Preface

The CERT® C Coding Standard, Second Edition, provides rules for coding in the C programming language. The goal of these rules is to develop safe, reliable, and secure systems, for example, by eliminating undefined behaviors that can lead to unexpected program behaviors and exploitable vulnerabilities. Conformance to the coding rules defined in this standard are necessary (but not sufficient) to ensure the safety, reliability, and security of software systems developed in the C programming language. It is also necessary, for example, to have a safe and secure design. Safety-critical systems typically have stricter requirements than are imposed by this coding standard, for example, requiring that all memory be statically allocated. However, the application of this coding standard will result in high-quality systems that are reliable, robust, and resistant to attack.

Each rule consists of a *title*, a *description*, and *noncompliant code examples* and *compliant solutions*. The title is a concise, but sometimes imprecise, description of the rule. The description specifies the normative requirements of the rule. The noncompliant code examples are examples of code that would constitute a violation of the rule. The accompanying compliant solutions demonstrate equivalent code that does not violate the rule or any other rules in this coding standard.

A well-documented and enforceable coding standard is an essential element of coding in the C programming language. Coding standards encourage programmers to follow a uniform set of rules determined by the requirements of the project and organization rather than by the programmer's familiarity.

Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes).

CERT's coding standards are being widely adopted by industry. Cisco Systems, Inc., announced its adoption of the CERT C Secure Coding Standard as a baseline programming standard in its product development in October 2011 at Cisco's annual SecCon conference. Recently, Oracle has integrated all of CERT's secure coding standards into its existing Secure Coding Standards. Note that this adoption is the most recent step of a long collaboration: CERT and Oracle previously worked together in authoring *The CERT® Oracle Secure Coding Standard for Java* (Addison-Wesley, 2011).

Scope

The CERT® C Coding Standard, Second Edition, was developed specifically for versions of the C programming language defined by

- ISO/IEC 9899:2011, *Programming Languages—C, Third Edition* [ISO/IEC 9899:2011]
- ISO/IEC 9899:2011/Cor.1:2012, Technical Corrigendum 1

The CERT® C Coding Standard, Second Edition, updates and replaces *The CERT® C Secure Coding Standard* (Addison-Wesley, 2008). The scope of the first edition of this book is C99 (the second edition of the C Standard) [ISO/IEC 9899:1999]. Although the rules in this book were developed for C11, they can also be applied to earlier versions of the C programming language, including C99. Variations between versions of the C Standard that would affect the proper application of these rules are noted where applicable.

Most rules have a noncompliant code example that is a C11-conforming program to ensure that the problem identified by the rule is within the scope of the standard. However, the best solutions to coding problems are often platform specific. In many cases, this standard provides appropriate compliant solutions for both POSIX and Windows operating systems. Language and library extensions that have been published as ISO/IEC technical reports or technical specifications are frequently given precedence, such as those described by ISO/IEC TR 24731-2, *Extensions to the C Library—Part II: Dynamic Allocation Functions* [ISO/IEC TR 24731-2:2010]. In many cases, compliant solutions are also provided for specific platforms such as Linux or OpenBSD. Occasionally, interesting or illustrative implementation-specific behaviors are described.

Rationale

A coding standard for the C programming language can create the highest value for the longest period of time by focusing on the C Standard (C11) and the relevant post-C11 technical reports.

The C Standard documents existing practice where possible. That is, most features must be tested in an implementation before being included in the standard. *The CERT® C Coding Standard, Second Edition,* has a different purpose: to establish a set of best practices, which sometimes requires introducing new practices that may not be widely known or used when existing practices are inadequate. To put it a different way, *The CERT® C Coding Standard, Second Edition,* attempts to drive change rather than just document it.

For example, the optional but normative Annex K, "Bounds-Checking Interfaces," introduced in C11, is gaining support but at present is implemented by only a few vendors. It introduces functions such as `memcpy_s()`, which serve the purpose of security by adding the destination buffer size to the API. A forward-looking document could not reasonably ignore these functions simply because they are not yet widely implemented. The base C Standard is more widely implemented than Annex K, but even if it were not, it is the direction in which the industry is moving. Developers of new C code, especially, need guidance that is usable on and makes the best use of the compilers and tools that are now being developed.

