, "The `setbuf` function"; 7.19.9.2, "The `fseek` function"; 7.19.9.5 "The `rewind` function"; and 7.21, "String handling `<string.h>`"

[ISO/IEC 9899:1999] Section 7.20.1.4, "The `strtol`, `strtoll`, `strtoul`, and `strtoull` functions," and Section 7.19.6, "Formatted input/output functions"

[ISO/IEC 9899:1999] Section 7.21.5.8, "The `strtok` function"

[ISO/IEC PDTR 24772] "TRJ Use of Libraries"

[ISO/IEC TR 24731-1:2007]

[Klein 02]

[Linux 07] `strtok`(3)

[MITRE 07] CWE ID 73 "External Control of File Name or Path," CWE ID 367, "Time-of-check Time-of-use Race Condition," and CWE ID 676, "Use of Potentially Dangerous Function"

[MITRE 07] CWE ID 192, "Integer Coercion Error"; and CWE ID 197, "Numeric Truncation Error"

[MITRE 07] CWE ID 464, "Addition of Data Structure Sentinel"

[MITRE 07] CWE ID 676, "Use of Potentially Dangerous Function," and CWE ID 20, "Insufficient Input Validation"

[Open Group 04] "The `open` function"

[Seacord 05a] Chapter 2, "Strings"

[Seacord 05a] Chapter 7, "File I/O"

[Seacord 05b]

[Seacord 09] "FIO01-C. Be careful using functions that use file names for identification"

[Seacord 09] "FIO07-C. Prefer `fseek` to `rewind`"

[Seacord 09] "FIO12-C. Prefer `setvbuf` to `setbuf`"

[Seacord 09] "INT05-C. Do not use input functions to convert character data if they cannot handle all possible inputs"

[Seacord 09] "INT06-C. Use `strtol` or a related function to convert a string token to an integer"

[Seacord 09] "STR06-C. Do not assume that `strtok` leaves the parse string unchanged"

[Seacord 09] "STR07-C. Use TR 24731 for remediation of existing string manipulation code"

6.10 Do not truncate a null-terminated byte string [STR003]

Truncating a null-terminated byte string shall be diagnosed because this is often a sign of programmer error.

Noncompliant Code Example

In this noncompliant code example, `string_data` is truncated if it refers to a string that consists of 16 or more characters.

```
char *string_data;
char a[16];
/* ... */
```



```
strncpy(a, string_data, sizeof(a));
```

Bibliography

[ISO/IEC 9899:1999] Section 7.21, "String handling <string.h>"

[ISO/IEC PDTR 24772] "CJM String Termination"

[ISO/IEC TR 24731-1:2007]

[MITRE 07] CWE ID 170, "Improper Null Termination," CWE ID 464, "Addition of Data Structure Sentinel"

[Seacord 05a] Chapter 2, "Strings"

[Seacord 09] "STR03-C. Do not inadvertently truncate a null-terminated byte string"

6.11 Do not use pointer arithmetic that results in an invalid pointer [ARR038]

Pointer arithmetic that results in an invalid pointer shall be diagnosed (subject to exceptions below) because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the address `&ar[21]` is not a valid pointer. Additionally, `ip` will become invalid.

```
int ar[20];
int *ip;

for (ip = &ar[0]; ip < &ar[21]; ip++) {
  *ip = 0;
}
```

Noncompliant Code Example

In this noncompliant code example, the pointer `buf + len` may not be a valid pointer.

```
char *buf;
size_t len = 1 << 30;

/* Check for overflow */
if (buf + len < buf) {
  len = -(uintptr_t)buf-1;
}
```

Bibliography

[Banahan 03] Section 5.3, "Pointers," and Section 5.7, "Expressions involving pointers"

[ISO/IEC 9899:1999] Section 6.5.6, "Additive operators"

[ISO/IEC PDTR 24772] "XYX Boundary Beginning Violation" and "XYZ Unchecked Array Indexing"

[MITRE 07] CWE ID 129, "Unchecked Array Indexing"

[VU#162289]

[Seacord 09] "ARR38-C. Do not add or subtract an integer to a pointer if the resulting value does not refer to a valid array element"

6.12 Do not loop beyond the end of an array [ARR035]

A loop that iterates beyond the end of an array shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the while loop may loop beyond the end of the arrays `serverName` and `temp`.

```
int getMachineName(char *path) {
  char *temp = path + 2;
  char serverName[MAX_NAME_LENGTH + 1];
  while (temp != L'\\')
    *serverName++ = *temp++;

  /* ... */
}
```

Bibliography

[Finlay 03]

[ISO/IEC 9899:1999] Section 7.1.1, "Definitions of terms," Section 7.20.3.4, "The realloc function," and Section 7.21, "String handling <string.h>"

[Microsoft 03]

[MITRE 07] CWE ID 119, "Failure to Constrain Operations within the Bounds of an Allocated Memory Buffer"

[Pethia 03]

[Seacord 05a] Chapter 1, "Running with Scissors"

[Seacord 09] "ARR35-C. Do not allow loops to iterate beyond the end of an array"

6.13 Do not copy data into an object of insufficient storage size [ARR033]

Copying data into an object of insufficient storage size for that data shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the array `dest` may not have sufficient storage to hold a copy of `src`.

```
enum { WORKSPACE_SIZE = 256 };

void func(const int src[], size_t len) {
  int dest[WORKSPACE_SIZE];
  memcpy(dest, src, len * sizeof(int));
  /* ... */
}
```


Bibliography

[ISO/IEC 9899:1999] Section 7.21.2, "Copying functions," Section 7.21.2.1, "The memcpy function," and Section 5.1.2.2.1, "Program Startup"

[ISO/IEC PDTR 24772] "XYB Buffer Overflow in Heap," "XYW Buffer Overflow in Stack," and "XYZ Unchecked Array Indexing"

[MITRE 07] CWE ID 119, "Failure to Constrain Operations within the Bounds of an Allocated Memory Buffer"

[Seacord 05a] Chapter 2, "Strings"

[Seacord 09] "ARR33-C. Guarantee that copies are made into storage of sufficient size"

6.14 Use array indices that are within the range of the array bounds [ARR030]

Using array indices that outside the bounds of the array shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the array index `pos` may not be within range of the array `table`.

```
enum { TABLESIZE = 100 };

int *table = NULL;

int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *)malloc(sizeof(int) * TABLESIZE);
  }
  if (pos >= TABLESIZE) {
    return -1;
  }
  table[pos] = value;  /* pos could be negative */
  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.5.2, "Array declarators"

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

[MITRE 07] CWE ID 129, "Unchecked Array Indexing"

[Seacord 09] "ARR30-C. Guarantee that array indices are within the valid range"

[Viega 05] Section 5.2.13, "Unchecked array indexing"

6.15 Do not apply the sizeof operator to a pointer when taking the size of an array [ARR001]

Using the `sizeof` operator on a function parameter that has array type shall be diagnosed because this typically indicates programmer error and can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the `sizeof` operator is used on the array parameter `array`.

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

void dowork(void) {
  int dis[12];

  clear(dis);
  /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, `sizeof` is used on the array parameter `a`.

```
enum {ARR_LEN = 100};

void clear(int a[ARR_LEN]) {
  memset(a, 0, sizeof(a)); /* error */
}

int main(void) {
  int b[ARR_LEN];
  clear(b);
  assert(b[ARR_LEN / 2]==0); /* may fail */
  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.5.2, "Array declarators" [Drepper 06] Section 2.1.1, "Respecting Memory Bounds"

[MITRE 07] CWE ID 467, "Use of sizeof on a Pointer Type"

[Seacord 09] "ARR01-C. Do not apply the sizeof operator to a pointer when taking the size of an array"

6.16 Evaluate integer expressions in a larger size before using with that size [INT035]

Evaluating an integer expression in a size smaller than the size it is being compared to or assigned to shall be diagnosed because doing so can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, `length` is not cast to `unsigned long long` before being added to `BLOCK_HEADER_SIZE` and compared to another `unsigned long long` value.

```
enum { BLOCK_HEADER_SIZE = 16 };

void *AllocateBlock(size_t length) {
  struct memBlock *mBlock;

  if (length + BLOCK_HEADER_SIZE > (unsigned long long)SIZE_MAX)
    return NULL;
  mBlock = (struct memBlock *)malloc(
    length + BLOCK_HEADER_SIZE
  );
```



```
  if (!mBlock) return NULL;

  /* fill in block header and return data portion */

  return mBlock;
}
```

Noncompliant Code Example

In this noncompliant code example, `cBlocks` is not cast to `unsigned long long` before being multiplied by 16 and stored in a variable of type `unsigned long long`.

```
void* AllocBlocks(size_t cBlocks) {
  if (cBlocks == 0) return NULL;
  unsigned long long alloc = cBlocks * 16;
  return (alloc < UINT_MAX) ? malloc(cBlocks * 16) : NULL;
}
```

Bibliography

[Dowd 06] Chapter 6, "C Language Issues"

[ISO/IEC 9899:1999] Section 6.3.1, "Arithmetic operands"

[ISO/IEC PDTR 24772] "FLC Numeric Conversion Errors"

[MITRE 07] CWE ID 681, "Incorrect Conversion between Numeric Types," and CWE ID 190, "Integer Overflow (Wrap or Wraparound)"

[Seacord 05a] Chapter 5, "Integer Security"

[Seacord 09] "INT35-C. Evaluate integer expressions in a larger size before comparing or assigning to that size"

6.17 Do not let operations on signed integers result in overflow [INT032]

Signed integer overflow shall be diagnosed because it results in undefined behavior.

The following table indicates which operators can result in overflow.

Operator	Overflow	Operator	Overflow	Operator	Overflow	Operator	Overflow
+	yes	-=	yes	<<	yes	<	no
-	yes	*=	yes	>>	no	>	no
*	yes	/=	yes	&	no	>=	no
/	yes	%=	yes	|	no	<=	no
%	yes	<<=	yes	^	no	==	no
++	yes	>>=	no	~	no	!=	no
--	yes	&=	no	!	no	&&	no
=	no	|=	no	un +	no	||	no

+=	yes	^=	no	un -	yes	?:	no

Noncompliant Code Example

In this noncompliant code example, all of the arithmetic operations can result in integer overflow.

```
int si1, si2, result;

/* Initialize si1 and si2 */
result = si1 + si2;
result = si1 - si2;
result = si1 * si2;
result = si1 / si2;
result = si1 % si2;
result = si1 << si2;
result = -si1;
```

Bibliography

[Dowd 06] Chapter 6, "C Language Issues" (Arithmetic Boundary Conditions, pp. 211-223)

[ISO/IEC 9899:1999] Section 6.5, "Expressions," and Section 7.10, "Sizes of integer types <limits.h>"

[ISO/IEC PDTR 24772] "XYY Wrap-around Error"

[MITRE 07] CWE ID 129, "Unchecked Array Indexing" and CWE ID 190, "Integer Overflow (Wrap or Wraparound)"

[Seacord 05] Chapter 5, "Integers"

[Seacord 09] "INT32-C. Ensure that operations on signed integers do not result in overflow"

[Viega 05] Section 5.2.7, "Integer overflow"

[VU#551436]

[Warren 02] Chapter 2, "Basics"

6.18 Do not let integer conversions result in lost data or sign [INT031]

Integer conversions that result in lost data or sign shall be diagnosed because this can result in undefined or unexpected behavior. Conversely, integer conversions that are provably correct need not be diagnosed. The only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness.

Noncompliant Code Example

In this noncompliant code example, the conversion of `ul` from `unsigned long` to `signed char` results in a truncation error on most implementations.

```
unsigned long ul = ULONG_MAX;
signed char sc = (signed char)ul;
```


Noncompliant Code Example

In this noncompliant code example, the conversion of `si` from `signed int` to `unsigned int` results in a loss of sign error.

```
signed int si = INT_MIN;
unsigned int ui = (unsigned int)si;
```

Noncompliant Code Example

In this noncompliant code example, the conversion of `sl` from a `signed long` to a `signed char` results in a truncation error on most implementations.

```
signed long sl = LONG_MAX;
signed char sc = (signed char)sl;
```

Noncompliant Code Example

In this noncompliant code example, the conversion of `ul` from `unsigned long` to `unsigned char` results in a truncation error on most implementations.

```
unsigned long ul = ULONG_MAX;
unsigned char uc = (unsigned char)ul;
```

Bibliography

[Dowd 06] Chapter 6, "C Language Issues" (Type Conversions, pp. 223-270)

[ISO/IEC 9899:1999] 6.3, "Conversions"

[ISO/IEC PDTR 24772] "FLC Numeric Conversion Errors"

[MISRA 04] Rules 10.1, 10.3, 10.5, and 12.9

[MITRE 07] CWE ID 192, "Integer Coercion Error," CWE ID 197, "Numeric Truncation Error," and CWE ID 681, "Incorrect Conversion between Numeric Types"

[Seacord 05a] Chapter 5, "Integers"

[Seacord 09] "INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data"

[Viega 05] Section 5.2.9, "Truncation error," Section 5.2.10, "Sign extension error," Section 5.2.11, "Signed to unsigned conversion error," and Section 5.2.12, "Unsigned to signed conversion error"

[Warren 02] Chapter 2, "Basics"

6.19 Verify that all integer values are in range [INT008]

Integer operations shall result in an integer value within the range of the integer type. A program that detects an integer overflow has occurred or is inevitable may either signal an exception[2] or produce an integer result that is within the range of representable integers on that system[3].

---

[2] This approach is implemented by the as-if infinitley ranged integer model.

[3] For example, by implementing saturation semantics. For saturation semantics, if the mathematical result of a computation result is greater than the maximum representable integer value for a type MAX the operation returns MAX. If

Some situations are best handled by an error condition, where an overflow causes a change in control flow (such as the system complaining about bad input and requesting alternative input from the user). Others are better handled by the latter option because it allows the computation to proceed and generate an integer result, thereby avoiding a denial-of-service attack.

Noncompliant Code Example

In this noncompliant example, `i + 1` will overflow on a 16-bit machine. The C standard allows signed integers to overflow and produce incorrect results. Compilers can take advantage of this to produce faster code by assuming an overflow will not occur. As a result, the `if` statement that is intended to catch an overflow might be optimized away.

```
int i = /* Expression that evaluates to the value 32767 */;
/* ... */
if (i + 1 <= i) {
  /* handle overflow */
}
/* expression involving i + 1 */
```

Bibliography

[ISO/IEC PDTR 24772] "FLC Numeric Conversion Errors"

[Keaton 2009] David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky. As-if Infinitely Ranged Integer Model. CMU/SEI-2009-TN-XXX.

[Seacord 09] "INT08-C. Verify that all integer values are in range"

6.20 Enforce limits on integer values originating from untrusted sources [INT004]

Untrusted integer values that are not restricted to some subset of integer values shall be diagnosed because using untrusted integer values can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, `length` is an unrestricted user-supplied argument that is used to determine the length of `table`.

```
int create_table(size_t length) {
  char **table;

  if (sizeof(char *) > SIZE_MAX/length) {
    /* handle overflow */
    return -1;
  }

  size_t table_length = length * sizeof(char *);
  table = (char **)malloc(table_length);

  if (table == NULL) {
    /* Handle error condition */
    return -1;
  }
  /* ... */
  return 0;
```

---

result is less the minimum representable integer value for a type MIN the operation returns MIN. In all other cases the operation returns result.


```
}
```

Bibliography

[Seacord 05a] Chapter 5, "Integer Security"

[Seacord 09] "INT04-C. Enforce limits on integer values originating from untrusted sources"

7 Direct Result

7.1 Do not create a universal character name through concatenation [PRE030]

A universal character name created via concatenation shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, a universal character name is produced by token concatenation.

```
#define assign(byte1, byte2, byte3, byte4, val) \
  \U ## byte1 ## byte2 ## byte3 ## byte4 = val;

int \U00010401;
assign(00, 01, 04, 01, 4);
```

Bibliography

[ISO/IEC 10646-2003]

[ISO/IEC 9899:1999] Section 5.1.1.2, "Translation phases," Section 6.4.3, "Universal character names," and Section 6.10.3.3, "The ## operator"

[Seacord 09] "PRE30-C. Do not create a universal character name through concatenation"

7.2 Do not shift a negative number of bits or more bits than exist in the operand [INT034]

Using a bitwise shift to shift an operand a negative number of bits or more bits than exist in the operand shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, `si1` could be shifted a negative number of bits or more bits than exist in `si1`.

```
int si1, si2, sresult;

/* Initialize si1 and si2 */

sresult = si1 << si2;
```

Noncompliant Code Example

In this noncompliant code example, `ui1` could be shifted by more bits than exist in `ui1`.

```
unsigned int ui1;
unsigned int ui2;
unsigned int uresult;
```

```
/* Initialize ui1 and ui2 */

uresult = ui1 << ui2;
```

Bibliography

[Dowd 06] Chapter 6, "C Language Issues"

[ISO/IEC 9899:1999] Section 6.5.7, "Bitwise shift operators"

[ISO/IEC PDTR 24772] "XYY Wrap-around Error"

[Seacord 05a] Chapter 5, "Integers"

[Viega 05] Section 5.2.7, "Integer overflow"

[ISO/IEC 03] Section 6.5.7, "Bitwise shift operators"

[Seacord 09] "INT34-C. Do not shift a negative number of bits or more bits than exist in the operand"

7.3 Use the correct syntax for flexible array members [MEM033]

Incorrectly declaring a flexible array member shall be diagnosed because this can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the member array `data` is declared and initialized as a C89 flexible array member, but this is incorrect.

```
struct flexArrayStruct {
  int num;
  int data[1];
};

/* ... */

size_t array_size;
size_t i;

/* initialize array_size */

/* space is allocated for the struct */
struct flexArrayStruct *structP
  = (struct flexArrayStruct *)
     malloc(sizeof(struct flexArrayStruct)
          + sizeof(int) * (array_size - 1));
if (structP == NULL) {
  /* Handle malloc failure */
}
structP->num = 0;

/* access data[] as if it had been allocated
 * as data[array_size] */
for (i = 0; i < array_size; i++) {
  structP->data[i] = 1;
}
```


Bibliography

[ISO/IEC 9899:1999] Section 6.7.2.1, "Structure and union specifiers"

[McCluskey 01] ;login:, July 2001, Volume 26, Number 4

[Seacord 09] "MEM33-C. Use the correct syntax for flexible array members"

7.4 Do not call remove on an open file [FIO008]

Invoking `remove` on an open file shall be diagnosed because doing so results in implementation-defined behavior.

Noncompliant Code Example

In this noncompliant code example, a file is removed while it is still open.

```
char *file_name;
FILE *file;

/* initialize file_name */

file = fopen(file_name, "w+");
if (file == NULL) {
  /* Handle error condition */
}

/* ... */

if (remove(file_name) != 0) {
  /* Handle error condition */
}

/* continue performing I/O operations on file */

fclose(file);
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.4.1, "The `remove` function"

[Seacord 09] "FIO08-C. Take care when calling remove on an open file"

7.5 Do not use non-standard mode parameters when calling fopen [FIO011]

Using non-standard `mode` parameters when calling `fopen` shall be diagnosed because this results in undefined behavior.

The following table details the `mode` strings available for use with `fopen`.

`mode` string	Result
`r`	open text file for reading
`w`	truncate to zero length or create text file for writing

a	append; open or create text file for writing at end-of-file
rb	open binary file for reading
wb	truncate to zero length or create binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create text file for update
a+	append; open or create text file for update, writing at end-of-file
r+b or rb+	open binary file for update (reading and writing)
w+b or wb+	truncate to zero length or create binary file for update
a+b or ab+	append; open or create binary file for update, writing at end-of-file

Bibliography

[ISO/IEC 9899:1999] Section 7.9.15.3, "The `fopen` function"

[Seacord 09] "FIO11-C. Take care when specifying the mode parameter of `fopen`"

7.6 Do not push back anything other than one read character [FIO013]

Multiple subsequent calls to `ungetc` or `ungetwc` on the same stream without an intervening call to a read function or a file-positioning function on that stream shall be diagnosed. Only one character of pushback is guaranteed with the `ungetc` and `ungetwc` functions.

Noncompliant Code Example

In this noncompliant code example, more than one character is pushed back on the stream referenced by `fp` without an intervening call to a read function or a file-positioning function.

```
FILE *fp;
char *file_name;

/* initialize file_name */

fp = fopen(file_name, "rb");
if (fp == NULL) {
  /* Handle error */
}

/* read data */

if (ungetc('\n', fp) == EOF) {
  /* Handle error */
}
if (ungetc('\r', fp) == EOF) {
```



```
  /* Handle error */
}

/* continue on */
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.7.11, "The `ungetc` function"

[Seacord 09] "FIO13-C. Never push back anything other than one read character"

7.7 Do not simultaneously open the same file multiple times [FIO031]

Simultaneously opening a file multiple times shall be diagnosed because doing so results in implementation-defined behavior.

Noncompliant Code Example

In this noncompliant code example, the file named `log` is opened twice simultaneously.

```
void do_stuff(void) {
  FILE *logfile = fopen("log", "a");
  if (logfile == NULL) {
    /* Handle error */
  }

  /* Write logs pertaining to do_stuff */
  fprintf(logfile, "do_stuff\n");

  /* ... */
}

int main(void) {
  FILE *logfile = fopen("log", "a");
  if (logfile == NULL) {
    /* Handle error */
  }

  /* Write logs pertaining to main */
  fprintf(logfile, "main\n");

  do_stuff();

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

Bibliography

[ISO/IEC 9899:1999] Section 7.19.3, "Files"

[MITRE 07] CWE ID 362, "Race Condition," CWE ID 675, and "Duplicate Operations on Resource"

[Seacord 09] "FIO31-C. Do not simultaneously open the same file multiple times"

7.8 Use feof and ferror to detect file conditions when sizeof(int) == sizeof(char) [FIO035]

When `sizeof(int) == sizeof(char)`, the comparison of the return value of a character IO function to `EOF` shall be diagnosed because character values may be indistinguishable from `EOF`.

Noncompliant Code Example

In this noncompliant code example, `sizeof(int) == sizeof(char)` and the return value of `getchar` is compared to `EOF`.

```
int c;

do {
  /* ... */
  c = getchar;
  /* ... */
} while (c != EOF);
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.7, "Character input/output functions," Section 7.19.10.2, "The `feof` function," and Section 7.19.10.3, "The `ferror` function"

[Kettlewell 02] Section 1.2, "<`stdio.h`> and Character Types"

[Summit 05] Question 12.2

[Seacord 09] "FIO35-C. Use `feof` and `ferror` to detect end-of-file and file errors when `sizeof(int) == sizeof(char)`"

7.9 Do not use a copy of a FILE object for input and output [FIO038]

Copying a `FILE` object for use in input or output operations shall be diagnosed because the copy need not serve in place of the original.

Noncompliant Code Example

In this noncompliant code example, the `FILE` object `stdout` is copied and used in the call to `fputs`.

```
int main(void) {
  FILE my_stdout = *(stdout);
  if (fputs("Hello, World!\n", &my_stdout) == EOF) {
    /* Handle error */
  }
  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.3, "Files"

[Seacord 09] "FIO38-C. Do not use a copy of a `FILE` object for input and output"

7.10 Do not input and output from a stream without a flush or positioning call [FIO039]

Receiving input from a stream directly following an output to that stream without an intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`, or outputting to a stream after receiving input from it without a call to `fseek`, `fsetpos`, `rewind` if the file is not at end-of-file shall be diagnosed because doing so results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, data is appended to a file and then the same file is read without an intervening flush of the stream.

```
char data[BUFSIZE];
char append_data[BUFSIZE];
char *file_name;
FILE *file;

/* Initialize file_name */

file = fopen(file_name, "a+");
if (file == NULL) {
  /* Handle error */
}

/* initialize append_data */

if (fwrite(append_data, BUFSIZE, 1, file) != BUFSIZE) {
  /* Handle error */
}
if (fread(data, BUFSIZE, 1, file) != 0) {
  /* Handle there not being data */
}

fclose(file);
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.5.3, "The `fopen` function"

[Seacord 09] "FIO39-C. Do not alternately input and output from a stream without an intervening flush or positioning call"

7.11 Only use values for fsetpos that are returned from fgetpos [FIO044]

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

Noncompliant Code Example

In this noncompliant code example, a value other than the one returned from `fgetpos` is used in a call to `fsetpos`.

```
enum { NO_FILE_POS_VALUES = 3 };

int opener(FILE* file, /* ... */ ) {
  int rc;
  fpos_t offset;

  /* ... */

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

  if (file == NULL) { return EINVAL; }

  /* Read in data from file */
```

```
  rc = fsetpos(file, &offset);
  if (rc != 0 ) { return rc; }

  /* ... */

  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.9.3, "The `fsetpos` function"

[Seacord 09] "FIO44-C. Only use values for `fsetpos` that are returned from `fgetpos`"

7.12 Use consistent array notation across all source files [ARR031]

Using inconsistent notation to declare arrays shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the array `a` is declared inconsistently in the files `main.c` and `insert_a.c`.

```
/* main.c source file */
#include <stdlib.h>

enum { ARRAYSIZE = 100 };

char *a;

void insert_a(void);

int main(void) {
  a = (char *)malloc(ARRAYSIZE);
  if (a == NULL) {
    /* Handle allocation error */
  }
  insert_a;
  return 0;
}

/* insert_a.c source file */
char a[];

void insert_a(void) {
  a[0] = 'a';
}
```

Bibliography

[Hatton 95] Section 2.8.3

[ISO/IEC 9899:1999] Section 6.7.5.2, "Array declarators," and Section 6.2.2, "Linkages of identifiers"

[Seacord 09] "ARR31-C. Use consistent array notation across all source files"

7.13 Do not manipulate time_t typed values directly [MSC005]

Manipulating values of type `time_t` shall be diagnosed because the encoding of `time_t` values is unspecified.

Noncompliant Code Example

In this noncompliant code example, `start`, which has type `time_t`, is added to `seconds_to_work`.

```
int do_work(int seconds_to_work) {
  time_t start = time(NULL);

  if (start == (time_t)(-1)) {
    /* Handle error */
  }
  while (time(NULL) < start + seconds_to_work) {
    /* ... */
  }
  return 0;
}
```

Bibliography

[Kettlewell 02] Section 4.1, "time_t"

[ISO/IEC 9899:1999] Section 7.23, "Date and time <time.h>"

[Seacord 09] "MSC05-C. Do not manipulate time_t typed values directly"

7.14 Do not invoke an unsafe macro with arguments containing side effects [PRE031]

An unsafe function-like macro is one that evaluates a parameter more than once in the code expansion or never evaluates the parameter at all. Invoking an unsafe function-like macro with arguments containing assignment, increment, decrement, volatile access, function call, or other side effects shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the function-like macro `ABS` evaluates its argument multiple times.

```
#define ABS(x) (((x) < 0) ? -(x) : (x))
/* ... */
m = ABS(++n);
```

The above code example expands to

```
m = (((++n) < 0) ? -(++n) : (++n))
```

which results in undefined behavior.

Bibliography

[ISO/IEC 9899:1999] Section 5.1.2.3, "Program execution"

[ISO/IEC PDTR 24772] "NMP Pre-processor Directions"

[MISRA 04] Rule 19.6

[Plum 85] Rule 1-11

[Seacord 09] "PRE31-C. Never invoke an unsafe macro with arguments containing assignment, increment, decrement, volatile access, or function call"

7.15 Do not refer to an object outside its lifetime [DCL030]

Accessing an object outside of its lifetime shall be diagnosed because doing so results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the local variable `str` is assigned to the variable `p`, which has file scope.

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

void innocuous(void) {
    const char str[] = "Surprise, surprise";
}
/* ... */
dont_do_this();
innocuous();
/* p might be pointing to "Surprise, surprise" */
```

Noncompliant Code Example

In this noncompliant code example, the function `init_array` returns a pointer to a local stack variable, which could be accessed by the caller.

```
char *init_array(void) {
   char array[10];
   /* Initialize array */
   return array;
}
```

Bibliography

[Coverity 07]

[ISO/IEC 9899:1999] Section 6.2.4, "Storage durations of objects," and Section 7.20.3, "Memory management functions"

[ISO/IEC PDTR 24772] "DCM Dangling references to stack frames"

[MISRA 04] Rule 8.6

[Seacord 09] "DCL30-C. Do not refer to an object outside its lifetime"

7.16 Do not use restrict-qualified pointers that overlap as parameters [DCL033]

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.

Noncompliant Code Example

In this noncompliant code example, the values of objects referenced by `ptr1` and `ptr2` become unpredictable after the call to `memcpy` because their memory areas overlap.

```
char str[]= "test string";
char *ptr1 = str;
char *ptr2;

ptr2 = ptr1 + 3;
memcpy(ptr2, ptr1, 6);
```

Bibliography

[ISO/IEC 9899:1999] Section 7.21.2, "Copying functions," and Section 6.7.3, "Type qualifiers"

[ISO/IEC PDTR 24772] "CSJ Passing parameters and return values"

[Seacord 09] "DCL33-C. Ensure that `restrict`-qualified source and destination pointers in function arguments do not reference overlapping objects"

7.17 Do not declare an identifier with conflicting linkage classifications [DCL036]

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

Noncompliant Code Example

In this noncompliant code example, `i2` and `i5` are defined as having both internal and external linkage. Future use of either identifier results in undefined behavior.

```
int i1 = 10;  /* definition, external linkage */
static int i2 = 20;  /* definition, internal linkage */
extern int i3 = 30;  /* definition, external linkage */
int i4;  /* tentative definition, external linkage */
static int i5;  /* tentative definition, internal linkage */

int i1;  /* valid tentative definition */
int i2;  /* not valid, linkage disagreement with previous */
int i3;  /* valid tentative definition */
int i4;  /* valid tentative definition */
int i5;  /* not valid, linkage disagreement with previous */

int main(void) {
  /* ... */
}
```

Bibliography

[Banahan 03] Section 8.2, "Declarations, Definitions and Accessibility"

[ISO/IEC 9899:1999] Section 6.2.2, "Linkages of identifiers"

[Kirch-Prinz 02]

[MISRA 04] Rule 8.1

[Seacord 09] "DCL36-C. Do not declare an identifier with conflicting linkage classifications"

7.18 Do not cast away const qualification [EXP005]

Using a cast to remove `const` qualification shall be diagnosed because doing so results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the `const` qualification of `str` is cast away.

```
void remove_spaces(const char *str, size_t slen) {
  char *p = (char *)str;
  size_t i;
  for (i = 0; i < slen && str[i]; i++) {
    if (str[i] != ' ') *p++ = str[i];
  }
  *p = '\0';
}
```

Noncompliant Code Example

In this noncompliant code example, the `const` qualification of the array `vals` is implicitly cast away in the call to `memset`.

```
const int vals[3] = {3, 4, 5};
memset(vals, 0, sizeof(vals));
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.3, "Type qualifiers"

[ISO/IEC PDTR 24772] "HFC Pointer casting and pointer type changes" and "IHN Type system"

[MISRA 04] Rule 11.5

[MITRE 07] CWE ID 704, "Incorrect Type Conversion or Cast"

[Seacord 09] "EXP05-C. Do not cast away a `const` qualification"

7.19 Do not apply operators expecting one type to data of an incompatible type [EXP011]

Applying operators to operands of the wrong type shall be diagnosed because doing so results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, both an `int` cast to a `float` and a `float` cast to an `int` are incremented.

```
float f = 0.0;
int i = 0;
float *fp;
int *ip;

assert(sizeof(int) == sizeof(float));
ip = (int*) &f;
fp = (float*) &i;
printf("int is %d, float is %f\n", i, f);
(*ip)++;
```



```
(*fp)++;

printf("int is %d, float is %f\n", i, f);
```

Noncompliant Code Example

In this noncompliant code example, the behavior depends on how bit-fields are implemented.

```
struct bf {
  unsigned int m1 : 8;
  unsigned int m2 : 8;
  unsigned int m3 : 8;
  unsigned int m4 : 8;
}; /* 32 bits total */

void function {
  struct bf data;
  unsigned char *ptr;

  data.m1 = 0;
  data.m2 = 0;
  data.m3 = 0;
  data.m4 = 0;
  ptr = (unsigned char *)&data;
  (*ptr)++; /* can increment data.m1 or data.m4 */
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.2, "Type specifiers"

[ISO/IEC PDTR 24772] "STR Bit Representations"

[MISRA 04] Rule 3.5

[Plum 85] Rule 6-5

[Seacord 09] "EXP11-C. Do not apply operators expecting one type to data of an incompatible type"

7.20 Do not depend on order of evaluation between sequence points [EXP030]

Expressions that rely on a specific order of evaluation between sequence points shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the order of evaluation of the operands to the + operator is unspecified.

```
a = i + b[++i];
```

Noncompliant Code Example

In this noncompliant code example, the order of evaluation for function arguments is unspecified.

```
func(i++, i);
```

Bibliography

[ISO/IEC 9899:1999] Section 5.1.2.3, "Program execution," Section 6.5, "Expressions," and Annex C, "Sequence points"

[ISO/IEC PDTR 24772] "JCW Operator precedence/Order of Evaluation" and "SAM Side-effects and order of evaluation"

[MISRA 04] Rule 12.1

[Saks 07]

[Seacord 09] "EXP30-C. Do not depend on order of evaluation between sequence points"

[Summit 05] Questions 3.1, 3.2, 3.3, 3.3b, 3.7, 3.8, 3.9, 3.10a, 3.10b, and 3.11

7.21 Do not cast away a volatile qualification [EXP032]

Using a cast to remove `volatile` qualification shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, a `volatile`-qualified object is accessed through a non-`volatile`-qualified reference, resulting in undefined behavior.

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

printf("i = %d.\n", i);

ipp = &ip; /* produces warnings in modern compilers */
ipp = (int**) &ip; /* constraint violation, also produces warnings */
*ipp = &i; /* valid */
if (*ip != 0) { /* valid */
  /* ... */
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.3, "Type qualifiers," and Section 6.5.16.1, "Simple assignment"

[ISO/IEC PDTR 24772] "HFC Pointer casting and pointer type changes" and "IHN Type system"

[MISRA 04] Rule 11.5

[Seacord 09] "EXP32-C. Do not cast away a `volatile` qualification"

7.22 Do not reference uninitialized memory [EXP033]

Referencing unitialized memory shall be diagnosed because the value of uninitialized memory is indeterminate.

Noncompliant Code Example

In this noncompliant code example, the variable `sign` may be unintialized depending on the value of `number`.

```
void set_flag(int number, int *sign_flag) {
```



```
  if (sign_flag == NULL) {
    return;
  }
  if (number > 0) {
    *sign_flag = 1;
  }
  else if (number < 0) {
    *sign_flag = -1;
  }
}

void func(int number) {
  int sign;

  set_flag(number, &sign);
  /* use sign */
}
```

Noncompliant Code Example

In this noncompliant code example, the variable `error_log` remains uninitialized in the call to `sprintf`.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int do_auth(void) {
  char *username;
  char *password;

  /* Get username and password from user, return -1 if invalid */
}

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

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

int main(void) {
  if (do_auth == -1) {
    report_error("Unable to login");
  }
  return 0;
}
```

Bibliography

[Flake 06]

[ISO/IEC 9899:1999] Section 6.7.8, "Initialization"

[ISO/IEC PDTR 24772] "LAV Initialization of Variables"

[Mercy 06]

[Seacord 09] "EXP33-C. Do not reference uninitialized memory"

7.23 Do not call functions with incorrect arguments [EXP037]

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

Noncompliant Code Example

In this noncompliant code example, the function pointer `fp` is used to call `strchar` with the wrong type of arguments.

```
#include <stdio.h>
#include <string.h>

char *(*fp) ;

int main(void) {
  char *c;
  fp = strchr;
  c = fp(12, 2);
  printf("%s\n", c);
  return 0;
}
```

Noncompliant Code Example

In this noncompliant code example, the third argument to `open`, a variadic function, has been omitted.

```
fd = open(ms, O_CREAT|O_EXCL|O_WRONLY|O_TRUNC);
```

Bibliography

[CVE] CVE-2006-1174

[ISO/IEC 9899:1999] Forward and Section 6.9.1, "Function definitions"

[ISO/IEC PDTR 24772] "OTR Subprogram Signature Mismatch"

[MISRA 04] Rule 16.6

[MITRE 07] CWE ID 628, "Function Call with Incorrectly Specified Arguments"

[Seacord 09] "EXP37-C. Call functions with the arguments intended by the API"

[Spinellis 06] Section 2.6.1, "Incorrect Routine or Arguments"

7.24 Do not dereference a null pointer [EXP034]

Dereferencing a null pointer shall be diagnosed because doing so results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, `str` could be a null pointer if `malloc` fails. Then the call to `memcpy` would result in dereferencing a null pointer.

```
size_t size = strlen(input_str)+1;
str = (char *)malloc(size);
memcpy(str, input_str, size);
/* ... */
```



```
free(str);
str = NULL;
```

Bibliography

[ISO/IEC 9899:1999] Section 6.3.2.3, "Pointers"

[ISO/IEC PDTR 24772] "HFC Pointer casting and pointer type changes" and "XYH Null Pointer Dereference"

[Jack 07]

[MITRE 07] CWE ID 476, "NULL Pointer Dereference"

[Seacord 09] "EXP34-C. Ensure a null pointer is not dereferenced"

[van Sprundel 06]

[Viega 05] Section 5.2.18, "Null-pointer dereference"

7.25 Do not access an array in the result of a function call after a sequence point [EXP035]

Accessing or modifying an array in the result of a function call after a subsequent sequence point shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the array `a` in the struct returned from `addressee` is accessed in the call to `printf`, resulting in undefined behavior.

```
#include <stdio.h>

struct X { char a[6]; };

struct X addressee(void) {
  struct X result = { "world" };
  return result;
}

int main(void) {
  printf("Hello, %s!\n", addressee().a);
  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.5.2.2, "Function calls"

[ISO/IEC PDTR 24772] "DCM Dangling references to stack frames" and "SAM Side-effects and order of evaluation"

[Seacord 09] "EXP35-C. Do not access or modify an array in the result of a function call after a subsequent sequence point"

7.26 Do not convert pointers into more strictly aligned pointer types [EXP036]

Converting a pointer into a more strictly aligned pointer type shall be diagnosed because this results in undefined behavior if the pointer becomes unaligned.