Some vendors have extensions to C, and some also have implemented only part of the C Standard before stopping development. Consequently, it is not possible to back up and discuss only C99, C95, or C90. The vendor support equation is too complicated to draw a line and say that a certain compiler supports exactly a certain standard. Whatever demarcation point is selected, different vendors are on opposite sides of it for different parts of the language. Supporting all possibilities would require testing the cross-product of each compiler with each language feature. Consequently, we have selected a demarcation point that is the most recent in time so that the rules defined by the standard will be applicable for as long as possible. As a result of the variations in support, source-code portability is enhanced when the programmer uses only the features specified by C99. This is one of many trade-offs between security and portability inherent to C language programming.

The value of forward-looking information increases with time before it starts to decrease. The value of backward-looking information starts to decrease immediately.

For all of these reasons, the priority of this standard is to support new code development using C11 and the post-C11 technical reports that have not been incorporated into the C Standard. A close-second priority is supporting remediation of old code using C99 and the technical reports.

This coding standard does make contributions to support older compilers when these contributions can be significant and doing so does not compromise other priorities. The intent is not to capture all deviations from the C Standard but to capture only a few important ones.

Issues Not Addressed

A number of issues are not addressed by this coding standard.

Coding Style. Coding style issues are subjective, and it has proven impossible to develop a consensus on appropriate style guidelines. Consequently, *The CERT® C Coding Standard, Second Edition,* does not require the enforcement of any particular coding style but only suggests that development organizations define or adopt style guidelines and apply these guidelines consistently. The easiest way to apply a coding style consistently is to use a code-formatting tool. Many interactive development environments (IDEs) provide such capabilities.

Controversial Rules. In general, the CERT coding standards try to avoid the inclusion of controversial rules that lack a broad consensus.

Who Should Read This Book

The CERT® C Coding Standard, Second Edition, is primarily intended for developers of C language programs but may also be used by software acquirers to define the requirements for bespoke software. This book is of particular interest to developers who are interested in building high-quality systems that are reliable, robust, and resistant to attack.

While not intended for C++ programmers, this book may also be of some value because the vast majority of issues identified for C language programs are also issues in C++ programs, although in many cases the solutions are different.

History

The idea of a CERT secure coding standard arose at the Spring 2006 meeting of the C Standards Committee (more formally, ISO/IEC JTC1/SC22/WG14) in Berlin, Germany [Seacord 2013a]. The C Standard is an authoritative document, but its audience is primarily compiler implementers, and, as noted by

many, its language is obscure and often impenetrable. A secure coding standard would be targeted primarily toward C language programmers and would provide actionable guidance on how to code securely in the language.

The CERT C Secure Coding Standard was developed on the CERT Secure Coding wiki (http://www.securecoding.cert.org) following a community-based development process. Experts from the community, including members of the WG14 C Standards Committee, were invited to contribute and were provided with edit privileges on the wiki. Members of the community can register for a free account on the wiki and comment on the coding standards and the individual rules. Reviewers who provide high-quality comments are frequently extended edit privileges so that they can directly contribute to the development and evolution of the coding standard. Today, the CERT Secure Coding wiki has 1,576 registered contributors.

This wiki-based community development process has many advantages. Most important, it engages a broad group of experts to form a consensus opinion on the content of the rules. The main disadvantage of developing a secure coding standard on a wiki is that the content is constantly evolving. This instability may be acceptable if you want the latest information and are willing to entertain the possibility that a recent change has not yet been fully vetted. However, many software development organizations require a static set of rules and recommendations that they can adopt as requirements for their software development process. Toward this end, a stable snapshot of the CERT C Secure Coding Standard was produced after two and a half years of community development and published as *The CERT® C Secure Coding Standard*. With the production of the manuscript for the book in June 2008, version 1.0 (the book) and the wiki versions of the secure coding standard began to diverge.

The CERT C secure coding guidelines were first reviewed by WG14 at the London meeting in April 2007 and again at the Kona, Hawaii, meeting in August 2007.