Noncompliant Code Example

In this noncompliant code example, a `char` pointer is converted to an `int` pointer, which has stricter alignment.

```
char *loop_ptr;
int *int_ptr;

int *loop_function(void *v_pointer) {
  /* ... */
  return v_pointer;
}
int_ptr = loop_function(loop_ptr);
```

Bibliography

[Bryant 03]

[ISO/IEC 9899:1999] Section 6.2.5, "Types"

[ISO/IEC PDTR 24772] "HFC Pointer casting and pointer type changes"

[MISRA 04] Rules 11.2 and 11.3

[Seacord 09] "EXP36-C. Do not convert pointers into more strictly aligned pointer types"

7.27 Do not call offsetof on bit-field members or invalid types [EXP038]

Using the `offsetof` macro on bit-field members or invalid types shall be diagnosed because this results in undefined behavior. An invalid type is one whose members do not include the member provided to `offsetof`.

Noncompliant Code Example

In this noncompliant code example, `offsetof` is called on a bit-field structure member.

```
struct S {
  unsigned int a: 8;
} bits = {255};

int main(void) {
  size_t offset = offsetof(struct S, a);  /* error */
  printf("offsetof(bits, a) = %d.\n", offset );
  return 0;
}
```

Noncompliant Code Example

In this noncompliant code example, `offsetof` is used the invalid type `struct T`.

```
struct S {
  int i, j;
} s;

struct T {
  float f;
} t;

int main(void) {
```

```
  return offsetof(struct T, j);
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.17, "Common definitions <stddef.h>"

[ISO/IEC PDTR 24772] "STR Bit Representations"

[Jones 04]

[MISRA 04] Rule 20.6, "The macro offsetof, in library <stddef.h>, shall not be used."

[Seacord 09] "EXP38-C. Do not call offsetof on bit-field members or invalid types"

7.28 Do not access a variable through a pointer of an incompatible type [EXP039]

Accessing a variable through a pointer of an incompatible type shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, an object of type double is accessed through a pointer to int.

```
union a_union {
  int i;
  double d;
};

int f() {
  a_union t;
  int *ip;
  t.d = 3.0;
  ip = &t.i;
  return *ip;
}
```

Noncompliant Code Example

In this noncompliant code example, access by taking the address, casting the resulting pointer and dereferencing the result has undefined behavior, even if the cast uses a union type.

```
union a_union {
  int i;
  double d;
};

int f {
  double d = 3.0;
  return ((union a_union *) &d)->i;
}
```

Noncompliant Code Example

In this noncompliant code example, an array of two shorts is assigned a value as if it were an int.

```
short a[2];
```

```
a[0] = 0x1111;
a[1] = 0x1111;

*(int *)a = 0x22222222;  /* violation of aliasing rules */

printf("%x %x\n", a[0], a[1]);
```

Bibliography

[Acton 06] Mike Acton. Understanding Strict Aliasing. June 01, 2006.

[ISO/IEC 9899:1999] Section 6.5, "Expressions"

[Seacord 09] "EXP39-C. Do not access a variable through a pointer of an incompatible type"

[Walfridsson 03] Krister Walfridsson. Aliasing, pointer casts and gcc 3.3 Aliasing issue. August, 2003

7.29 Do not convert a pointer to integer or integer to pointer [INT011]

Converting an integer to a pointer or a pointer to an integer shall be diagnosed because the result is implementation-defined.

Noncompliant Code Example

In this noncompliant code example, the pointer `ptr` is converted to an integer value and the integer `number` is converted to a pointer value.

```
char *ptr;
unsigned int flag;
/* ... */
unsigned int number = (unsigned int)ptr;
number = (number & 0x7fffff) | (flag << 23);
ptr = (char *)number;
```

Non-Compliant Code Example

In this noncompliant code example, an integer literal is converted to a pointer.

```
unsigned int *ptr = 0xdeadbeef;
```

Bibliography

[ISO/IEC 9899:1999] Section 6.3.2.3, "Pointers"

[ISO/IEC PDTR 24772] "HFC Pointer casting and pointer type changes"

[MITRE 07] CWE ID 466, "Return of Pointer Value Outside of Expected Range," and CWE ID 587, "Assignment of a Fixed Address to a Pointer"

[Seacord 09] "INT11-C. Take care when converting from pointer to integer or integer to pointer"

7.30 Do not let division and modulo operations result in divide-by-zero errors [INT033]

Divide-by-zero errors shall be diagnosed because they result in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the division of `sl1` by `sl2` can result in a divide-by-zero error.


```
signed long sl1, sl2, result;

/* Initialize sl1 and sl2 */
result = sl1 / sl2;
```

Noncompliant Code Example

In this noncompliant code example, the modulus of `sl1` taken by `sl2` can result in a divide-by-zero error.

```
signed long sl1, sl2, result;

/* Initialize sl1 and sl2 */
result = sl1 % sl2;
```

Bibliography

[ISO/IEC 9899:1999] Section 6.5.5, "Multiplicative operators"

[MITRE 07] CWE ID 369, "Divide By Zero"

[Seacord 05] Chapter 5, "Integers"

[Seacord 09] "INT33-C. Ensure that division and modulo operations do not result in divide-by-zero errors"

[Warren 02] Chapter 2, "Basics"

7.31 Do not call functions expecting real values with complex values [FLP031]

Calling functions with complex values that expect real values shall be diagnosed because doing so results in undefined behavior.

The following functions should not be called with complex values.

atan2	cbrt	ceil	copysign	erf
erfc	exp2	expm1	fdim	floor
fma	fmax	fmin	fmod	frexp
hypot	ilogb	ldexp	lgamma	llrint
llround	log10	log1p	log2	logb
lrint	lround	nearbyint	nextafter	nexttoward
remainder	remquo	rint	round	scalbn
scalbln	tgamma	trunc		

Noncompliant Code Example

In this noncompliant code example, `log2c` is called with a complex value.

```
double complex c = 2.0 + 4.0 * I;
/* ... */
double complex result = log2(c);
```

Bibliography

[ISO/IEC 9899:1999] Section 7.22, "Type-generic math <tgmath.h>"

[ISO/IEC PDTR 24772] "OTR Subprogram Signature Mismatch"

[MITRE 07] CWE ID 686, "Function Call With Incorrect Argument Type"

[Seacord 09] "FLP31-C. Do not call functions expecting real values with complex values"

7.32 Do not compare floating point values [FLP035]

Comparing floating point values shall be diagnosed because this results in an implementation-defined value.

Noncompliant Code Example

In this noncompliant code example, two floating point values are compared for equality.

```
float a = 3.0;
float b = 7.0;
float c = a / b;

if (c == a / b) {
  puts("Comparison succeeds");
} else {
  puts("Unexpected result");
}
```

Bibliography

[Hatton 95] Section 2.7.3 Floating-point misbehavior.

[Lockheed Martin 05] AV Rule 202 Floating point variables shall not be tested for exact equality or inequality.

[Seacord 09] "FLP35-C. Take granularity into account when comparing floating point values"

7.33 Do not modify string literals [STR030]

Modifying a string literal shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the string literal referred to by `p` is modified.

```
char *p  = "string literal";
p[0] = 'S';
```

Noncompliant Code Example

In this noncompliant code example, a string literal is passed to a function that modifies it.

```
mktemp("/tmp/edXXXXXX");
```

Bibliography

[ISO/IEC 9899:1999] Section 6.4.5, "String literals"

[Summit 95] comp.lang.c FAQ list - Question 1.32

[Plum 91] Topic 1.26, "strings - string literals"

[Seacord 09] "STR30-C. Do not attempt to modify string literals"

7.34 Pass only unsigned char to character handling functions [STR037]

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

Noncompliant Code Example

In this noncompliant code example, `isspace` may be passed values that are not representable as `unsigned char`.

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

  /* possibly *t < 0 */
  while (isspace(*t) && (t - s < length)) {
    ++t;
  }
  return t - s;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.4, "Character handling <ctype.h>"

[Kettlewell 02] Section 1.1, "<ctype.h> And Characters Types"

[MITRE 07] CWE ID 704, "Incorrect Type Conversion or Cast," CWE ID 686, "Function Call With Incorrect Argument Type"

[Seacord 09] "STR37-C. Arguments to character handling functions must be representable as an `unsigned char`"

7.35 Do not perform zero length allocations [MEM004]

Allocating zero bytes of memory shall be diagnosed (subject to exceptions below) because this results in implementation-defined behavior.

Noncompliant Code Example

In this noncompliant code example, an allocation of zero bytes will occur if `size` is zero.

```
size_t size;

/* initialize size, possibly by user-controlled input */

int *list = (int *)malloc(size);
if (list == NULL) {
  /* Handle allocation error */
}
else {
  /* Continue processing list */
}
```

Noncompliant Code Example

In this noncompliant code example, an allocation of zero bytes will occur if `nsize` is zero.

```
size_t nsize = /* some value, possibly user supplied */;
char *p2;
char *p = (char *)malloc(100);
if (p == NULL) {
  /* Handle error */
}

/* ... */

if ((p2 = (char *)realloc(p, nsize)) == NULL) {
  free(p);
  p = NULL;
  return NULL;
}
p = p2;
```

MEM004-EX1: Some library implementations accept and ignore an allocation of zero. If all libraries used by a project have been validated to this behavior, then this violation need not be diagnosed.

Bibliography

[ISO/IEC 9899:1999] Section 7.20.3, "Memory Management Functions"

[MITRE 07] CWE ID 687, "Function Call With Incorrectly Specified Argument Value"

[Seacord 05] Chapter 4, "Dynamic Memory Management"

[Seacord 09] "MEM04-C. Do not perform zero length allocations"

7.36 Only pass arguments to calloc that, when multiplied, fit in size_t [MEM007]

Arguments supplied to `calloc` that cannot be represented as `size_t` when multiplied shall be diagnosed (subject to exceptions below) because this results in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the result of the multiplication of `num_elements` and `sizeof(long)` may not be representable as a `size_t`.


```
size_t num_elements;

long *buffer = (long *)calloc(num_elements, sizeof(long));
if (buffer == NULL) {
  /* Handle error condition */
}
/*...*/
free(buffer);
buffer = NULL;
```

MEM007-EX1: Some library implementations cause a runtime-constraint violation if the arguments to `calloc`, when multiplied, exceed the range of `size_t`. If all libraries used by a project have been validated to this behavior, then this violation need not be diagnosed.

Bibliography

[ISO/IEC 9899:1999] Section 7.18.3, "Limits of other integer types"

[MITRE 07] CWE ID 190, "Integer Overflow (Wrap or Wraparound)," and CWE ID 128, "Wrap-around Error"

[RUS-CERT] Advisory 2002-08:02, "Flaw in `calloc` and similar routines"

[Seacord 05] Chapter 4, "Dynamic Memory Management"

[Seacord 09] "MEM07-C. Ensure that the arguments to `calloc`, when multiplied, can be represented as a `size_t`"

[Secunia] Advisory SA10635, "HP-UX calloc Buffer Size Miscalculation Vulnerability"

7.37 Use realloc only to resize dynamically allocated arrays [MEM008]

Using `realloc` on a pointer that refers to memory that was not dynamically allocated shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, `realloc` is used to double the size of an array that was not dynamically allocated.

```
#include <stdlib.h>
char buf[BUFSIZE];

/* initialize buf, do not allocate memory dynamically */

buf = (char *)realloc(buf, 2 * BUFSIZE);
/* ... */
```

Bibliography

[ISO/IEC 9899:1999] Section 7.20.3.4, "The `realloc` function"

[ISO/IEC PDTR 24772] "AMV Type-breaking reinterpretation of data"

[MITRE 07] CWE ID 628, "Function Call with Incorrectly Specified Arguments"

[Seacord 09] "MEM08-C. Use `realloc` only to resize dynamically allocated arrays"

7.38 Only free memory allocated dynamically [MEM034]

Freeing memory that was not allocated dynamically shall be diagnosed (subject to exceptions below) because doing so results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the pointer passed to `free` could refer to a string literal.

```
enum { MAX_ALLOCATION = 1000 };

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

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

MEM034-EX1: Some library implementations accept and ignore a deallocation of non-allocated memory (or alternatively cause a runtime-constraint violation). If all libraries used by a project have been validated to this behavior, then this violation need not be diagnosed.

Bibliography

[ISO/IEC 9899:1999] Section 7.20.3, "Memory management functions"

[MITRE 07] CWE ID 590, "Free of Invalid Pointer Not on the Heap"

[Seacord 05] Chapter 4, "Dynamic Memory Management"

[Seacord 09] "MEM34-C. Only free memory allocated dynamically"

7.39 Do not perform operations on devices that are only appropriate for files [FIO032]

Performing file operations on special devices shall be diagnosed because this results in implementation-defined behavior.

Noncompliant Code Example

In this noncompliant code example, the user can specify a locked device or a FIFO file name, causing the program to hang on the call to `fopen`.


```
char *file_name;
FILE *file;

/* initialize file_name */

if (!fgets(file_name, sizeof(file_name), stdin)) {
  /* Handle error */
}

if ((file = fopen(file_name, "wb")) == NULL) {
  /* Handle error */
}

/* operate on file */

fclose(file);
```

Bibliography

[Garfinkel 96] Section 5.6, "Device Files"

[Howard 02] Chapter 11, "Canonical Representation Issues"

[MITRE 07] CWE ID 67, "Failure to Handle Windows Device Names"

[ISO/IEC 9899:1999] Section 7.19.4, "Operations on Files"

[Open Group 04] `open`

[Seacord 09] "FIO32-C. Do not perform operations on devices that are only appropriate for files"

7.40 Only register atexit handlers that return normally [ENV032]

Using an `atexit`-registered handler that terminates in a way other than by returning shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the `atexit`-registered handler `exit2` calls `exit`.

```
#include <stdio.h>
#include <stdlib.h>

void exit1(void) {
  /* ...cleanup code... */
  return;
}

void exit2(void) {
  if (/* condition */) {
    /* ...more cleanup code... */
    exit(0);
  }
  return;
}

int main(void) {
  if (atexit(exit1) != 0) {
    /* Handle error */
```

```
  }
  if (atexit(exit2) != 0) {
    /* Handle error */
  }
  /* ...program code... */
  exit(0);
}
```

Noncompliant Code Example

In this noncompliant code example, `longjmp` is used to terminate the `atexit`-registered handler `exit1`.

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env;
int val;

void exit1(void) {
  /* ... */
  longjmp(env, 1);
}

int main(void) {
  if (atexit(exit1) != 0) {
    /* Handle error */
  }
  /* ... */
  if (setjmp(env) == 0) {
    exit(0);
  }
  else {
    return 0;
  }
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.20.4.3, "The `exit` function"

[ISO/IEC PDTR 24772] "EWD Structured Programming" and "REU Termination Strategy"

[MITRE 07] CWE ID 705, "Incorrect Control Flow Scoping"

[Seacord 09] "ENV32-C. All `atexit` handlers must return normally"

7.41 Do not call non-asynchronous-safe functions from signal handlers [SIG030]

Calling a non-asynchronous-safe function from within a signal handler shall be diagnosed because doing so results in undefined behavior. The only asynchronous-safe functions in the C standard library are `abort`, `_Exit`, and `signal`.

Noncompliant Code Example

In this noncompliant code example, the signal handler `handler` calls the non-asynchronous-safe function `log_message`.

```
#include <signal.h>
```

```
#include <stdio.h>
#include <stdlib.h>

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

void log_message(void) {
  fprintf(stderr, info);
}

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

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

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

    log_message();

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

Bibliography

[Dowd 06] Chapter 13, "Synchronization and State"

[ISO/IEC 03] Section 5.2.3, "Signals and interrupts"

[ISO/IEC 9899:1999] Section 7.14, "Signal handling `<signal.h>`"

[MITRE 07] CWE ID 479, "Unsafe Function Call from a Signal Handler"

[Open Group 04] `longjmp`

[OpenBSD] `signal` Man Page

[Seacord 09] "SIG30-C. Call only asynchronous-safe functions within signal handlers"

[Zalewski 01]

7.42 Do not call raise from a signal handler [SIG033]

A signal handler that calls the `raise` function shall be diagnosed (subject to exceptions below) because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the signal handler `handler` calls `raise` and is invoked by `raise`.

```
#include <signal.h>

void log_msg(int signum) {
```

```
  /* Log error message in some asynchronous-safe manner */
}

void handler(int signum) {
  /* SIGINT handling specific */
  if (raise(SIGUSR1) != 0) {
    /* Handle error */
  }
}

int main(void) {
  if (signal(SIGUSR1, log_msg) == SIG_ERR) {
    /* Handle error */
  }
  if (signal(SIGINT, handler) == SIG_ERR) {
    /* Handle error */
  }

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

  return 0;
}
```

Exceptions

SIG033-EX1: A signal handler that does not occur as a result of calling the `abort` or `raise` function need not be diagnosed because this does not result in undefined behavior.

Bibliography

[Dowd 06] Chapter 13, "Synchronization and State"

[ISO/IEC 9899:1999] Section 7.14.1.1, "The `signal` function"

[MITRE 07] CWE ID 479, "Unsafe Function Call from a Signal Handler"

[Open Group 04]

[OpenBSD] `signal` Man Page

[Seacord 09] "SIG33-C. Do not recursively invoke the `raise` function"

7.43 Do not access or modify shared objects in signal handlers [SIG031]

Accessing or modifying a shared object not of type `volatile sig_atomic_t` in a signal handler shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the shared pointer `err_msg` is modified from the signal handler `handler`.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>
```



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

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

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

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

  /* Main code loop */

  return 0;
}
```

Bibliography

[Dowd 06] Chapter 13, Synchronization and State

[ISO/IEC 03] "Signals and Interrupts"

[MITRE 07] CWE ID 662, "Insufficient Synchronization"

[Open Group 04] `longjmp`

[OpenBSD] `signal` Man Page

[Seacord 09] "SIG31-C. Do not access or modify shared objects in signal handlers"

[Zalewski 01]

7.44 Do not redefine errno [ERR031]

Redefining `errno` shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, `errno` is redefined.

```
extern int errno;
```

Bibliography

[ISO/IEC 9899:1999] Section 7.5, "Errors `<errno.h>`"

[Seacord 09] "ERR31-C. Don't redefine `errno`"

7.45 Invoke functions using the correct type [DCL035]

Invoking a function using a type that does not match the function definition shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the function pointer `new_function` refers to a function that accepts a single argument and returns `void`, which is incompatible with the declared type of the pointer. A subsequent call through the pointer results in undefined behavior.

```
static void my_function(int a) {
  /* ... */
  return;
}

int main(void) {
  int x;
  int (*new_function)(int a) = my_function;
  x = (*new_function)(10);
  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.3.2.3, "Pointers"

[ISO/IEC PDTR 24772] "IHN Type system" and "OTR Subprogram Signature Mismatch"

[MITRE 07] CWE ID 686, "Function Call With Incorrect Argument Type"

[Seacord 09] "DCL35-C. Do not invoke a function using a type that does not match the function definition"

7.46 Do not use incompatible array types in an expression [ARR034]

Using incompatible array types in an expression shall be diagnosed because this results in undefined behavior. For two array types to be compatible, both should have compatible underlying element types, and both size specifiers should have the same constant value.

Noncompliant Code Example

In this noncompliant code example, `arr1` is assigned to an incompatible array `arr2`.

```
enum { a = 10, b = 15, c = 20 };

int arr1[c][b];
int (*arr2)[a];

arr2 = arr1; /* Not compatible because a != b */
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.5.2, "Array declarators"

[MITRE 07] CWE ID 119, "Failure to Constrain Operations within the Bounds of an Allocated Memory Buffer"

[Seacord 09] "ARR34-C. Ensure that array types in expressions are compatible"

7.47 Do not subtract or compare two pointers that do not refer to the same array [ARR036]

Subtracting or comparing two pointers that do not refer to the same array object or one element past the same array object shall be diagnosed (subject to exceptions below) because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the pointers `strings` and `(char **)next_num_ptr` do not refer to the same array.

```
int nums[SIZE];
char *strings[SIZE];
int *next_num_ptr = nums;
int free_bytes;

/* increment next_num_ptr as array fills */

free_bytes = strings - (char **)next_num_ptr;
```

Exceptions

ARR036-EX1: Comparing two pointers within the same structure need not be diagnosed because this does not result in undefined behavior.

ARR036-EX2: Subtracting two pointers to `char` within the same structure need not be diagnosed because this does not result in undefined behavior.

Bibliography

[Banahan 03] Section 5.3, "Pointers," and Section 5.7, "Expressions involving pointers"

[ISO/IEC 9899:1999] Section 6.5.6, "Additive operators"

[MITRE 07] CWE ID 469, "Use of Pointer Subtraction to Determine Size"

[Seacord 09] "ARR36-C. Do not subtract or compare two pointers that do not refer to the same array"

7.48 Do not modify the string returned by getenv [ENV030]

Modifying the string returned by `getenv` shall be diagnosed because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the string returned by `getenv` is modified.

```
void strtr(char *str, char orig, char rep) {
  while (*str != '\0') {
    if (*str == orig) {
      *str = rep;
    }
    str++;
  }
}

/* ... */

char *env = getenv("TEST_ENV");
if (env == NULL) {
  /* Handle error */
}

strtr(env,'"', '_');

/* ... */
```

Bibliography

[ISO/IEC 9899:1999] Section 7.20.4.5, "The `getenv` function"

[Open Group 04] getenv

[Seacord 09] "ENV30-C. Do not modify the string returned by getenv"

7.49 Free dynamically allocated memory exactly once [MEM031]

Freeing memory multiple times shall be diagnosed (subject to exceptions below) because this results in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, `x` could be freed twice depending on the value of `error_condition`.

```
size_t num_elem = /* some initial value */;
int error_condition = 0;

int *x = (int *)malloc(num_elem * sizeof(int));
if (x == NULL) {
  /* handle allocation error */
}
/* ... */
if (error_condition == 1) {
  /* handle error condition*/
  free(x);
  x = NULL;
}
/* ... */
free(x);
x = NULL;
```

MEM031-EX1: Some library implementations accept and ignore a deallocation of already-free memory. If all libraries used by a project have been validated to this behavior, then this violation need not be diagnosed.

Bibliography

[ISO/IEC PDTR 24772] "XYK Dangling Reference to Heap" and "XYL Memory Leak"

[MIT 05]

[MITRE 07] CWE ID 415, "Double Free"

[OWASP, Double Free]

[Seacord 09] "MEM31-C. Free dynamically allocated memory exactly once"

[Viega 05] "Doubly freeing memory"

[VU#623332]

8 Indirect Result

8.1 Do not let unsigned integer operations wrap [INT030]

Unsigned integer wrapping shall be diagnosed because this can result in unexpected behavior.

The following table indicates which operators can result in wrapping.

Operator	Wrap	Operator	Wrap	Operator	Wrap	Operator	Wrap
+	yes	-=	yes	<<	yes	<	no
-	yes	*=	yes	>>	no	>	no
*	yes	/=	no	&	no	>=	no
/	no	%=	no	|	no	<=	no
%	no	<<=	yes	^	no	==	no
++	yes	>>=	no	~	no	!=	no
--	yes	&=	no	!	no	&&	no
=	no	|=	no	un +	no	||	no
+=	yes	^=	no	un -	yes	?:	no

Noncompliant Code Example

In this noncompliant code example, all of the arithmetic operations can result in unsigned integer wrapping.

```
unsigned int ui1, ui2, result;

/* Initialize ui1 and ui2 */
result = ui1 + ui2;
result = ui1 - ui2;
result = ui1 * ui2;
result = ui1 << ui2;
result = -ui1;
```

Bibliography

[Dowd 06] Chapter 6, "C Language Issues" (Arithmetic Boundary Conditions, pp. 211-223)

[ISO/IEC 9899:1999] Section 6.2.5, "Types," Section 6.5, "Expressions," and Section 7.10, "Sizes of integer types <limits.h>"

[ISO/IEC PDTR 24772] "XYY Wrap-around Error"

[MITRE 07] CWE ID 190, "Integer Overflow (Wrap or Wraparound)"

[Seacord 05] Chapter 5, "Integers"

[Seacord 09] "INT30-C. Ensure that unsigned integer operations do not wrap"

[Viega 05] Section 5.2.7, "Integer overflow"

[VU#551436]

[Warren 02] Chapter 2, "Basics"

[Wojtczuk 08]

8.2 Do not use a plain int bit-field [INT012]

Specifying a bit-field as type `int` shall be diagnosed because it is implementation-defined whether a bit-field of type `int` is the same as a bit-field of type `signed int` or `unsigned int`.

Noncompliant Code Example

In this noncompliant code example, the bit-field `a` has type `int`.

```
struct {
  int a: 8;
} bits = {255};

int main(void) {
  printf("bits.a = %d.\n", bits.a);
  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.2, "Type specifiers"

[ISO/IEC PDTR 24772] "STR Bit Representations"

[MISRA 04] Rule 12.7

[Seacord 09] "INT12-C. Do not make assumptions about the type of a plain `int` bit-field when used in an expression"

8.3 Do not hard code the size of a type [EXP009]

Using hard-coded values to represent the size of a type shall be diagnosed because type sizes are implementation-defined.

Noncompliant Code Example

In this noncompliant code example, the number 4 is used to allocate memory for an array of pointers and arrays of integers.

```
/* assuming 32-bit pointer, 32-bit integer */
size_t i;
int **matrix = (int **)calloc(100, 4);
if (matrix == NULL) {
  /* handle error */
}

for (i = 0; i < 100; i++) {
```



```
  matrix[i] = (int *)calloc(i, 4);
  if (matrix[i] == NULL) {
    /* handle error */
  }
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.2.6, "Representations of types," and Section 6.5.3.4, "The `sizeof` operator"

[Seacord 09] "EXP09-C. Use `sizeof` to determine the size of a type or variable"

8.4 Do not shift signed types [INT013]

Shifting a signed type shall be diagnosed because this can result in undefined or implementation-defined behavior.

If a signed type is left-shifted, and the value is negative or the result cannot be represented, then this results in undefined behavior.

A signed type that has negative value and is right-shifted results in an implementation-defined value.

Noncompliant Code Example

In this noncompliant code example, the result of `stringify >> 24` is an implementation-defined value.

```
int rc = 0;
int stringify = 0x80000000;
char buf[sizeof("256")];
rc = snprintf(buf, sizeof(buf), "%u", stringify >> 24);
if (rc == -1 || rc >= sizeof(buf)) {
  /* handle error */
}
```

Bibliography

[Dowd 06] Chapter 6, "C Language Issues"

[ISO/IEC 03] Section 6.5.7, "Bitwise shift operators"

[ISO/IEC 9899:1999] Section 6.5.7, "Bitwise shift operators"

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

[MITRE 07] CWE ID 682, "Incorrect Calculation"

[Seacord 09] "INT13-C. Use bitwise operators only on unsigned operands"

8.5 Use intmax_t or uintmax_t for formatted IO on defined integer types [INT015]

Using a formatted IO function with a programmer-defined integer type as an argument and a format string whose matching specifier does not have the length modifier `j` (which indicates `intmax_t` or `uintmax_t`) shall be diagnosed because this can result in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the value of `x` is printed as if it were an `unsigned long long` value.

```
#include <stdio.h>

mytypedef_t x;
/* ... */
printf("%llu", (unsigned long long) x);
```

Noncompliant Code Example

In this noncompliant code example, `scanf` is used to read an `unsigned long long` value into `x`, which has type `mytypedef_t`.

```
#include <stdio.h>

mytypedef_t x;
/* ... */
if (scanf("%llu", &x) != 1) {
  /* handle error */
}
```

Bibliography

[ISO/IEC 9899-1999] Section 7.18.1.5, "Greatest-width integer types," and Section 7.19.6, "Formatted input/output functions"

[MITRE 07] CWE ID 681, "Incorrect Conversion between Numeric Types"

[Seacord 09] "INT15-C. Use `intmax_t` or `uintmax_t` for formatted IO on programmer-defined integer types"

8.6 Do not use floating point variables as loop counters [FLP030]

Using a floating point variable as a loop counter shall be diagnosed because floating point numbers have precision limitations. Code that relies on floating point loop counters can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, a floating-point variable is used as a loop counter.

```
for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
  /* ... */
}
```

Bibliography

[ISO/IEC 14882:2003] Sections 2.13.3, "Floating literals," and 3.9.1, "Fundamental types"

[ISO/IEC PDTR 24772] "PLF Floating Point Arithmetic"

[Lockheed Martin 05] AV Rule 197, "Floating point variables shall not be used as loop counters"

[MISRA 04] Rules 13.3 and 13.4

[Seacord 09] "FLP30-C. Do not use floating point variables as loop counters"

8.7 Do not reuse variable names in subscopes [DCL001]

A variable that is in the subscope of and shares its name with another variable shall be diagnosed because reusing variable names can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the identifier `msg` is used at two scopes.

```
char msg[100];

void report_error(const char *error_msg) {
  char msg[80];
  /* ... */
  strncpy(msg, error_msg, sizeof(msg));
  return;
}

int main(void) {
  char error_msg[80];
  /* ... */
  report_error(error_msg);
  /* ... */
}
```

Bibliography

[ISO/IEC 9899:1999] Section 5.2.4.1, "Translation limits"

[MISRA 04] Rule 5.2

[Seacord 09] "DCL01-C. Do not reuse variable names in subscopes"

8.8 Include the appropriate type information in function declarators [DCL007]

A function declarator that does not have the same number of parameters, parameter types, and return type as the function definition shall be diagnosed because this can result in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the definition of `func` in `file_a.c` expects three parameters but is supplied only two in `file_b.c`.

```
/* file_a.c source file */
int func(int one, int two, int three) {
  printf("%d %d %d", one, two, three);
  return 1;
}
```

However, because there is no prototype for `func` in `file_b.c`, a compiler coulde assume that the correct number of arguments have been supplied and use the next value on the program stack as the missing third argument.

```
/* file_b.c source file */
func(1, 2);
```

Noncompliant Code Example

In this noncompliant code example, the function pointer `fn_ptr` refers to the function `add`, which accepts three integer arguments. However, `fn_ptr` is specified to accept two integer arguments, resulting in undefined behavior.

```
int add(int x, int y, int z) {
   return x + y + z;
```

```
}

int main(int argc, char *argv[]) {
   int (*fn_ptr) (int, int);
   int res;
   fn_ptr = add;
   res = fn_ptr(2, 3);  /* incorrect */
   /* ... */
   return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Forward and Section 6.9.1, "Function definitions"

[ISO/IEC PDTR 24772] "IHN Type system" and "OTR Subprogram Signature Mismatch"

[MISRA 04] Rule 8.2

[Seacord 09] "DCL07-C. Include the appropriate type information in function declarators"

[Spinellis 06] Section 2.6.1, "Incorrect Routine or Arguments"

8.9 Operands to the sizeof operator should not contain side effects [EXP006]

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

Noncompliant Code Example

In this noncompliant code example, the expression `a++` is not evaluated and the side effects in the expression are not executed.

```
int a = 14;
int b = sizeof(a++);
```

Bibliography

[ISO/IEC 9899:1999] Section 6.5.3.4, "The sizeof operator"

[Seacord 09] "EXP06-C. Operands to the sizeof operator should not contain side effects"

8.10 Use rsize_t or size_t for all integer values representing the size of an object [INT001]

Using a type other than `rsize_t` or `size_t` for an integer value representing the size of an object shall be diagnosed because this can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the variable `i` of type `int`. If `n > INT_MAX` then `i` will overflow.

```
char *copy(size_t n, const char *str) {
  int i;
  char *p;

  if (n == 0) {
    /* Handle unreasonable object size error */
```



```
  }
  p = (char *)malloc(n);
  if (p == NULL) {
    /* Handle malloc failure */
  }
  for ( i = 0; i < n; ++i ) {
    p[i] = *str++;
  }
  return p;
}

/* ... */

char str[] = "hi there";
char *p = copy(sizeof(str), str);
```

Noncompliant Code Example

In this noncompliant code example, `length` is of type `unsigned long`. If `sizeof(unsigned long) > sizeof(unsigned int)` then the value stored in `length` may be truncated when passed as an argument to `alloc`.

```
void *alloc(unsigned int blocksize) {
  return malloc(blocksize);
}

int read_counted_string(int fd) {
  unsigned long length;
  unsigned char *data;

  if (read_integer_from_network(fd, &length) < 0) {
    return -1;
  }

  data = (unsigned char*)alloc(length);

  if (read_network_data(fd, data, length) < 0) {
    free(data);
    return -1;
  }
  data[length] = '\0';

  /* ... */
  free( data);
  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.17, "Common definitions `<stddef.h>`", Section 7.20.3, "Memory management functions"

[ISO/IEC TR 24731-1:2007]

[Seacord 09] "INT01-C. Use `rsize_t` or `size_t` for all integer values representing the size of an object"

8.11 Use only explicitly signed or unsigned char type for numeric values [INT007]

Using the `char` type that is not qualified as `signed` or `unsigned` to store numeric values shall be diagnosed (subject to exceptions below) because the `char` type is incompatible with the `signed char` and `unsigned char` types and because the `char` type is implementation-defined.

Noncompliant Code Example

In this noncompliant code example, the variable `c` has type `char`, resulting in implementation-defined behavior.

```
char c = 200;
int i = 1000;
printf("i/c = %d\n", i/c);
```

Exceptions

INT007-EX1: Storing the result of a character IO function that returns `int` in a variable of type `char` need not be diagnosed because such a value is not numeric.

Bibliography

[ISO/IEC 9899:1999] Section 6.2.5, "Types"

[ISO/IEC PDTR 24772] "STR Bit Representations"

[MISRA 04] Rule 6.2, "`Signed` and `unsigned char` type shall be used only for the storage and use of numeric values"

[MITRE 07] CWE ID 682, "Incorrect Calculation"

[Seacord 09] "INT07-C. Use only explicitly `signed` or `unsigned char` type for numeric values"

8.12 Do not use enumeration constants that map to nonunique values [INT009]

Declaring enumeration constants that map to nonunique values shall be diagnosed because their usage can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, several enumeration type members have the same value.

```
enum {red=4, orange, yellow, green, blue, indigo=6, violet};
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.2.2, "Enumeration specifiers"

[ISO/IEC PDTR 24772] "CCB Enumerator issues"

[MISRA 04] Rule 9.3

[Seacord 09] "INT09-C. Ensure enumeration constants map to unique values"

8.13 Do not place a semicolon on the same line as an if, for, or while statement [EXP017]

Using a semicolon on the same line as an `if`, `for`, or `while` statement shall be diagnosed because this typically indicates programmer error and can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, a semicolon is used on the same line as an `if` statement.

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

Bibliography

[Hatton 95] Section 2.7.2, "Errors of omission and addition"

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

[MITRE 07] CWE ID 480, "Use of Incorrect Operator"

[Seacord 09] "MSC03-C. Avoid errors of addition"

8.14 Do not compare function pointers to constant values [EXP018]

Comparing a function pointer to a value that is not a null function pointer of the same type shall be diagnosed because this typically indicates programmer error and can result in unexpected behavior. Implicit comparisons shall be diagnosed as well.

Noncompliant Code Example

In this noncompliant code example, the function pointers `getuid` and `geteuid` are compared to 0.

```
/* First the options that are only allowed for root */
if (getuid == 0 || geteuid != 0) {
  /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, the function pointer `do_xyz` is implicitly compared unequal to 0.

```
int do_xyz(void);

if (do_xyz) {
  /* handle error */
}
```

Bibliography

[Hatton 95] Section 2.7.2, "Errors of omission and addition"

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

[Seacord 09] "MSC02-C. Avoid errors of omission"

8.15 Do not perform bitwise operations in conditional expressions [EXP016]

Using the bitwise AND (`&`, ampersand) or bitwise OR (`|`, pipe) operator in a conditional expression shall be diagnosed because this typically indicates programmer error and can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, a bitwise expression is used in a conditional expression.

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

Bibliography

[Hatton 95] Section 2.7.2, "Errors of omission and addition"

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

[Seacord 09] "MSC02-C. Avoid errors of omission"

8.16 Do not perform assignments in conditional expressions [EXP015]

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

Noncompliant Code Example

In this noncompliant code example, an assignment expression is the outermost expression in a conditional expression.

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

Bibliography

[Hatton 95] Section 2.7.2, "Errors of omission and addition"

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

[MITRE 07] CWE ID 482, "Comparing instead of Assigning," CWE ID 480, "Use of Incorrect Operator"

[Seacord 09] "MSC02-C. Avoid errors of omission"

8.17 Check all possible data paths in an if-chain or switch statement [MSC001]

Not checking all possible data paths in an `if`-chain or `switch` statement shall be diagnosed (subject to exceptions below) because doing so can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, there is no test for the condition where `a` is neither `b` nor `c`.

```
if (a == b) {
  /* ... */
}
else if (a == c) {
  /* ... */
}
```


Noncompliant Code Example

In this noncompliant code example, the switch statement does not cover all possible values of `widget_type`.

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

widget_type = 45;

switch (widget_type) {
  case WE_X:
    /* ... */
    break;
  case WE_Y:
    /* ... */
    break;
  case WE_Z:
    /* ... */
    break;
}
```

Exceptions

MSC001-EX1: A `if` statement by itself shall not be diagnosed because this is a common idiom.

Bibliography

[Hatton 95] Section 2.7.2, "Errors of omission and addition"

[ISO/IEC PDTR 24772] "CLL Switch statements and static analysis"

[Seacord 09] "MSC01-C. Strive for logical completeness"

[Viega 05] Section 5.2.17, "Failure to account for default case in switch"

8.18 Do not use trigraphs [PRE007]

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

Noncompliant Code Example

In this noncompliant code example, `a++` is not executed because the trigraph sequence `??/` is replaced by `\`, logically putting `a++` on the same line as the comment.

```
// what is the value of a now??/
a++;
```

Noncompliant Code Example

In this noncompliant code example, the trigraph sequence `??!` is replaced by the character `|`, which is clearly not intended.

```
size_t i = /* some initial value */;
if (i > 9000) {
   if (puts("Over 9000!??!") == EOF) {
     /* Handle Error */
   }
}
```

Bibliography

[ISO/IEC 9899:1999] Section 5.2.1.1, "Trigraph sequences"

[MISRA 04] Rule 4.2

[Seacord 09] "PRE07-C. Avoid using repeated question marks"

8.19 Declare identifiers before using them [DCL031]

Undeclared identifiers, including implicitly-typed variables and functions, shall be diagnosed because their use can result in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the type specifier is omitted from the declaration of `foo`, resulting in `foo` having type `int`.

```
extern foo;
```

Noncompliant Code Example

In this noncompliant code example, `foo` is not prototyped before it is invoked in `main`.

```
int main(void) {
  int c = foo();
  printf("%d\n", c);
  return 0;
}

int foo(int a) {
  return a;
}
```

Noncompliant Code Example

In this noncompliant code example, the return type of `foo` is implicitly declared to be `int`.

```
foo(void) {
  return UINT_MAX;
}

int main(void) {
  long long c = foo();
  printf("%lld\n", c);
  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.2, "Type specifiers", Section 6.5.2.2, "Function calls"

[ISO/IEC PDTR 24772] "OTR Subprogram Signature Mismatch"

[Jones 08]

[MISRA 04]

[Seacord 09] "DCL31-C. Declare identifiers before using them"

8.20 Use only unique mutually visible identifiers [DCL032]

Using nonunique identifiers in mutually visible scopes shall be diagnosed (subject to exceptions below) because this can result in unexpected behavior.

Section 5.2.4.1 of the C standard defines the following minimum requirements for uniqueness.

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

Noncompliant Code Example

In this noncompliant code example, the two identifiers are not unique because the first 31 characters are identical.