The topic of whether INCITS PL22.11 should submit the CERT C Secure Coding Standard to WG14 as a candidate for publication as a type 2 or type 3 technical report was discussed at the J11/U.S. TAG Meeting, April 15, 2008, as reported in the minutes. J11 is now Task Group PL22.11, Programming Language C, and this technical committee is the U.S. Technical Advisory Group to ISO/IEC JTC 1 SC22/WG14. A straw poll was taken on the question, "Who has time to work on this project?" for which the vote was 4 (has time) to 12 (has no time). Some of the feedback we received afterwards was that although the CERT C Secure Coding Standard was a strong set of guidelines that had been developed with input from many of the technical experts at WG14 and had been reviewed by WG14 on several occasions, WG14 was not normally in the business of "blessing" guidance to developers. However, WG14 was certainly in the business of defining normative requirements for tools such as compilers.

Armed with this knowledge, we proposed that WG14 establish a study group to consider the problem of producing analyzable secure coding guidelines for the C language. The study group first met on October 27, 2009. CERT contributed an automatically enforceable subset of the C secure coding rules to ISO/IEC for use in the standardization process.

Participants in the study group included analyzer vendors such as Coverity, Fortify, GammaTech, Gimpel, Klocwork, and LDRA; security experts; language experts; and consumers. A new work item to develop and publish ISO/IEC TS 17961, C Secure Coding Rules, was approved for WG14 in March 2012, and the study group concluded. Roberto Bagnara, the Italian National Body representative to WG 14, later joined the WG14 editorial committee. *ISO/IEC TS 17961:2013(E), Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules* [ISO/IEC TS 17961:2013] was officially published in November 2013 and is available for purchase at the ISO store (http://www.iso.org/iso/catalogue_detail.htm?csnumber=61134).

ISO/IEC TS 17961 C Secure Coding Rules

The purpose of ISO/IEC TS 17961 is to establish a baseline set of requirements for analyzers, including static analysis tools and C language compilers, to be applied by vendors that wish to diagnose insecure code beyond the requirements of the language standard. All rules are meant to be enforceable by static analysis. The criterion for selecting these rules is that analyzers that implement these rules must be able to effectively discover secure coding errors without generating excessive false positives.

To date, the application of static analysis to security has been performed in an ad hoc manner by different vendors, resulting in nonuniform coverage of significant security issues. ISO/IEC TS 17961 enumerates secure coding rules and requires analysis engines to diagnose violations of these rules as a matter of conformance to the specification. These rules may be extended in an implementation-dependent manner, which provides a minimum coverage guarantee to customers of any and all conforming static analysis implementations.

ISO/IEC TS 17961 specifies rules for secure coding in the C programming language and includes code examples for each rule. Noncompliant code examples demonstrate language constructs that have weaknesses with potentially exploitable security implications; such examples are expected to elicit a diagnostic from a conforming analyzer for the affected language construct. Compliant examples are expected not to elicit a diagnostic. ISO/IEC TS 17961 does not specify the mechanism by which these rules are enforced or any particular coding style to be enforced.

Table P–1. ISO/IEC TS 17961 Compared with Other Standards

Coding Standard	C Standard	Security Standard	Safety Standard	International Standard	Whole Language
CWE	None/all	Yes	No	No	N/A
MISRA C2	C89	No	Yes	No	No
MISRA C3	C99	No	Yes	No	No
CERT C99	C99	Yes	No	No	Yes
CERT C11	C11	Yes	Yes	No	Yes
ISO/IEC TS 17961	C11	Yes	No	Yes	Yes

Table P–1 shows how ISO/IEC TS 17961 relates to other standards and guidelines. Of the publications listed, ISO/IEC TS 17961 is the only one for which the immediate audience is analyzers and not developers.

A conforming analyzer must be capable of producing a diagnostic for each distinct rule in the technical specification upon detecting a violation of that rule in isolation. If the same program text violates multiple rules simultaneously, a conforming analyzer may aggregate diagnostics but must produce at least one diagnostic. The diagnostic message might be of the form

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

ISO/IEC TS 17961 does not require an analyzer to produce a diagnostic message for any violation of any syntax rule or constraint specified by the C Standard. Conformance is defined only with respect to source code that is visible to the analyzer. Binary-only libraries, and calls to them, are outside the scope of these rules.