```
extern int *global_symbol_definition_lookup_table_a;
extern int *global_symbol_definition_lookup_table_b;
```

Noncompliant Code Example

In this noncompliant code example, both external identifiers consist of four universal character names. Because the first three universal character names of each identifier are identical, both identify the same integer array.

```
extern int *\U00010401\U00010401\U00010401\U00010401;
extern int *\U00010401\U00010401\U00010401\U00010402;
```

Exceptions

DCL032-EX1: Code written for implementations that support longer restrictions need not be diagnosed.

Bibliography

[ISO/IEC 9899:1999] Section 5.2.4.1, "Translation limits"

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

[MISRA 04] Rules 5.1 and 8.9

[Seacord 09] "DCL32-C. Guarantee that mutually visible identifiers are unique"

8.21 Do not perform byte-by-byte comparisons between structures [EXP004]

Performing a byte-by-byte comparison between structures shall be diagnosed because structure padding is implementation-defined.

Noncompliant Code Example

In this noncompliant code example, `memcmp` is used to compare two structures.

```
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))) {
    return 1;
  }
  return 0;
}
```

Bibliography

[Dowd 06] Chapter 6, "C Language Issues" (Structure Padding 284-287)

[ISO/IEC 9899:1999] Section 6.7.2.1, "Structure and union specifiers"

[Kerrighan 88] Chapter 6, "Structures" (Structures and Functions 129)

[Seacord 09] "EXP04-C. Do not perform byte-by-byte comparisons between structures"

[Summit 95] Question 2.8, Question 2.12

8.22 Avoid domain and range errors in math functions [FLP032]

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

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

Function	Domain	Range Error
`acos(x), asin(x)`	`-1 <= x && x <= 1`	no
`atan2(y, x)`	`x != 0 || y != 0`	no
`acosh(x)`	`x >= 1`	no
`atanh(x)`	`-1 < x && x < 1`	no
`cosh(x), sinh(x)`	none	yes
`exp(x), exp2(x), expm1(x)`	none	yes

`ldexp(x, exp)`	none	yes
`log(x), log10(x), log2(x)`	`x > 0`	no
`log1p(x)`	`x > -1`	no
`ilogb(x), logb(x)`	`x != 0`	yes
`scalbn(x, n), scalbln(x, n)`	none	yes
`hypot(x, y)`	none	yes
`pow(x, y)`	`x > 0 || (x == 0 && y > 0) || (x < 0 && y` is an integer)	yes
`sqrt(x)`	`x >= 0`	no
`erfc(x)`	none	yes
`lgamma(x), tgamma(x)`	`x != 0 && !(x < 0 && x` is an integer)	yes
`lrint(x), lround(x)`	none	yes
`fmod(x, y)`	`y != 0`	no
`nextafter(x, y), nexttoward(x, y)`	none	yes
`fdim(x, y)`	none	yes
`fma(x, y, z)`	none	yes

Noncompliant Code Example

In this noncompliant code example, `x` may not be in the domain of the `sqrt` function.

```
double x;
double result;

result = sqrt(x);
```

Noncompliant Code Example

In this noncompliant code example, the result of `cosh(x)` is not checked for a range error.

```
double x;
double result;

result = cosh(x);
/* rest of program that does not check for range error */
```

Noncompliant Code Example

In this noncompliant code example, `x` and `y` may not be in the domain of the `pow` function, and the result of `pow(x, y)` is not checked for a range error.

```
double x;
double y;
double result;

result = pow(x, y);
/* rest of program that does not check for range error */
```

Bibliography

[ISO/IEC 9899:1999] Section 7.3, "Complex arithmetic <complex.h>", and Section 7.12, "Mathematics <math.h>"

[MITRE 07] CWE ID 682, "Incorrect Calculation"

[Plum 85] Rule 2-2

[Plum 89] Topic 2.10, "conv - conversions and overflow"

[Seacord 09] "FLP32-C. Prevent or detect domain and range errors in math functions"

8.23 Convert integers to floating point for floating point operations [FLP033]

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

Noncompliant Code Example

In this noncompliant code example, the division and multiplication operations take place on integers and are then converted to floating point.

```
short a = 533;
int b = 6789;
long c = 466438237;

float d = a / 7; /* d is 76.0 */
double e = b / 30; /* e is 226.0 */
double f = c * 789; /*  f may be negative due to overflow */
```

Bibliography

[Hatton 95] Section 2.7.3, "Floating-point misbehavior"

[ISO/IEC 9899:1999] Section 5.2.4.2.2, "Characteristics of floating types <float.h>"

[MITRE 07] CWE ID 681, "Incorrect Conversion between Numeric Types," and CWE ID 682, "Incorrect Calculation"

[Seacord 09] "FLP33-C. Convert integers to floating point for floating point operations"

8.24 Do not declare a variable length array with an untrusted value [ARR032]

Using a value that is not within a trusted range to declare a variable length array shall be diagnosed because this can result in undefined behavior.

Noncompliant Code Example

In this noncompliant code example, the variable `s` used to declare the variable length array `vla` cannot be trusted.

```
void func(size_t s) {
  int vla[s];
  /* ... */
}
/* ... */
func(size);
/* ... */
```

Bibliography

[Griffiths 06]

[ISO/IEC PDTR 24772] "XYX Boundary Beginning Violation" and "XYZ Unchecked Array Indexing"

[Seacord 09] "ARR32-C. Ensure size arguments for variable length arrays are in a valid range"

8.25 Cast characters to unsigned types before converting to larger integer sizes [STR034]

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

Noncompliant Code Example

In this noncompliant code example, the character of type `char` pointed to by `string` is converted to `int` without being cast to `unsigned char` first. On a platform where char is represented as `signed char`, this code results in unexpected behavior if `string` points to character code 255 since the value of `c` would be -1 (`EOF`).

```
static int yy_string_get {

  register char *string;
  register int c;

  string = bash_input.location.string;
  c = EOF;

  /* If the string doesn't exist, or is empty, EOF found. */
  if (string && *string) {
    c = *string++;
    bash_input.location.string = string;
  }
  return (c);
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.2.5, "Types"

[MISRA 04] Rule 6.1, "The plain `char` type shall be used only for the storage and use of character values."

[MITRE 07] CWE ID 704, "Incorrect Type Conversion or Cast"

[Seacord 09] "STR34-C. Cast characters to unsigned types before converting to larger integer sizes"

8.26 Cast the result of memory allocation into a pointer to the allocated type [MEM002]

Not casting the result of a memory allocation function into a pointer to the allocated type shall be diagnosed because doing so can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the result from `malloc` is not cast to a pointer to type `gadget` (the intended type of `p`).

```
#include <stdlib.h>

typedef struct gadget gadget;
struct gadget {
  int i;
  double d;
};

typedef struct widget widget;
struct widget {
  char c[10];
  int i;
  double d;
};

widget *p;

/* ... */

p = malloc(sizeof(gadget)); /* imminent problem */
if (p != NULL) {
  p->i = 0;               /* undefined behavior */
  p->d = 0.0;             /* undefined behavior */
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.20.3, "Memory management functions"

[Summit 05] Question 7.7 and Question 7.7b

[Seacord 09] "MEM02-C. Immediately cast the result of a memory allocation function call into a pointer to the allocated type"

8.27 Use a variable of type int to capture the return value of character IO functions [FIO034]

Using a variable that has type other than `int` to capture the return value of a character IO function shall be diagnosed because character values may be indistinguishable from `EOF` otherwise.

Noncompliant Code Example

In this noncompliant code example, a variable of type `char` is used to capture the return value from `getchar`.

```
char buf[BUFSIZE];
char c;
int i = 0;

while ( (c = getchar()) != '\n' && c != EOF ) {
  if (i < BUFSIZE-1) {
```



```
    buf[i++] = c;
  }
}
buf[i] = '\0';
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.7, "Character input/output functions"

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

[NIST 06] SAMATE Reference Dataset Test Case ID 000-000-088

[Seacord 09] "FIO34-C. Use `int` to capture the return value of character IO functions"

8.28 Do not call getc or putc with stream arguments that have side effects [FIO040]

Invoking `getc` and `putc` with stream arguments that have side effects shall be diagnosed. If these functions are implemented as macros the stream argument may be evaluated multiple times.

Noncompliant Code Example

In this noncompliant code example, an expression with side effects is passed as the stream argument to `getc`. If `getc` is implemented as a macro, the file may be opened several times.

```
char *file_name;
FILE *fptr;

/* Initialize file_name */

int c = getc(fptr = fopen(file_name, "r"));
if (c == EOF) {
  /* Handle error */
}
```

Noncompliant Code Example

In this noncompliant code example, an expression with side effects is passed as the stream argument to `putc`.

```
char *file_name;
FILE *fptr = NULL;

/* Initialize file_name */

int c = 'a';
while (c <= 'z') {
  if (putc(c++, fptr ? fptr :
       (fptr = fopen(file_name, "w")) == EOF) {
    /* Handle error */
  }
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.7.5, "The `getc` function," and Section 7.19.7.8, "The `putc` function"

8.29 Do not store the pointer to the string returned by getenv [ENV000]

Storing the pointer to the string returned by `getenv` shall be diagnosed because it may be overwritten by a subsequent call to `getenv` or invalidated as a result of changes made to the environment.

Noncompliant Code Example

In this noncompliant code example, the pointer returned by `getenv` is stored for later use.

```
char *tmpvar;
char *tempvar;

tmpvar = getenv("TMP");
if (!tmpvar) return -1;
tempvar = getenv("TEMP");
if (!tempvar) return -1;

if (strcmp(tmpvar, tempvar) == 0) {
  if (puts("TMP and TEMP are the same.\n") == EOF) {
    /* Handle error */
  }
}
else {
  if (puts("TMP and TEMP are NOT the same.\n") == EOF) {
    /* Handle error */
  }
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.20.4, "Communication with the environment"

[ISO/IEC PDTR 24731-2]

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

[Open Group 04] Chapter 8, and "Environment Variables", `strdup`

[Seacord 09] "ENV00-C. Do not store the pointer to the string returned by `getenv`"

[Viega 03] Section 3.6, "Using Environment Variables Securely"

8.30 Do not call signal from interruptible signal handlers [SIG034]

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

Noncompliant Code Example

In this noncompliant code example, the signal handler `handler` calls `signal` on a platform where signal handlers are nonpersistent.

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



```
if (signal(SIGUSR1, handler) == SIG_ERR) {
  /* Handle error */
}
```

Bibliography

[ISO/IEC 9899-1999TR2] Section 7.14.1.1, "The `signal` function"

[MITRE 07] CWE ID 479, "Unsafe Function Call from a Signal Handler"

[Seacord 09] "SIG34-C. Do not call `signal` from within interruptible signal handlers"

8.31 Do not call longjmp from inside a signal handler [SIG032]

Calling the `longjmp` function from inside a signal handler shall be diagnosed because doing so can result in undefined behavior if a non-asynchronous-safe function is called as a result.

Noncompliant Code Example

In this noncompliant code example, `longjmp` is called from within the signal handler `handler`.

```
#include <setjmp.h>
#include <signal.h>
#include <stdlib.h>

enum { MAXLINE = 1024 };
static jmp_buf env;

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

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

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

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

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

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

int main(void) {
  if (signal(SIGINT, handler) == SIG_ERR) {
    /* Handle error */
  }
  char *info1;
```

```
  char *info2;

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

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

  return 0;
}
```

Bibliography

[Dowd 06] Chapter 13, "Synchronization and State"

[Greenman 97]

[ISO/IEC PDTR 24772] "EWD Structured Programming"

[MISRA 04] Rule 20.7

[MITRE 07] CWE ID 479, "Unsafe Function Call from a Signal Handler"

[Open Group 04] `longjmp`

[OpenBSD] `signal` Man Page

[Seacord 09] "SIG32-C. Do not call `longjmp` from inside a signal handler"

[VU #834865]

[Zalewski 01]

8.32 Do not use abort or assert when atexit handlers are registered [ERR006]

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

Noncompliant Code Example

In this noncompliant code example, the `cleanup` function is registered with `atexit` and `assert` is called afterwards. If the assertion fails then `cleanup` is not called.

```
void cleanup(void) {
  /* Delete temporary files, restore consistent state, etc. */
}

int main(void) {
  if (atexit(cleanup) != 0) {
    /* Handle error */
  }
```



```
  /* ... */

  assert(/* something bad didn't happen */);

  /* ... */
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.2.1.1, "The `assert` macro," and Section 7.20.4.1, "The `abort` function"

[ISO/IEC PDTR 24772] "REU Termination Strategy"

[Seacord 09] "ERR06-C. Understand the termination behavior of `assert` and `abort`"

8.33 Set and check errno correctly [ERR030]

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.

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 inspect `errno` before a subsequent library function call.

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

Function name	Return value	`errno` value
`ftell`	`-1L`	positive
`stroumax`	`UINTMAX_MAX`	`ERANGE`
`strtod`[4], `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`

[4] However, according to the C99 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."

wcstoumax	UINTMAX_MAX	ERANGE

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

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

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

Function name	Return value	`errno` value
`fgetpos`	nonzero	positive
`fgetwc`	`WEOF`	`EILSEQ`
`fputwc`	`WEOF`	`EILSEQ`
`fsetpos`	nonzero	positive
`mbrtowc`	`(size_t)(-1)`	`EILSEQ`
`mbsrtowcs`	`(size_t)(-1)`	`EILSEQ`
`signal`[5]	`SIG_ERR`	positive
`wcrtomb`	`(size_t)(-1)`	`EILSEQ`
`wcsrtombs`	`(size_t)(-1)`	`EILSEQ`

Library functions that may or may not set `errno`

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

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

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

- If there is a domain error and the integer expression `math_errhandling & MATH_ERRNO` is nonzero, then `errno` is set to `EDOM`.
- According to the C99 standard, "If a floating result overflows and default rounding is in effect, or if the mathematical result is an exact infinity (for example `log(0.0)`), then the function returns the value of the macro `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL` according to the return type, with the same sign as the correct value of the function; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value `ERANGE`."

[5] The value of `errno` is inderminate 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`.

- Similarly, according to the C99 standard, "The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, whether `errno` acquires the value `ERANGE` is implementation-defined."

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

Library functions that do not explicitly set `errno`

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

Noncompliant Code Example

In this noncompliant code example, `errno` is not set to zero before calling `strtoul`, a function that returns an in-band error indicator.

```
unsigned long number;
char *string;
char *endptr;
/* ... */
number = strtoul(string, &endptr, 0);
if (endptr == string || (number == ULONG_MAX && errno == ERANGE)) {
  /* handle the error */
} else {
  /* computation succeeded */
}
```

Noncompliant Code Example

In this noncompliant code example, the return value of `signal`, a function that returns an out-of-band error indicator, is not checked to make sure that an error occurred before checking `errno`.

```
signal(SIGINT, SIG_DFL);
if (errno != 0) {
  /* handle error */
}
```

Noncompliant Code Example

In this noncompliant code example, `errno` is set to zero then examined after calling `setlocale`, a function that does not explicitly set `errno`.

```
errno = 0;
setlocale(LC_ALL, "");
if (errno != 0) {
  /* handle error */
}
```

Bibliography

[Brainbell.com] Macros and Miscellaneous Pitfalls

[Horton 90] Section 11 p. 168 and Section 14 p. 254

[ISO/IEC 9899:1999] Section 7.1.4, "Use of library functions," and Section 7.5, "Errors <errno.h>"

[Koenig 89] Section 5.4 p. 73

[MITRE 07] CWE ID 456, "Missing Initialization"

[Seacord 09] "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"

8.34 Finish case labels with a break statement [MSC017]

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

Noncompliant Code Example

In this noncompliant code example, the case where widget_type has value WE_W lacks a concluding break statement.

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

switch (widget_type) {
  case WE_W:
    /* ... */
  case WE_X:
    /* ... */
    break;
  case WE_Y:
  case WE_Z:
    /* ... */
    break;
  default: /* can't happen */
         /* handle error condition */
}
```

Exceptions

MSC473:EX1: The last label in a switch statement, if it does not conclude with a break statement, need not be diagnosed.

Bibliography

[Seacord 09] "MSC17-C. Finish every set of statements associated with a case label with a break statement"

8.35 Do not assume a positive remainder when using the % operator [INT010]

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

Non-Compliant Code Example

In this noncompliant code example, the result of (index + 1) % size may be negative.

```
int insert(int index, int *list, int size, int value) {
  if (size != 0) {
    index = (index + 1) % size;
    list[index] = value;
```



```
    return index;
  }
  else {
    return -1;
  }
}
```

Bibliography

[Beebe 05]

[ISO/IEC 9899-1999] Section 6.5.5, "Multiplicative operators"

[Microsoft 07] C Multiplicative Operators

[MITRE 07] CWE ID 682, "Incorrect Calculation," and CWE ID 129, "Unchecked Array Indexing"

[Seacord 09] "INT10-C. Do not assume a positive remainder when using the % operator"

[Sun 05] Appendix E, "Implementation-Defined ISO/IEC C90 Behavior"

8.36 Ensure that floating point conversions are within range of the new type [FLP034]

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

Noncompliant Code Example

In this noncompliant code example, `f1` may not be within the range of `int`.

```
float f1;
int i1;

/* initialize f1 */

i1 = f1; /* Unspecified value if the integral part of f1 > INT_MAX */
```

Noncompliant Code Example

In this noncompliant code example, the conversions of the longer type may not be within the range of the smaller type.

```
long double ld;
double d1;
double d2;
float f1;
float f2;

/* initializations */

f1 = (float)d1;
f2 = (float)ld;
d2 = (double)ld;
```

Bibliography

[ISO/IEC 9899:1999] Section 6.3.1.4, "Real floating and integer," and Section 6.3.1.5, "Real floating types"

[ISO/IEC PDTR 24772] "FLC Numeric Conversion Errors"

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

[MITRE 07] CWE ID 681, "Incorrect Conversion between Numeric Types"

[Seacord 09] "FLP34-C. Ensure that floating point conversions are within range of the new type"

8.37 Null-terminate strings that are not null-terminated [STR032]

Failure to properly terminate null-terminated byte strings shall be diagnosed because operating on non-terminated byte strings can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, `dest` may not be null-terminated after the call to `strncpy`.

```
char dest[DEST_SIZE];
char src[SRC_SIZE];

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

strncpy(dest, src, sizeof(dest));
```

Noncompliant Code Example

In this noncompliant code example, `cur_msg` is not properly null-terminated after the call to `realloc`.