An interesting aspect of the technical specification is the portability assumptions, known within the group as the "San Francisco rule" because the assumptions evolved at a meeting hosted by Coverity at its headquarters. The San Francisco rule states that a conforming analyzer must be able to diagnose violations of guidelines for at least one C implementation but does not need to diagnose a rule violation if the result is documented for the target implementation and does not cause a security flaw. Variations in quality of implementation permit an analyzer to produce diagnostics concerning portability issues. For example, the following program fragment can produce a diagnostic, such as the mismatch between `%d` and `long int`:

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

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

In addition to other goals already stated, *The CERT® C Coding Standard, Second Edition,* has been updated for consistency with ISO/IEC TS 17961. Although the documents serve different audiences, consistency between the documents should improve the ability of developers to use ISO/IEC TS 17961–conforming analyzers to find violations of rules from this coding standard.

The Secure Coding Validation Suite (https://github.com/SEI-CERT/scvs) is a set of tests developed by CERT to validate the rules defined in ISO/IEC TS 17961. These tests are based on the examples in this technical specification and are distributed with a BSD-style license.

Tool Selection and Validation

Although rule checking can be performed manually, with increasing program size and complexity, it rapidly becomes infeasible. For this reason, the use of static analysis tools is recommended.

When choosing a compiler (which should be understood to include the linker), a C-compliant compiler should be used whenever possible. A conforming implementation will produce at least one diagnostic message if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as *undefined* or *implementation-defined*. It is also likely that any analyzers you may use assume a C-compliant compiler.

When choosing a source code analysis tool, it is clearly desirable that the tool be able to enforce as many of the recommendations on the wiki as possible. Not all recommendations are enforceable; some are strictly meant to be informative.

Although CERT recommends the use of an ISO/IEC TS 17961–conforming analyzer, the Software Engineering Institute, as a federally funded research and development center (FFRDC), is not in a position to endorse any particular vendor or tool. Vendors are encouraged to develop conforming analyzers, and users of this coding standard are free to evaluate and select whichever analyzers best suit their purposes.

Completeness and Soundness

It should be recognized that, in general, determining conformance to coding rules is computationally undecidable. The precision of static analysis has practical limitations. For example, the halting theorem of computer science

states that programs exist in which exact control flow cannot be determined statically. Consequently, any property dependent on control flow—such as halting—may be indeterminate for some programs. A consequence of undecidability is that it may be impossible for any tool to determine statically whether a given rule is satisfied in specific circumstances. The widespread presence of such code may also lead to unexpected results from an analysis tool.

However checking is performed, the analysis may generate

- *False negatives:* Failure to report a real flaw in the code is usually regarded as the most serious analysis error, as it may leave the user with a false sense of security. Most tools err on the side of caution and consequently generate false positives. However, in some cases, it may be deemed better to report some high-risk flaws and miss others than to overwhelm the user with false positives.
- *False positives:* The tool reports a flaw when one does not exist. False positives may occur because the code is too complex for the tool to perform a complete analysis. The use of features such as function pointers and libraries may make false positives more likely.

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

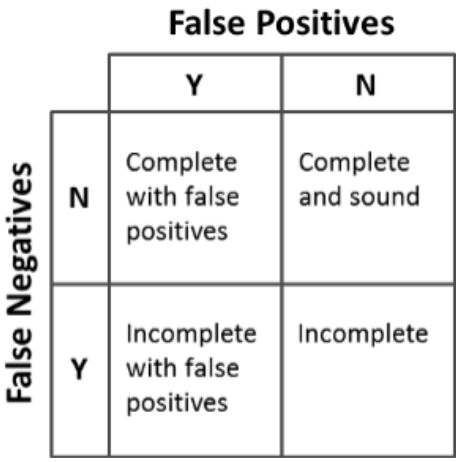

Figure P–1. False-negative and false-positive possibilities

Compilers and source code analysis tools are *trusted* processes, meaning that a degree of reliance is placed on the output of the tools. Accordingly, developers must ensure that this trust is not misplaced. Ideally, trust should be achieved by the tool supplier running appropriate validation tests such as the Secure Coding Validation Suite.

False Positives

Although many rules list common exceptions, it is difficult if not impossible to develop a complete list of exceptions for each guideline. Consequently, it is important that source code comply with the *intent* of each rule and that tools, to the greatest extent possible, minimize false positives that do not violate the intent of the rule. The degree to which tools minimize false-positive diagnostics is a quality-of-implementation issue.