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

/* ... */

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

  /* ... */

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

/* ... */
```

Bibliography

[ISO/IEC 9899:1999] Section 7.1.1, "Definitions of terms," Section 7.20.3.4 "The realloc function," and Section 7.21, "String handling <string.h>"

[ISO/IEC PDTR 24772] "CJM String Termination"

[ISO/IEC TR 24731-1:2007] Section 6.7.1.4, "The strncpy_s function"

[MITRE 07] CWE ID 119, "Failure to Constrain Operations within the Bounds of an Allocated Memory Buffer," CWE ID 170, "Improper Null Termination"

[Schwarz 05]

[Seacord 05a] Chapter 2, "Strings"

[Seacord 09] "STR32-C. Null-terminate byte strings as required"

[Viega 05] Section 5.2.14, "Miscalculated NULL termination"

8.38 Do not assume character data does not contain a null byte [FIO037]

Assuming that the length of a string read by `fgets` or character functions in binary mode is the full length of the character data read shall be diagnosed because these functions can return character data that contains null bytes.

Noncompliant Code Example

In this noncompliant code example, `strlen(buf)` is assumed to be positive.

```
char buf[BUFSIZE + 1];

if (fgets(buf, sizeof(buf), stdin) == NULL) {
  /* Handle error */
}
buf[strlen(buf) - 1] = '\0';
```

Bibliography

[ISO/IEC 9899:1999] Section 7.19.7.2, "The `fgets` function"

[Lai 06]

[MITRE 07] CWE ID 119, "Failure to Constrain Operations within the Bounds of an Allocated Memory Buffer," CWE ID 241, and "Failure to Handle Wrong Data Type"

[Seacord 05a] Chapter 2, "Strings"

[Seacord 09] "FIO37-C. Do not assume character data has been read"

8.39 Close files when they are no longer needed [FIO042]

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

Noncompliant Code Example

In this noncompliant code example, a file containing sensitive data is opened but not closed before a call to `system`. If `system` spawns a child process that process could have access to the sensitive data file opened by the parent process.

```
FILE* f;
const char *editor;
char *file_name;

/* Initialize file_name */
```

```
f = fopen(file_name, "r");
if (f == NULL) {
  /* Handle fopen error */
}
/* ... */
editor = getenv("EDITOR");
if (editor == NULL) {
  /* Handle getenv error */
}
if (system(editor) == -1) {
  /* Handle error */
}
```

Bibliography

[Austin Group 08]

[Dowd 06] Chapter 10, "UNIX Processes" (File Descriptor Leaks 582-587)

[MITRE 07] CWE ID 404, "Improper Resource Shutdown or Release," and CWE ID 403, "UNIX File Descriptor Leak"

[MSDN] Inheritance (Windows)

[NAI 98]

[Seacord 09] "FIO42-C. Ensure files are properly closed when they are no longer needed"

8.40 Sanitize the environment when invoking external programs [ENV003]

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

Noncompliant Code Example

In this noncompliant code example, a string is passed to the command processor in the host environment to be executed. On a POSIX system, if the environment variable `IFS` is set to "." then the intended directory will not be found.

```
if (system("/bin/ls dir.`date +%Y%m%d`") == -1) {
  /* Handle error */
}
```

Bibliography

[Austin Group 08] Vol. 2, System Interfaces, `confstr`

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

[Dowd 06] Chapter 10, "UNIX II: Processes"

[ISO/IEC 9899:1999] Section 7.20.4, "Communication with the environment"

[ISO/IEC PDTR 24772] "XYS Executing or Loading Untrusted Code"

[MITRE 07] CWE ID 426, "Untrusted Search Path," CWE ID 88, "Argument Injection or Modification," and CWE ID 78, "Failure to Sanitize Data into an OS Command (aka 'OS Command Injection')"

[Open Group 04] Chapter 8, "Environment Variables", and confstr

[Seacord 09] "ENV03-C. Sanitize the environment when invoking external programs"

[Viega 03] Section 1.1, "Sanitizing the Environment"

[Wheeler 03] Section 5.2, "Environment Variables"

8.41 Do not take the size of a pointer to determine the size of the pointed-to type [EXP001]

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

Noncompliant Code Example

In this noncompliant code example, the `sizeof` operator is used incorrectly on the variable `d_array` instead of `*d_array`.

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

  if (num_elems > SIZE_MAX/sizeof(d_array)) {
    /* handle error condition */
  }
  d_array = (double *)malloc(sizeof(d_array) * num_elems);
  if (d_array == NULL) {
    /* handle error condition */
  }
  return d_array;
}
```

Bibliography

[Drepper 06] Section 2.1.1, "Respecting Memory Bounds"

[ISO/IEC 9899:1999] Section 6.5.3.4, "The `sizeof` operator"

[MITRE 07] CWE ID 467, "Use of sizeof on a Pointer Type"

[Seacord 09] "EXP01-C. Do not take the size of a pointer to determine the size of the pointed-to type"

[Viega 05] Section 5.6.8, "Use of `sizeof` on a pointer type"

8.42 Do not add or subtract a scaled integer to a pointer [EXP008]

Adding or subtracting a scaled integer value to a pointer shall be diagnosed because this can result in an invalid pointer.

Violations of this guidelines are typically indicated when a pointer to an array is added to the result of the `sizeof` operator or `offsetof` macro. However, adding an array pointer to the number of array elements, for example by using the `arr[sizeof(arr)/sizeof(arr[0])])` idiom, is allowed.

Noncompliant Code Example

In this noncompliant code example, `sizeof(buf)`, a value scaled by `sizef(int)`, is added to the pointer `buf`.

```
int buf[INTBUFSIZE];
int *buf_ptr = buf;
```

```
while (havedata && buf_ptr < (buf + sizeof(buf))) {
    *buf_ptr++ = parseint(getdata);
}
```

Noncompliant Code Example

In this noncompliant code example, `skip` represents the byte offset of `ull_2` in `struct big`. When added to `s`, `skip` is scaled by the size of `struct big`.

```
struct big {
  unsigned long long ull_1; /* typically 8 bytes */
  unsigned long long ull_2; /* typically 8 bytes */
  unsigned long long ull_3; /* typically 8 bytes */
  int si_4; /* typically 4 bytes */
  int si_5; /* typically 4 bytes */
};
/* ... */
size_t skip = offsetof(struct big, ull_2);
struct big *s = (struct big *)malloc(sizeof(struct big));
if (!s) {
  /* Handle malloc error */
}

memset(s + skip, 0, sizeof(struct big) - skip);
/* ... */
free(s);
s = NULL;
```

Noncompliant Code Example

In this noncompliant code example, `wcslen(error_msg) * sizeof(wchar_t)`, a value scaled by `sizeof(wchar_t)`, is added to the pointer `error_msg`.

```
/* ... */
wchar_t error_msg[WCHAR_BUF];

wcscpy(error_msg, L"Error: ");
fgetws(error_msg + wcslen(error_msg) * sizeof(wchar_t), WCHAR_BUF - 7, stdin);
/* ... */
```

Bibliography

[Dowd 06] Chapter 6, "C Language Issues"

[ISO/IEC PDTR 24772] "HFC Pointer casting and pointer type changes" and "RVG Pointer Arithmetic"

[MISRA 04] Rules 17.1-17.4

[MITRE 07] CWE ID 468, "Incorrect Pointer Scaling"

[Murenin 07]

[Seacord 09] "EXP08-C. Ensure pointer arithmetic is used correctly"

8.43 Do not ignore values returned by functions [EXP012]

Ignoring the value returned by a function shall be diagnosed (subject to exceptions below) because this can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the return value from `puts` is discarded.

```
puts("foo");
```

Exceptions

EXP012-EX1: The use of a `void` cast to signify programmer intent to ignore a return value from a function need not be diagnosed. The following example shows an acceptable use of this exception.

```
(void)fclose(fp);
```

EXP012-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.

Bibliography

[ISO/IEC 9899:1999] Section 6.8.3, "Expression and null statements"

[ISO/IEC PDTR 24772] "CSJ Passing Parameters and Return Values"

[Seacord 09] "EXP12-C. Do not ignore values returned by functions"

8.44 Detect and handle memory allocation errors [MEM032]

Not detecting and handling errors from memory allocation functions shall be diagnosed because doing so can result in undefined or unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the return value of `malloc` is not checked for error.

```
char *input_string= /* initialize from untrusted data */;

size_t size = strlen(input_string) + 1;
char *str = (char *)malloc(size);
strcpy(str, input_string);
/* ... */
free(str);
str = NULL;
```

Bibliography

[ISO/IEC 9899:1999] Section 7.20.3, "Memory management functions"

[MITRE 07] CWE ID 476, "NULL Pointer Dereference," and CWE ID 252, "Unchecked Return Value"

[Seacord 05] Chapter 4, "Dynamic Memory Management"

[Seacord 09] "MEM32-C. Detect and handle memory allocation error"

[VU#159523]

9 Good Practices

9.1 Avoid performing bitwise and arithmetic operations on the same data [INT014]

Using bitwise and arithmetic operatons on the same data shall be diagnosed because doing so reduces code portability and maintainability.

Noncompliant Code Example

In this noncompliant code example, both bit manipulation and arithmetic manipulation are performed on the integer `x` that make assumptions about the representation of unsigned integers.

```
unsigned int x = 50;
x += (x << 2) + 1;
```

Noncompliant Code Example

In this noncompliant code example, the right shift operator is assumed to divide by two.

```
int x = -50;
x >>= 2;
```

Bibliography

[ISO/IEC 9899:1999] Section 6.2.6.2, "Integer types"

[ISO/IEC PDTR 24772] "STR Bit Representations"

[MISRA 04] Rules 6.4 and 6.5

[Seacord 09] "INT14-C. Avoid performing bitwise and arithmetic operations on the same data"

[Steele 77]

9.2 Use typedefs to define types [PRE003]

Using a macro definition to define a type shall be diagnosed because the expansion of such macros can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, `s1` is declared as `char *` but `s2` is declared as a `char`, which is probably not what the programmer intended.

```
#define cstring char *
cstring s1, s2;
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7, "Declarations"

[ISO/IEC PDTR 24772] "NMP Pre-processor Directives"

[Saks 99]

[Seacord 09] "PRE03-C. Prefer `typedef`s to defines for encoding types"

[Summit 05] Question 1.13, Question 11.11

9.3 Use parentheses within macros around replacement lists [PRE002]

Macro replacement lists that are not parenthesized shall be diagnosed (subject to exceptions below) because the expansion of such macros can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the replacement list in the `CUBE` macro definition is not parenthesized.

```
#define CUBE(X) (X) * (X) * (X)
int i = 3;
int a = 81 / CUBE(i);
```

As a result, the invocation

```
int a = 81 / CUBE(i);
```

expands to

```
int a = 81 / i * i * i;
```

which evaluates as

```
int a = ((81 / i) * i) * i);  /* evaluates to 243 */
```

which is clearly not the desired behavior.

Exceptions

PRE002-EX1: A macro that expands to a single identifier or function call need not be diagnosed. The following code example shows acceptable use of an unparenthesized replacement list in a macro definition.

```
#define MY_PID getpid
```

Bibliography

[ISO/IEC 9899:1999] Section 6.10, "Preprocessing directives," and Section 5.1.1, "Translation environment"

[ISO/IEC PDTR 24772] "JCW Operator precedence/Order of Evaluation", "NMP Pre-processor Directions"

[Plum 85] Rule 1-1

[Seacord 09] "PRE02-C. Macro replacement lists should be parenthesized"

[Summit 05] Question 10.1

9.4 Use parentheses within macros around parameter names [PRE001]

Unparenthesized parameter names in function-like macro definitions shall be diagnosed (subject to exceptions below) because the expansion of such macros can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the function-like macro `CUBE` expands unexpectedly because the parameter `I` is not parenthesized.

```
#define CUBE(I) (I * I * I)
```

As a result, the invocation

```
int a = 81 / CUBE(2 + 1);
```

expands to

```
int a = 81 / (2 + 1 * 2 + 1 * 2 + 1);  /* evaluates to 11 */
```

which is clearly not the desired result.

Exceptions

PRE001-EX1: Parameter names surrounded by comma operators or comma separators in the body of a macro need not be diagnosed. The following code example shows an acceptable use of unparenthesized parameter names surrounded by comma separators.

```
#define FOO(a, b, c) bar(a, b, c)
/* ... */
FOO(arg1, arg2, arg3);
```

PRE001-EX2: Parameter names that are operands of the `##` or `#` operators or are adjacent string literals need not be diagnosed. The following code example shows an acceptable use of unparenthesized parameter names in these cases.

```
#define JOIN(a, b) (a ## b)
#define SHOW(a) printf(#a " = %d\n", a)
```

Bibliography

[ISO/IEC 9899:1999] Section 6.10, "Preprocessing directives," and Section 5.1.1, "Translation environment"

[ISO/IEC PDTR 24772] "JCW Operator precedence/Order of Evaluation"

[MISRA 04] Rule 19.1

[Plum 85]

[Seacord 09] "PRE01-C. Use parentheses within macros around parameter names"

[Summit 05] Question 10.1

9.5 Do not conclude a single-statement macro definition with a semicolon [PRE011]

A single-statement macro definition that is concluded with a semicolon shall be diagnosed because the expansion of such macros can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, `FOR_LOOP(3)` expands into a `for`-loop with a null body.

```
#define FOR_LOOP(n)  for(i=0; i<(n); i++);

int i;
FOR_LOOP(3)
  puts("Inside for loop\n");
```

The above example expands to

```
int i;
```



```
for(i=0; i<(n); i++)
  ;

puts("Inside for loop\n");
```

which produces unexpected results.

Bibliography

[Seacord 09] "PRE11-C. Do not conclude a single-statement macro definition with a semicolon"

9.6 Wrap multi-statement macros in a do-while loop [PRE010]

A multi-statement macro that is not contained within a do-while loop shall be diagnosed because the expansion of such macros can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the macro `SWAP` expands unexpectedly in an `if` statement.

```
/*
 * Swaps two values.
 * Requires tmp variable to be defined.
 */
#define SWAP(x, y) \
  tmp = x; \
  x = y; \
  y = tmp

int x, y, z, tmp;
if (z == 0)
  SWAP( x, y);
```

This expands to

```
int x, y, z, tmp;
if (z == 0)
  tmp = x;
x = y;
y = tmp;
```

which is certainly not what the author intended.

Bibliography

[ISO/IEC PDTR 24772] "NMP Pre-processor Directions"

[Seacord 09] "PRE10-C. Wrap multi-statement macros in a do-while loop"

9.7 Use visually distinct identifiers [DCL002]

Using multiple identifiers that vary only with respect to one or more visually similar characters shall be diagnosed because visually distinct identifiers aid the programmer.

The following characters are visually similar in some fonts.

1 (one)	I (capital i)	l (lowercase L)

0 (zero)	O (capital o)	D (capital d)
2 (two)	Z (capital z)	
5 (five)	S (capital s)	
8 (eight)	B (capital b)	
n (lowercase N)	h (lowercase H)	
rn (lowercase R, lowercase N)	m (lowercase M)	

Bibliography

[ISO/IEC 9899:1999] Section 5.2.4.1, "Translation limits"

[ISO/IEC PDTR 24772] "AJN Choice of Filenames and other External Identifiers," "BRS Leveraging human experience," and "NAI Choice of Clear Names"

[MISRA 04] Rule 5.6

[Seacord 09] "DCL02-C. Use visually distinct identifiers"

9.8 Use a static assertion to test the value of a constant expression [DCL003]

Using a runtime assertion to test the value of a constant expression shall be diagnosed because a static assertion incurs less runtime overhead and is not affected by program flow.

Noncompliant Code Example

In this noncompliant code example, a runtime assertion is used where a static assertion could be used.

```
struct timer {
  uint8_t MODE;
  uint32_t DATA;
  uint32_t COUNT;
};

int func(void) {
  assert(offsetof(timer, DATA) == 4);
}
```

Bibliography

[Becker 08]

[Eckel 07]

[ISO/IEC 9899:1999] Section 6.10.1, "Conditional inclusion," and Section 6.10.3.3, "The ## operator," and Section 7.2.1, "Program diagnostics"

[Klarer 04]

[Saks 05]

[Saks 08]

[Seacord 09] "DCL03-C. Use a static assertion to test the value of a constant expression"

9.9 Use typedefs to improve code readability [DCL005]

Using sufficiently complicated types in expressions shall be diagnosed because they decrease code readability.

Noncompliant Code Example

In this noncompliant code example, the declaration of the `signal` function is difficult to read and comprehend.

```
void (*signal(int, void (*)(int)))(int);
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.7, "Type definitions"

[ISO/IEC PDTR 24772] "BRS Leveraging human experience"

[Seacord 09] "DCL05-C. Use `typedefs` to improve code readability"

9.10 Do not declare more than one variable per declaration [DCL004]

Declaring multiple variables in one declaration shall be diagnosed (subject to exceptions below) because this can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the variable `c` could be mistaken to have type `char *` when it actually has type `char`.

```
char *src = 0, c = 0;
```

Noncompliant Example

In this noncompliant code example, the variable `i` could be mistaken to have been initialized to 1 when it is actually uninitialized.

```
int i, j = 1;
```

Exceptions

DCL004-EX1: Trivial declarations for loop counters need not be diagnosed. The following example shows an acceptable use of this exception.

```
for (size_t i = 0; i < mx; ++i ) {
  /* ... */
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7, "Declarations"

[ISO/IEC PDTR 24772] "BRS Leveraging human experience"

[Seacord 09] "DCL04-C. Do not declare more than one variable per declaration"

9.11 Use 'L', not 'l', to indicate a long value [DCL016]

Declaring a long value that is indicated using the lower case letter 'l' (ell) shall be diagnosed because it is easily confused with the number '1' (one).

Noncompliant Code Example

In this noncompliant code example, the integers 1111 and 111 are added, but the second integer appears to be 1111 due to an appended 'l' (lowercase ell).

```
printf("Sum is %ld\n", 1111 + 111l);
```

Bibliography

[Seacord 09] "DCL16-C. Use 'L', not 'l', to indicate a long value"

9.12 Use parentheses for precedence of operation [EXP000]

Using the `&`, `|`, `^`, `<<`, and `>>` operators without enclosing parentheses shall be diagnosed because these operators have unintuitive low-precedence that can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the statement

```
x & 1 == 0
```

is evaluated as if it had parentheses like

```
x & (1 == 0)
```

which evaluates to

```
(x & 0)
```

and then to `0`, which is the unintended result.

Bibliography

[Dowd 06] Chapter 6, "C Language Issues" (Precedence, pp. 287—288)

[ISO/IEC 9899:1999] Section 6.5, "Expressions"

[ISO/IEC PDTR 24772] "JCW Operator precedence/Order of Evaluation"

[Kernighan 88]

[MISRA 04] Rule 12.1

[NASA-GB-1740.13] Section 6.4.3, "C Language"

[Seacord 09] "EXP00-C. Use parentheses for precedence of operation"

9.13 Do not use relational operators on boolean values [EXP013]

Using a relational operator on a boolean value shall be diagnosed because this typically indicates programmer error and can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the programmer mistook the meaning of `a < b < c` and `a == b == c`.

```
int a = 2;
int b = 2;
int c = 2;
/* ... */
if (a < b < c) /* misleading, likely bug */
/* ... */
if (a == b == c) /* misleading, likely bug */
```

Bibliography

[Seacord 09] "EXP13-C. Treat relational and equality operators as if they were nonassociative"

9.14 Do not include unused values [MSC013]

Including unused values shall be diagnosed because they typically indicate programmer error.

Noncompliant Code Example

In this noncompliant code example, the value returned by `bar` is never used.

```
int *p1, *p2;
p1 = foo();
p2 = bar();

if (baz()) {
  return p1;
}
else {
  p2 = p1;
}

return p2;
```

Bibliography

[Coverity 07]

[ISO/IEC PDTR 24772] "BRS Leveraging human experience," "KOA Likely incorrect expressions," "XYQ Dead and Deactivated Code," and "XYR Unused Variable"

[Seacord 09] "MSC13-C. Detect and remove unused values"

9.15 Do not use code that has no effect [MSC012]

Code that has no effect shall be diagnosed because this typically indicates programmer error and can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, the comparison of `a` to `b` has no effect.

```
int a;
int b;
/* ... */
a == b;
```

Noncompliant Code Example

In this noncompliant code example, dereferencing `p++` has no effect.

```
int *p;
/* ... */
*p++;
```

Bibliography

[Coverity 07]

[ISO/IEC PDTR 24772] "BRS Leveraging human experience," "BVQ Unspecified Functionality," "KOA Likely incorrect expressions," and "XYQ Dead and Deactivated Code"

[MISRA 04] Rule 14.1 and Rule 14.2

[Seacord 09] "MSC12-C. Detect and remove code that has no effect"

9.16 Do not use dead code [MSC007]

Code that is never executed shall be diagnosed (subject to exceptions below) because it typically indicates programmer error and reduces readability.

Noncompliant Code Example

In this noncompliant code example, the conditional `if (s)` never evaluates to true since `s` is always `NULL`.

```
int func(int condition) {
    char *s = NULL;
    if (condition) {
        s = (char *)malloc(10);
        if (s == NULL) {
           /* Handle Error */
        }
        /* Process s */
        return 0;
    }
    /* ... */
    if (s) {
        /* This code is never reached */
    }
    return 0;
}
```

Noncompliant Code Example

In this noncompliant code example, the conditional `if (str[i] == '\0')` never evaluates to true since `strlen` returns the number of characters that precede the null terminator.

```
int string_loop(char *str) {
    size_t i;
    size_t len = strlen(str);
    for (i=0; i < len; i++) {
        /* ... */
        if (str[i] == '\0')
            /* This code is never reached */
    }
```



```
    return 0;
}
```

Exceptions

MSC007-EX1: Dead code that makes software resilient to future changes need not be diagnosed (i.e. the default case on a switch statement).

MSC007-EX2: Dead code that is intentionally temporarily removed need not be diagnosed because it may be used in the future.

Bibliography

[Fortify 06] Code Quality, "Dead Code"

[ISO/IEC PDTR 24772] "BRS Leveraging human experience," "BVQ Unspecified Functionality," and "XYQ Dead and Deactivated Code"

[MISRA 04] Rule 2.4

[MITRE 07] CWE ID 561, "Dead Code"

[Seacord 09] "MSC07-C. Detect and remove dead code"

9.17 Use comments consistently and in a readable fashion [MSC004]

The use of the character sequence `/*` within a comment shall be diagnosed because using a consistent comment style aids the programmer. Additionally, the following comment styles shall be diagnosed for the same reasons.

```
// */          /* comment, not syntax error */

f = g/**//h;   /* equivalent to f = g / h; */

//\
i;           /* part of a two-line comment */

/\
/ j;         /* part of a two-line comment */


/*//*/ l;    /* equivalent to l; */

m = n//**/o
+ p;           /* equivalent to m = n + p; */

a = b //*divisor:*/c
+d;            /* interpreted as a = b/c +d; in c90
                * compiler and a = b+d; in c99 compiler */
```

Noncompliant Code Example

In this noncompliant code example, the character sequence `/*` is used within a comment.

```
/* comment with end comment marker unintentionally omitted
security_critical_function();
/* some other comment */
```

Bibliography

[ISO/IEC 9899:1999] Section 6.4.9, "Comments," and Section 6.10.1, "Conditional inclusion"

[MISRA 04] Rule 2.2, "Source code shall only use /* ... */ style comments," Rule 2.3, "The character sequence /* shall not be used within a comment," and Rule 2.4, "Sections of code should not be "commented out"

[Seacord 09] "MSC04-C. Use comments consistently and in a readable fashion"

[Summit 05] Question 11.19

9.18 Use unique header file names [PRE008]

Using header file names that do not meet the following criteria shall be diagnosed (subject to exceptions below) because they may conflict:

- the first eight case-insensitive characters are distinct, and
- there is only one nondigit, case-insensitive character after the period.

Noncompliant Code Example

In this noncompliant code example, the headers `Library.h` and `library.h` as well as `utilities_math.h` and `utilities_physics.h` may refer to the same file. Also, if a file such as `my_libraryOLD.h` exists in the header search path, it may be included instead of `my_library.h`.

```
#include "Library.h"
#include <stdio.h>
#include <stdlib.h>
#include "library.h"

#include "utilities_math.h"
#include "utilities_physics.h"

#include "my_library.h"

/* Rest of program */
```

Exceptions

PRE008-EX1: Code written for implementations that support longer restrictions need not be diagnosed.

Bibliography

[ISO/IEC 9899:1999] Section 6.10.2, "Source file inclusion"

[MISRA 04] Rule 19.5

[Seacord 09] "PRE08-C. Guarantee that header file names are unique"

9.19 Enclose header files in an inclusion guard [PRE006]

Including a header file that does not have an inclusion guard shall be diagnosed because including unguarded header files multiple times can result in unexpected behavior.

Bibliography

[ISO/IEC 9899:1999] Section 6.10, "Preprocessing directives," Section 5.1.1, "Translation environment," and Section 7.1.2, "Standard headers"

[MISRA 04] Rule 19.5

[Plum 85] Rule 1-14

[Seacord 09] "PRE06-C. Enclose header files in an inclusion guard"

9.20 Use plain char for characters in the basic character set [STR004]

Using a type other than unqualified `char` for characters in the basic character set shall be diagnosed becuase the unqualified `char` type is the most compatible.

Noncompliant Code Example

In this noncompliant code example, `unsigned` and `signed char` strings are used to represent character data.

```
size_t len;
char cstr[] = "char string";
signed char scstr[] = "signed char string";
unsigned char ucstr[] = "unsigned char string";

len = strlen(cstr);
len = strlen(scstr);  /* warns when char is unsigned */
len = strlen(ucstr);  /* warns when char is signed */
```

Bibliography

[ISO/IEC 9899:1999] Section 6.2.5, "Types"

[MISRA 04] Rule 6.1, "The plain `char` type shall be used only for the storage and use of character values"

[Seacord 09] "STR04-C. Use plain `char` for characters in the basic character set"

9.21 Reset strings after fgets failure [FIO040]

After the failure of the functions `fgets` and `fgetws`, not resetting the array parameter to a known string shall be diagnosed (subject to exceptions below) because the contents of the array parameter are undefined.

Noncompliant Code Example

In this noncompliant code example, the value of `buf` is not reset if `fgets` returns an error.

```
char buf[BUFSIZ];
FILE *file;
/* Initialize file */

if (fgets(buf, sizeof(buf), file) == NULL) {
  /* set error flag and continue */
}
```

Exceptions

FIO040-EX1: Not resetting the array passed to `fgets` or `fgetws` to a known string after failure need not be diagnosed if the array goes out of scope immediately or if it is not referenced in case of failure.

Bibliography

[ISO/IEC 9899:1999] Section 7.19.7.2, "The `fgets` function" and Section 7.24.3.2, "the `fgetws` function"

[Seacord 09] "FIO40-C. Reset strings on `fgets` failure"

9.22 Do not store or use naked function pointers [MSC016]

Storing or using naked function pointers shall be diagnosed because an attacker can execute a payload through a naked function pointer when another vulnerability is present. A naked function pointer is a function pointer that is not obfuscated by a secret at runtime.

Noncompliant Code Example

In this noncompliant code example, a naked function pointer is used to call `printf`. If a vulnerability exists in this program that allows an attacker to overwrite the `log_fn` function pointer (such as a buffer overflow or arbitrary memory write), the attacker may be able to overwrite the value of `printf` with the location of an arbitrary function.

```
int (*log_fn)(const char *, ...) = printf;
/* ... */
log_fn("foo");
```

Bibliography

[MSDN] EncodePointer, DecodePointer

[Seacord 09] "MSC16-C. Consider encrypting function pointers"

9.23 Const-qualify immutable variables [DCL000]

Using immutable variables that are not `const`-qualified shall be diagnosed (subject to exceptions below) because enforcing object immutability aids the programmer.

Noncompliant Code Example

In this noncompliant code example, `pi` is an immutable constant but is not `const`-qualified.

```
float pi = 3.14159f;
float degrees;
float radians;
/* ... */
radians = degrees * pi / 180;
```

Bibliography

[Dewhurst 02] Gotcha #25, "#define Literals"

[ISO/IEC 9899:1999] Section 6.7.3, "Type qualifiers"

[Saks 00]

[Seacord 09] "DCL00-C. Const-qualify immutable objects"

9.24 Declare functions that return an errno error code with a return type of errno_t [DCL009]

Declaring a function that returns the value of `errno` or one of the codes defined in `<errno.h>` and does not have return type `errno_t` shall be diagnosed because it is unclear what values the function may return.

Noncompliant Code Example

In this noncompliant code example, the function `opener` returns `errno` error codes but is declared as returning `int`.

```
enum { NO_FILE_POS_VALUES = 3 };

int opener(
  FILE *file,
  int *width,
  int *height,
  int *data_offset
) {
  int file_w;
  int file_h;
  int file_o;
  fpos_t offset;

  if (file == NULL) { return EINVAL; }
  errno = 0;
  if (fgetpos(file, &offset) != 0) { return errno; }
  if (fscanf(file, "%i %i %i", &file_w, &file_h, &file_o)
        != NO_FILE_POS_VALUES) {
    return EIO;
  }

  errno = 0;
  if (fsetpos(file, &offset) != 0) { return errno; }

  if (width != NULL) { *width = file_w; }
  if (height != NULL) { *height = file_h; }
  if (data_offset != NULL) { *data_offset = file_o; }

  return 0;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.7.5.3, "Function declarators (including prototypes)"

[ISO/IEC PDTR 24772] "NZN Returning error status"

[ISO/IEC TR 24731-1:2007]

[MISRA 04] Rule 20.5

[Open Group 04]

[Seacord 09] "DCL09-C. Declare functions that return an `errno` error code with a return type of `errno_t`"

9.25 Declare file-scope objects or functions without external linkage as static [DCL015]

Declaring a file-scope object or function without the `static` qualifier that is not used outside of the file shall be diagnosed because this creates less modular code and pollutes the global name space.

Noncompliant Code Example

In this noncompliant code example, the function `helper` is implicitly declared to have external linkage but is not used outside of this file.

```
enum { MAX = 100 };

int helper(int i) {
  /* perform some computation based on i */
}

int main(void) {
  size_t i;
  int out[MAX];

  for (i = 0; i < MAX; i++) {
    out[i] = helper(i);
  }

  /* ... */

}
```

Bibliography

[ISO/IEC 9899:1999] Section 6.2.2, "Linkages of identifiers"

[Seacord 09] "DCL15-C. Declare file-scope objects or functions that do not need external linkage as static"

9.26 Declare parameter pointers whose pointed-to values do not change as const [DCL013]

Declaring function parameters that are pointers to values that are not modified in the function body but are not `const`-qualified shall be diagnosed because `const`-qualification of such pointers aids the programmer.

Noncompliant Code Example

In this noncompliant code example, `x` has type pointer to `int` but the value pointed to by `x` is not modified.

```
void foo(int *x) {
  /* ... */
  /* value pointed to by x not modified in body. */
  /* ... */
}
```

Bibliography

[ISO/IEC 9899:1999]

[ISO/IEC PDTR 24772] "CSJ Passing parameters and return values"

[Seacord 09] "DCL13-C. Declare function parameters that are pointers to values not changed by the function as `const`"

9.27 Explicitly specify array bounds [ARR002]

Declaring an array without an initialization literal or with an incorrect initialization literal shall be diagnosed (subject to exceptions below) because providing an initialization literal aids the programmer.

Noncompliant Code Example

In this noncompliant code example, the array `a` is declared with an incorrect initialization literal.

```
int a[3] = {1, 2, 3, 4};
```

Noncompliant Code Example

In this noncompliant code example, the array `a` is declared without an initialization literal.

```
int a[] = {1, 2, 3, 4};
```

Exceptions

ARR002-EX1: Declaring an array that is initialized with a string literal and without an initialization literal shall not be diagnosed.

Bibliography

[ISO/IEC 9899:1999] Section 6.7.8, "Initialization"

[MITRE 07] CWE ID 665, "Incorrect or Incomplete Initialization"

[Seacord 09] "ARR02-C. Explicitly specify array bounds, even if implicitly defined by an initializer"

9.28 Store a new value in pointers immediately after free [MEM001]

Not storing a new value in a pointer immediately after it has been freed shall be diagnosed (subject to exceptions below) because storing a new value in a freed programmer aids the programmer.

Noncompliant Code Example

In this noncompliant code example, the pointer `message` is not set to a new value after it is freed.

```
char *message;
int message_type;

/* Initialize message and message_type */

if (message_type == value_1) {
  /* Process message type 1 */
  free(message);
}
/* ...*/
if (message_type == value_2) {
   /* Process message type 2 */
  free(message);
}
```

Exceptions

MEM001-EX1: Not storing a new value in a nonstatic variable that goes out of scope immediately after being freed need not be diagnosed. The following example shows an acceptable use of this exception.

```
void foo(void) {
  char *str;
  /* ... */
  free(str);
```

```
  return;
}
```

Bibliography

[ISO/IEC 9899:1999] Section 7.20.3.2, "The `free` function"

[ISO/IEC PDTR 24772] "DCM Dangling references to stack frames," "XYK Dangling Reference to Heap," and "XZH Off-by-one Error"

[MITRE 07] CWE ID 416, "Use After Free," and CWE ID 415, "Double Free"

[Plakosh 05]

[Seacord 05a] Chapter 4, "Dynamic Memory Management"

[Seacord 09] "MEM01-C. Store a new value in pointers immediately after free"

Annex A
(normative)

Tool-generated and Tool-maintained Code

A.1 General

The following rules and recommendations do not apply to tool-generated and tool-maintained code:

DCL02-C. Use visually distinct identifiers

DCL04-C. Do not declare more than one variable per declaration

DCL05-C. Use typedefs to improve code readability

DCL06-C. Use meaningful symbolic constants to represent literal values in program logic

DCL32-C. Guarantee that mutually visible identifiers are unique

A.2 Clause

Subclause (level 1)

A.2.1.1 Subclause (level 2)

Bibliography

[1] [Apple 06] Apple, Inc. *Secure Coding Guide*, May 2006.

[2] [Austin Group 08] "Draft Standard for Information Technology - Portable Operating System Interface (POSIX®) - Draft Technical Standard: Base Specifications, Issue 7," IEEE Unapproved Draft Std P1003.1 D5.1. Prepared by the Austin Group. New York: Institute of Electrical & Electronics Engineers, Inc., May 2008.

[3] [Banahan 03] Banahan, Mike. *The C Book*, 2003.

[4] [Beebe 05] Beebe, Nelson H. F. Re: Remainder (%) operator and GCC, 2005.

[5] [Becker 08] Becker, Pete. Working Draft, Standard for Programming Language C++, April 2008.

[6] [Black 2007] Paul E. Black, Michael Kass, Michael Koo. Source Code Security Analysis Tool Functional Specification Version 1.0. Special Publication 500-268. Information Technology Laboratory (ITL), Software Diagnostics and Conformance Testing Division, May 2007. http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268.pdf

[7] [Brainbell.com] Brainbell.com. *Advice and Warnings for C Tutorials*.

[8] [Bryant 03] Bryant, Randal E., & O'Halloran, David. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003 (ISBN 0-13-034074-X).

[9] [Burch 06] Burch, Hal, Long, Fred, & Seacord, Robert C. *Specifications for Managed Strings* (CMU/SEI-2006-TR-006). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

[10] [Callaghan 95] Callaghan, B., Pawlowski, B., & Staubach, P. IETF RFC 1813 NFS Version 3 Protocol Specification, June 1995.

[11] [CERT 06a] CERT/CC. CERT/CC Statistics 1988---2006.

[12] [CERT 06b] CERT/CC. US-CERT's Technical Cyber Security Alerts.

[13] [CERT 06c] CERT/CC. Secure Coding web site.

[14] [Chen 02] Chen, H., Wagner, D., & Dean, D. "Setuid demystified." USENIX Security Symposium, 2002.

[15] [Corfield 93] Corfield, Sean A. "Making String Literals 'const'," November 1993.

[16] [Coverity 07] Coverity Prevent User's Manual (3.3.0), 2007.

[17] [CVE] Common Vulnerabilities and Exposures.

[18] [C++ Reference] Standard C Library, General C+, C+ Standard Template Library

[19] [Dewhurst 02] Dewhurst, Stephen C. *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Boston: Addison-Wesley Professional, 2002.

[20] [Dewhurst 05] Dewhurst, Stephen C. *C++ Common Knowledge: Essential Intermediate Programming*. Boston, MA: Addison-Wesley Professional, 2005.

[21] [DHS 06] U.S. Department of Homeland Security. Build Security In.

[22] [DOD 5220] U.S. Department of Defense. DoD Standard 5220.22-M (Word document).

[23] [Dowd 06] Dowd, M., McDonald, J., & Schuh, J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Boston: Addison-Wesley, 2006. See http://taossa.com for updates and errata.

[24] [Drepper 06] Drepper, Ulrich. Defensive Programming for Red Hat Enterprise Linux (and What To Do If Something Goes Wrong), May 3, 2006.

[25] [Eckel 07] Eckel, Bruce. *Thinking in C++ Volume 2*, January 25, 2007.

[26] [ECTC 98] Embedded C++ Technical Committee. *The Embedded C++ Programming Guide Lines*, Version WP-GU-003. January 6, 1998.

[27] [Finlay 03] Finlay, Ian A. CERT Advisory CA-2003-16, Buffer Overflow in Microsoft RPC. CERT/CC, July 2003.

[28] [Fisher 99] Fisher, David & Lipson, Howard. "Emergent Algorithms - A New Method for Enhancing Survivability in Unbounded Systems." *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences (HICSS-32)*. Maui, HI, January 5-8, 1999.

[29] [Flake 06] Flake, Halvar. "Attacks on uninitialized local variables." Black Hat Federal 2006.

[30] [Fortify 06] Fortify Software Inc. Fortify Taxonomy: Software Security Errors, 2006.

[31] [FSF 05] Free Software Foundation. GCC online documentation, 2005.

[32] [Garfinkel 96] Garfinkel, Simson & Spafford, Gene. *Practical UNIX & Internet Security*, 2nd Edition. Sebastopol, CA: O'Reilly Media, April 1996 (ISBN 1-56592-148-8).

[33] [GNU Pth] Engelschall, Ralf S. GNU Portable Threads, 2006.

[34] [Goldberg 91] Goldberg, David. What Every Computer Scientist Should Know About Floating-Point Arithmetic. Sun Microsystems, March 1991.

[35] [Gough 2005] Gough, Brian J. An Introduction to GCC. Network Theory Ltd, Revised August 2005 (ISBN 0-9541617-9-3).

[36] [Graff 03] Graff, Mark G. & Van Wyk, Kenneth R. *Secure Coding: Principles and Practices*. Cambridge, MA: O'Reilly, 2003 (ISBN 0596002424).

[37] [Greenman 97] Greenman, David. *serious security bug in wu-ftpd v2.4*. BUGTRAQ Mailing List (bugtraq@securityfocus.com), January 2, 1997.