Taint Analysis

Taint and Tainted Sources

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

Restricted Sinks

Operands and arguments whose domain is a subset of the domain described by their types are called restricted sinks. Any integer operand used in a pointer arithmetic operation is a restricted sink for that operand. Certain parameters of certain library functions are restricted sinks because these functions perform address arithmetic with these parameters, or control the allocation of

a resource, or pass these parameters on to another restricted sink. All string input parameters to library functions are restricted sinks because it is possible to pass in a character sequence that is not null terminated. The exceptions are input parameters to `strncpy()` and `strncpy_s()`, which explicitly allow the source character sequence not to be null terminated.

Propagation

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

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

Sanitization

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

In general, sanitization cannot be recognized exactly using static analysis. Analyzers that perform taint analysis usually provide some extralinguistic mechanism to identify sanitizing functions that sanitize an argument (passed by address) in place, return a sanitized version of an argument, or return a status code indicating whether the argument is in the required domain. Because such extralinguistic mechanisms are outside the scope of this coding standard, we use a set of rudimentary definitions of sanitization that is likely to recognize real sanitization but might cause nonsanitizing or ineffectively sanitizing code to be misconstrued as sanitizing. The following definition of sanitization presupposes that the analysis is in some way maintaining a set of constraints on each value encountered as the simulated execution progresses: a given path through the code sanitizes a value with respect to a given

restricted sink if it restricts the range of that value to a subset of the defined domain of the restricted sink type. For example, sanitization of signed integers with respect to an array index operation must restrict the range of that integer value to numbers between zero and the size of the array minus one.

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

Rules versus Recommendations

This book contains 98 coding *rules*. The CERT Coding Standards wiki also has 178 recommendations at the time of writing. Rules are meant to provide normative requirements for code, whereas recommendations are meant to provide guidance that, when followed, should improve the safety, reliability, and security of software systems. However, a violation of a recommendation does not necessarily indicate the presence of a defect in the code.

Rules and recommendations are collectively referred to as *guidelines*. Rules must meet the following criteria:

1. Violation of the guideline is likely to result in a defect that may adversely affect the safety, reliability, or security of a system, for example, by introducing a security flaw that may result in an exploitable vulnerability.
2. The guideline does not rely on source code annotations or assumptions of programmer intent.
3. Conformance to the guideline can be determined through automated analysis (either static or dynamic), formal methods, or manual inspection techniques.

Recommendations are suggestions for improving code quality. Guidelines are defined to be recommendations when all of the following conditions are met:

1. Application of a guideline is likely to improve the safety, reliability, or security of software systems.
2. One or more of the requirements necessary for a guideline to be considered a rule cannot be met.

Figure P–2 shows how the 98 rules and 178 recommendations are organized.