[38] [Griffiths 06] Griffiths, Andrew. "Clutching at straws: When you can shift the stack pointer."

[39] [Gutmann 96] Gutmann, Peter. Secure Deletion of Data from Magnetic and Solid-State Memory, July 1996.

[40] [Haddad 05] Haddad, Ibrahim. "Secure Coding in C and C++: An interview with Robert Seacord, senior vulnerability analyst at CERT." *Linux World Magazine*, November 2005.

[41] [Hatton 95] 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).

[42] [Henricson 92] Henricson, Mats, & Nyquist, Erik. Programming in C++, Rules and Recommendations. Ellemtel Telecommunication Systems Laboratories, 1992.

[43] [Horton 90] Horton, Mark R. *Portable C Software*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1990 (ISBN:0-13-868050-7).

[44] [Howard 02] Howard, Michael, & LeBlanc, David C. *Writing Secure Code, 2nd ed. Redmond, WA:*. Microsoft Press, December 2002.

[45] [HP 03] Tru64 UNIX: Protecting Your System Against File Name Spoofing Attacks. Houston, TX: Hewlett-Packard Company, January 2003.

[46] [IEC 60812 2006] *Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA)*, 2nd ed. (IEC 60812). IEC, January 2006.

[47] [IEC 61508-4] *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 4: Definitions and abbreviations*, 1998.

[48] [IEEE Std 610.12 1990] *IEEE Standard Glossary of Software Engineering Terminology*, September 1990.

[49] [IEEE 754 2006] IEEE. *Standard for Binary Floating-Point Arithmetic* (IEEE 754-1985), 2006.

[50] [ilja 06] ilja. "readlink abuse." *ilja's blog*, August 13, 2006.

[51] [Intel 01] Intel Corp. *Floating-Point IEEE Filter for Microsoft* Windows* 2000 on the Intel® Itanium™ Architecture*, March 2001.

[52] [Internet Society 00] The Internet Society. Internet Security Glossary (RFC 2828), 2000.

[53] [ISO/IEC 646:1991] ISO/IEC. *Information technology: ISO 7-bit coded character set for information interchange* (ISO/IEC 646-1991). Geneva, Switzerland: International Organization for Standardization, 1991.

[54] [ISO/IEC 9945:2003] *ISO/IEC 9945:2003 (including Technical Corrigendum 1), Information technology — Programming languages, their environments and system software interfaces — Portable Operating System Interface (POSIX®)*.

[55] [ISO/IEC 9899:1999] ISO/IEC. *Programming Languages---C, 2nd ed* (ISO/IEC 9899:1999). Geneva, Switzerland: International Organization for Standardization, 1999.

[56] [ISO/IEC 10646:2003] *Information technology - Universal Multiple-Octet Coded Character Set (UCS)* (ISO/IEC 10646:2003). Geneva, Switzerland: International Organization for Standardization, 2003.

[57] [ISO/IEC 14882:2003] ISO/IEC. *Programming Languages — C++, Second Edition* (ISO/IEC 14882-2003). Geneva, Switzerland: International Organization for Standardization, 2003.

[58] [ISO/IEC 23360-1:2006] *Linux Standard Base (LSB) core specification 3.1 - Part 1: Generic specification*

[59] [ISO/IEC 03] ISO/IEC. *Rationale for International Standard — Programming Languages — C, Revision 5.10*. Geneva, Switzerland: International Organization for Standardization, April 2003.

[60] [ISO/IEC JTC1/SC22/WG11] ISO/IEC. *Binding Techniques* (ISO/IEC JTC1/SC22/WG11), 2007.

[61] [ISO/IEC DTR 24732] ISO/IEC JTC1 SC22 WG14 N1290. Extension for the programming language C to support decimal floating-point arithmetic, March 2008.

[62] [ISO/IEC PDTR 24731-2] Extensions to the C Library, — Part II: Dynamic Allocation Functions, August 2007.

[63] [ISO/IEC PDTR 24772] ISO/IEC PDTR 24772. *Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use*, March 2008.

[64] [ISO/IEC TR 24731-1:2007] ISO/IEC TR 24731. *Extensions to the C Library, — Part I: Bounds-checking interfaces*. Geneva, Switzerland: International Organization for Standardization, April 2006.

[65] [Jack 07] Jack, Barnaby. *Vector Rewrite Attack*, May 2007.

[66] [Jones 04] Jones, Nigel. "Learn a new trick with the offsetof macro." *Embedded Systems Programming*, March 2004.

[67] [Jones 08] Jones, Derek M. The New C Standard: An economic and cultural commentary. Knowledge Software Ltd., 2008.

[68] [Keaton 2009] David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky. As-if Infinitely Ranged Integer Model. CMU/SEI-2009-TN-XXX.

[69] [Keil 08] Keil, an ARM Company. "Floating Point Support." *RealView Libraries and Floating Point Support Guide*, 2008.

[70] [Kennaway 00] Kennaway, Kris. Re: /tmp topic, December 2000.

[71] [Kernighan 88] Kernighan , Brian W., & Ritchie, Dennis M. *The C Programming Language, 2nd ed.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

[72] [Kettlewell 02] Kettlewell, Richard. *C Language Gotchas*, February 2002.

[73] [Kettlewell 03] Kettlewell, Richard. *Inline Functions In C*, March 2003.

[74] [Kirch-Prinz 02] Kirch-Prinz, Ulla & Prinz, Peter. *C Pocket Reference*. Sebastopol, CA: O'Reilly, November 2002 (ISBN: 0-596-00436-2).

[75] [Klarer 04] Klarer, R., Maddock, J., Dawes, B. & Hinnant, H. "Proposal to Add Static Assertions to the Core Language (Revision 3)." ISO C++ committee paper ISO/IEC JTC1/SC22/WG21/N1720, October 2004. Available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html.

[76] [Klein 02] Klein, Jack. *Bullet Proof Integer Input Using strtol*, 2002.

[77] [Koenig 89] Koenig, Andrew. *C Traps and Pitfalls*. Addison-Wesley Professional, January 1, 1989.

[78] [Kuhn 06] Kuhn, Markus. *UTF-8 and Unicode FAQ for Unix/Linux*, 2006.

[79] [Lai 06] Lai, Ray. "Reading Between the Lines." *OpenBSD Journal*, October 2006.

[80] [Lewis 06] Lewis, Richard. "Security Considerations When Handling Sensitive Data." Posted on the Application Security by Richard Lewis blog October 2006.

[81] [Linux 07] Linux Programmer's Manual, July 2007.

[82] [Lions 96] Lions, J. L. ARIANE 5 Flight 501 Failure Report. Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[83] [Lipson 00] Lipson, Howard & Fisher, David. "Survivability: A New Technical and Business Perspective on Security," 33-39. *Proceedings of the 1999 New Security Paradigms Workshop*. Caledon Hills, Ontario, Canada, Sept. 22-24, 1999. New York: Association for Computing Machinery, 2000.

[84] [Lipson 06] Lipson, Howard. *Evolutionary Systems Design: Recognizing Changes in Security and Survivability Risks* (CMU/SEI-2006-TN-027). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

[85] [Lockheed Martin 05] Lockheed Martin. "Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program." Document Number 2RDU00001 Rev C., December 2005.

[86] [Loosemore 07] Loosemore, Sandra, Stallman, Richard M., McGrath, Roland, Oram, Andrew, & Drepper, Ulrich. The GNU C Library Reference Manual, Edition 0.11, September 2007.

[87] [McCluskey 01] *flexible array members and designators in C9X* ;login:, July 2001, Volume 26, Number 4, p. 29---32.

[88] [Mell 07] P. Mell, K. Scarfone, and S. Romanosky, "A Complete Guide to the Common Vulnerability Scoring System Version 2.0", FIRST, June 2007.

[89] [mercy] mercy. *Exploiting Uninitialized Data*, January 2006.

[90] [Microsoft 03] Microsoft Security Bulletin MS03-026, "Buffer Overrun In RPC Interface Could Allow Code Execution (823980)," September 2003.

[91] [Microsoft 07] C Language Reference, 2007.

[92] [Miller 99] Todd C. Miller and Theo de Raadt. strlcpy and strlcat - Consistent, Safe, String Copy and Concatenation. In Proceedings of the FREENIX Track, 1999 USENIX Annual Technical Conference.

[93] [Miller 04] Miller, Mark C., Reus, James F., Matzke, Robb P., Koziol, Quincey A., & Cheng, Albert P. "Smart Libraries: Best SQE Practices for Libraries with an Emphasis on Scientific Computing." *Proceedings of the Nuclear Explosives Code Developer's Conference*, December 2004.

[94] [MISRA 04] MISRA Limited. "MISRA C: 2004 Guidelines for the Use of the C Language in Critical Systems." Warwickshire, UK: MIRA Limited, October 2004 (ISBN 095241564X).

[95] [MIT 04] MIT. "MIT krb5 Security Advisory 2004-002, 2004.

[96] [MIT 05] MIT. "MIT krb5 Security Advisory 2005-003, 2005.

[97] [MITRE 07] MITRE. Common Weakness Enumeration, Draft 9, April 2008.

[98] [MSDN] Microsoft Developer Network.

[99] [Murenin 07] 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.

[100] [NAI 98] Network Associates Inc. Bugtraq: Network Associates Inc. Advisory (OpenBSD), 1998.

[101] [NASA-GB-1740.13] NASA Glenn Research Center, Office of Safety Assurance Technologies. *NASA Software Safety Guidebook* (NASA-GB-1740.13).

[102] [NIST 06] NIST. *SAMATE Reference Dataset*, 2006.

[103] [OpenBSD] Berkley Software Design, Inc. Manual Pages, June 2008.

[104] [Open Group 97a] The Open Group. *The Single UNIX® Specification, Version 2*, 1997.

[105] [Open Group 97b] The Open Group. *Go Solo 2---The Authorized Guide to Version 2 of the Single UNIX Specification*, May 1997.

[106] [Open Group 04] The Open Group and the IEEE. *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition*, 2004.

[107] [OWASP Double Free] Open Web Application Security Project, "Double Free."

[108] [OWASP Freed Memory] Open Web Application Security Project, "Using freed memory."

[109] [Pethia 03] Pethia, Richard D. "Viruses and Worms: What Can We Do About Them?" September 10, 2003.

[110] [Pfaff 04] Pfaff, Ken Thompson. "Casting (time_t)(-1)." *Google Groups comps.lang.c*, March 2, 2004.

[111] [Pike 93] Pike, Rob & Thompson, Ken. "Hello World." *Proceedings of the USENIX Winter 1993 Technical Conference*, San Diego, CA, January 25----50.

[112] [Plakosh 05] Plakosh, Dan. *Consistent Memory Management Conventions*, 2005.

[113] [Plum 85] Plum, Thomas. *Reliable Data Structures in C*. Kamuela, HI: Plum Hall, Inc., 1985 (ISBN 0-911537-04-X).

[114] [Plum 89] Plum, Thomas, & Saks, Dan. *C Programming Guidelines, 2nd ed*. Kamuela, HI: Plum Hall, 1989 (ISBN 0911537074).

[115] [Plum 91] Plum, Thomas. *C++ Programming*. Kamuela, HI: Plum Hall, 1991 (ISBN 0911537104).

[116] [Plum 08] Plum, Thomas. Static Assertions. June, 2008. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1330.pdf

[117] [Redwine 06] Redwine, Samuel T., Jr., ed. *Secure Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software Version 1.1*. U.S. Department of Homeland Security, September 2006. See Software Assurance Common Body of Knowledge on Build Security In.

[118] [RUS-CERT] RUS-CERT Advisory 2002-08:02, "Flaw in calloc and similar routines," 2002.

[119] [Saltzer 74] Saltzer, J. H. Protection and the Control of Information Sharing in Multics. *Communications of the ACM 17*, 7 (July 1974): 388---402.

[120] [Saltzer 75] Saltzer, J. H., & Schroeder, M. D. "The Protection of Information in Computer Systems." *Proceedings of the IEEE 63*, 9 (September 1975): 1278-1308.

[121] [Saks 99] Saks, Dan. "const T vs.T const." *Embedded Systems Programming*, February 1999, pp. 13-16.

[122] [Saks 00] Saks, Dan. "Numeric Literals." *Embedded Systems Programming*, September 2000.

[123] [Saks 01a] Saks, Dan. "Symbolic Constants." *Embedded Systems Design*, November 2001.

[124] [Saks 01b] Saks, Dan. "Enumeration Constants vs. Constant Objects." *Embedded Systems Design*, November 2001.

[125] [Saks 02] Saks, Dan. "Symbolic Constant Expressions." *Embedded Systems Design*, February 2002.

[126] [Saks 05] Saks, Dan. "Catching Errors Early with Compile-Time Assertions." *Embedded Systems Design*, June 2005.

[127] [Saks 07a] Saks, Dan. "Sequence Points" *Embedded Systems Design*, July 1, 2002.

[128] [Saks 07b] Saks, Dan. Bail, return, jump, or . . . throw?. *Embedded Systems Design*, March 2007.

[129] [Saks 08] Saks, Dan, & Dewhurst, Stephen C. "Sooner Rather Than Later: Static Programming Techniques for C++" (presentation, March 2008).

[130] [Schwarz 05] Schwarz, B., Wagner, Hao Chen, Morrison, D., West, G., Lin, J., & Tu, J. Wei. "Model checking an entire Linux distribution for security violations." *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005 (ISSN 1063-9527; ISBN 0-7695-2461-3).

[131] [Seacord 03] Seacord, Robert C., Plakosh, Daniel, & Lewis, Grace A. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, February 2003.

[132] [Seacord 05a] Seacord, Robert C. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005. See http://www.cert.org/books/secure-coding for news and errata.

[133] [Seacord 05b] Seacord, Robert C. "Managed String Library for C, C/C++." *Users Journal 23*, 10 (October 2005): 30---34.

[134] [Seacord 05c] Seacord, Robert C. *Variadic Functions: How they contribute to security vulnerabilities and how to fix them*. *Linux World Magazine*, November 2005.

[135] [Seacord 09] Seacord, Robert C. *The CERT C Secure Coding Standard*. Boston, MA: Addison-Wesley, 2009.

[136] [Secunia] Secunia Advisory SA10635, "HP-UX calloc Buffer Size Miscalculation Vulnerability," 2004.

[137] [SecurityFocus 07] SecurityFocus. "Linux Kernel Floating Point Exception Handler Local Denial of Service Vulnerability," 2001.

[138] [SecuriTeam 07] SecuriTeam. "Microsoft Visual C++ 8.0 Standard Library Time Functions Invalid Assertion DoS (Problem 3000)," February 13, 2007.

[139] [Sloss 04] Sloss, Andrew, Symes, Dominic, & Wright, Chris. *ARM System Developer's Guide*. San Francisco:Elsevier/Morgan Kauffman, 2004 (ISBN-10: 1558608745; ISBN-13: 978-1558608740).

[140] [Spinellis 06] Spinellis, Diomidis. *Code Quality: The Open Source Perspective*. Addison-Wesley, 2006.

[141] [Steele 77] Steele, G. L. "Arithmetic shifting considered harmful." *SIGPLAN Not.* 12, 11 (November 1977), 61-69.

[142] [Summit 95] Summit, Steve. *C Programming FAQs: Frequently Asked Questions*. Boston, MA: Addison-Wesley, 1995 (ISBN 0201845199).

[143] [Summit 05] Summit, Steve. *comp.lang.c Frequently Asked Questions*, 2005.

[144] [Sun] Sun Security Bulletin #00122, 1993.

[145] [Sun 05] C User's Guide. 819-3688-10. Sun Microsystems, Inc., 2005.

[146] [Sutter 04] Sutter, Herb & Alexandrescu, Andrei. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Boston, MA:Addison-Wesley Professional, 2004 (ISBN 0321113586).

[147] [van de Voort 07] van de Voort, Marco. Development Tutorial (a.k.a Build FAQ), January 29, 2007.

[148] [van Sprundel 06] van Sprundel, Ilja. Unusualbugs, 2006.

[149] [Viega 01] Viega, John. Protecting Sensitive Data in Memory, February 2001.

[150] [Viega 03] 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).

[151] [Viega 05] Viega, John. CLASP Reference Guide Volume 1.1. Secure Software, 2005.

[152] [VU#159523] Giobbi, Ryan. Vulnerability Note VU#159523, *Adobe Flash Player integer overflow vulnerability*, April 2008.

[153] [VU#162289] Dougherty, Chad. Vulnerability Note VU#162289, *gcc silently discards some wraparound checks*, April 2008.

[154] [VU#196240] Taschner, Chris & Manion, Art. Vulnerability Note VU#196240, *Sourcefire Snort DCE/RPC preprocessor does not properly reassemble fragmented packets*, 2007.

[155] [VU#286468] Burch, Hal. Vulnerability Note VU#286468, *Ettercap contains a format string error in the "curses_msg" function*, 2007.

[156] [VU#439395] Lipson, Howard. Vulnerability Note VU#439395, *Apache web server performs case sensitive filtering on Mac OS X HFS+ case insensitive filesystem,* 2001.

[157] [VU#551436] Giobbi, Ryan. Vulnerability Note VU#551436, *Mozilla Firefox SVG viewer vulnerable to buffer overflow,* 2007.

[158] [VU#568148] Finlay, Ian A. & Morda, Damon G. Vulnerability Note VU#568148, *Microsoft Windows RPC vulnerable to buffer overflow*, 2003.

[159] [VU#623332] Mead, Robert. Vulnerability Note VU#623332, *MIT Kerberos 5 contains double free vulnerability in "krb5_recvauth" function,* 2005.

[160] [VU#649732] Gennari, Jeff. Vulnerability Note VU#649732, *Samba AFS ACL Mapping VFS Plug-In Format String Vulnerability,* 2007.

[161] [VU#654390] Rafail, Jason A. Vulnerability Note VU#654390, *ISC DHCP contains C Includes that define vsnprintf to vsprintf creating potential buffer overflow conditions*, June 2004.

[162] [VU#743092] Rafail, Jason A. & Havrilla, Jeffrey S. Vulnerability Note VU#743092, *realpath(3) function contains off-by-one buffer overflow,* July 2003.

[163] [VU#834865] Gennari, Jeff. Vulnerability Note VU#834865, *Sendmail signal I/O race condition*, March 2008.

[164] [VU#837857] Dougherty, Chad. Vulnerability Note VU#837857, *SX.Org server fails to properly test for effective user ID*, August 2006.

[165] [VU#881872] Manion, Art & Taschner, Chris. Vulnerability Note VU#881872, *Sun Solaris telnet authentication bypass vulnerability,* 2007.

[166] [Warren 02] Warren, Henry S. *Hacker's Delight*. Boston, MA: Addison Wesley Professional, 2002 (ISBN 0201914654).

[167] [Wheeler 03] Wheeler, David. Secure Programming for Linux and Unix HOWTO, v3.010, March 2003.

[168] [Wheeler 04] Wheeler, David. *Secure programmer: Call components safely*. December 2004.

[169] [Wojtczuk 08] Wojtczuk, Rafal. "Analyzing the Linux Kernel vmsplice Exploit." McAfee Avert Labs Blog, February 13, 2008.

[170] [Yergeau 98] Yergeau, F. RFC 2279 - UTF-8, a transformation format of ISO 10646, January 1998.

[171] [Zalewski 01] Zalewski, Michal. *Delivering Signals for Fun and Profit: Understanding, exploiting and preventing signal-handling related vulnerabilities*, May 2001.