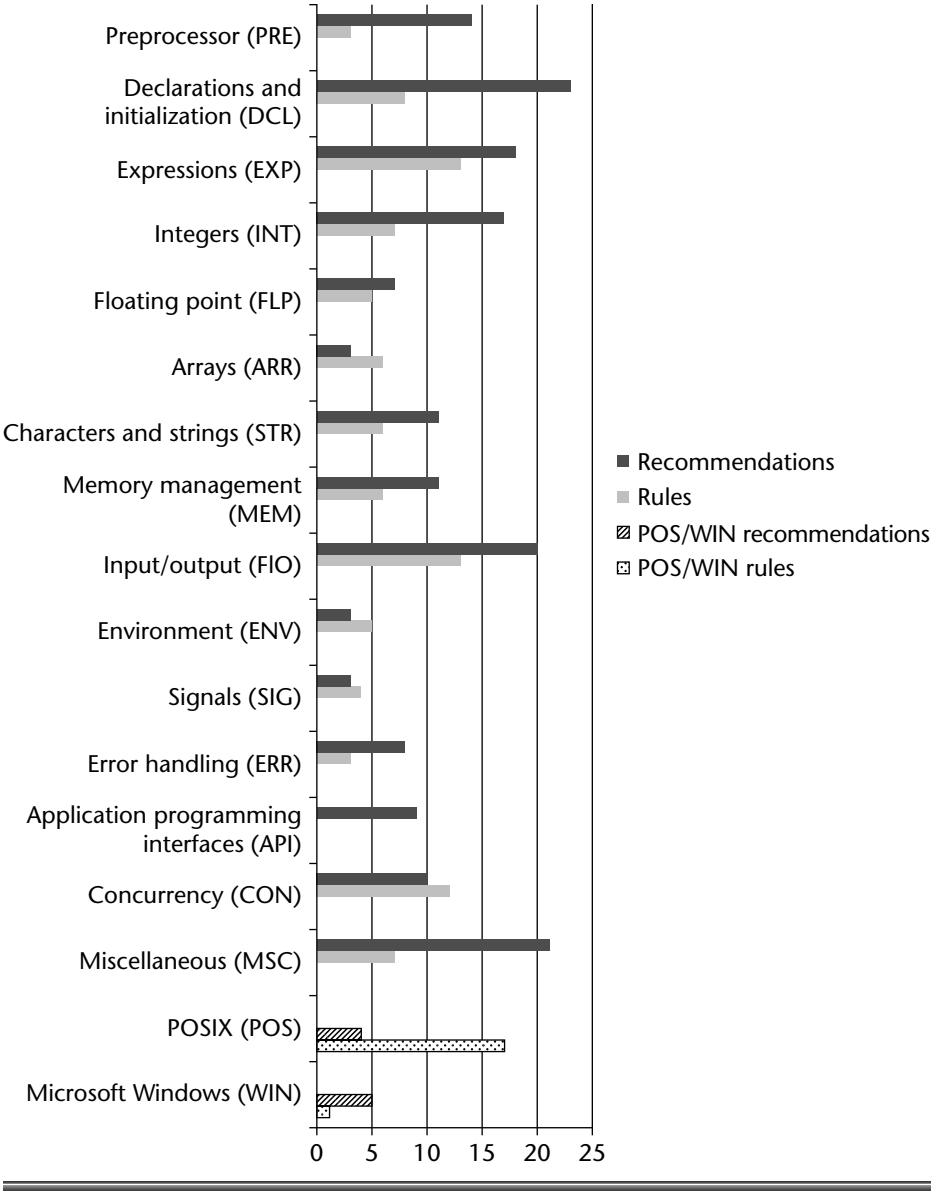

Figure P–2. CERT C coding guidelines

The wiki also contains two platform-specific annexes at the time of writing, one for POSIX and one for Windows, which have been omitted from this book because they are not part of the core standard.

The set of recommendations that a particular development effort adopts depends on the requirements of the final software product. Projects with stricter requirements may decide to dedicate more resources to ensuring the safety, reliability, and security of a system and consequently are likely to adopt a broader set of recommendations.

Usage

The rules in this standard may be extended with organization-specific rules. However, the rules in the standard must be obeyed to claim conformance with the standard.

Training may be developed to educate software professionals regarding the appropriate application of coding standards. After passing an examination, these trained programmers may also be certified as coding professionals. For example, the Software Developer Certification (SDC) is a credentialing program developed at Carnegie Mellon University. The SDC uses authentic examination to

1. Identify job candidates with specific programming skills
2. Demonstrate the presence of a well-trained software workforce
3. Provide guidance to educational and training institutions

Once a coding standard has been established, tools and processes can be developed or modified to determine conformance with the standard.

Conformance Testing

To ensure that the source code conforms to this coding standard, it is necessary to have measures in place that check for rule violations. The most effective means of achieving this goal is to use one or more ISO/IEC TS 17961–conforming analyzers. Where a rule cannot be checked by a tool, a manual review is required.

The Source Code Analysis Laboratory (SCALe) provides a means for evaluating the conformance of software systems against this and other coding standards. CERT coding standards provide a normative set of rules against which software systems can be evaluated. Conforming software systems should demonstrate improvements in the safety, reliability, and security over nonconforming systems.

The SCALe team at the CERT Division of Carnegie Mellon University's Software Engineering Institute analyzes a developer's source code and provides a detailed report of findings to guide the code's repair. After the developer has addressed these findings and the SCALe team determines that the product version conforms to the standard, the CERT Program issues the developer a certificate and lists the system in a registry of conforming systems. This report details the SCALe process and provides an analysis of selected software systems.

Conformance

Conformance to *The CERT® C Coding Standard* requires that the code not contain any violations of the rules specified in this book. If an exceptional condition is claimed, the exception must correspond to a predefined exceptional condition, and the application of this exception must be documented in the source code.

Conformance with the recommendations on the wiki is not necessary to claim conformance with *The CERT® C Coding Standard*. Conformance to the recommendations will, in many cases, make it easier to conform to the rules; eliminating many potential sources of defects.

Deviation Procedure

Strict adherence to all rules is unlikely and, consequently, deviations associated with specific rule violations are necessary. Deviations can be used in cases where a true-positive finding is uncontested as a rule violation but the code is nonetheless determined to be correct. An uncontested true-positive finding may be the result of a design or architecture feature of the software or may occur for a valid reason that was unanticipated by the coding standard. In this respect, the deviation procedure allows for the possibility that coding rules are overly strict [Seacord 2012].

Deviations are not granted for reasons of performance or usability. A software system that successfully passes conformance testing must not contain defects or exploitable vulnerabilities. Deviation requests are evaluated by the lead assessor, and if the developer can provide sufficient evidence that deviation will not result in a vulnerability, the deviation request is accepted. Deviations are used infrequently because it is almost always easier to fix a coding error than it is to provide an argument that the coding error does not result in a vulnerability.

System Qualities

The goal of this coding standard is to produce safe, reliable, and secure systems. Additional requirements might exist for safety-critical systems, such as the absence of dynamic memory allocation. Other software quality

attributes of interest include portability, usability, availability, maintainability, readability, and performance.

Many of these attributes are interrelated in interesting ways. For example, readability is an attribute of maintainability; both are important for limiting the introduction of defects during maintenance that can result in security flaws or reliability issues. In addition, readability aids code inspection by safety officers. Reliability and availability require proper resources management, which also contributes to the safety and security of the system. System attributes such as performance and security are often in conflict, requiring trade-offs to be considered.

How This Book Is Organized

This book is organized into 14 chapters containing rules in specific topic areas, three appendices, a bibliography, and an index. The first appendix is a glossary of terms used through this book. Terms that are listed in the glossary are printed in **bold font** the first time they appear and then in normal font in subsequent appearances. The second appendix lists the undefined behaviors from the C Standard, Annex J, J.2 [ISO/IEC 9899:2011], numbered and classified for easy reference. These numbered undefined behaviors are referenced frequently from the rules. The third appendix contains unspecified behaviors from the C Standard, Annex J, J.1 [ISO/IEC 9899:2011]. These unspecified behaviors are occasionally referenced from the rules as well. The bibliography is a compendium of the small bibliography sections from each rule as well as other references cited throughout the book.

Most rules have a consistent structure. Each rule in this standard has a unique *identifier,* which is included in the title. The title and the introductory paragraphs define the rule and are typically followed by one or more pairs of *noncompliant code examples* and *compliant solutions*. Each rule also includes a *risk assessment*, *related guidelines*, and a *bibliography* (where applicable). Rules may also include a table of *related vulnerabilities*. Recommendations on the CERT Coding Standards wiki are organized in a similar fashion.

Identifiers

Each rule and recommendation is given a unique identifier, which consists of three parts:

- A three-letter mnemonic representing the section of the standard
- A two-digit numeric value in the range of 00 to 99
- The letter *C* indicating that this is a C language guideline

The three-letter mnemonic is used to group similar coding practices and to indicate to which category a coding practice belongs.

The numeric value is used to give each coding practice a unique identifier. Numeric values in the range of 00 to 29 are reserved for recommendations, and values in the range of 30 to 99 are reserved for rules. Rules and recommendations are frequently referenced from the rules in this book by their identifier and title. Rules can be found in the book's table of contents, whereas recommendations can be found only on the wiki.

Noncompliant Code Examples and Compliant Solutions

Noncompliant code examples illustrate code that violates the guideline under discussion. It is important to note that these are only examples, and eliminating all occurrences of the example does not necessarily mean that the code being analyzed is now compliant with the guideline.

Noncompliant code examples are typically followed by compliant solutions, which show how the noncompliant code example can be recoded in a secure, compliant manner. Except where noted, noncompliant code examples should contain violations only of the rule under discussion. Compliant solutions should comply with all secure coding rules but may on occasion fail to comply with a recommendation.

Exceptions

Any rule or recommendation may specify a small set of exceptions detailing the circumstances under which the guideline is not necessary to ensure the safety, reliability, or security of software. Exceptions are informative only and are not required to be followed.

Risk Assessment

Each guideline in *The CERT® C Coding Standard, Second Edition,* contains a risk assessment section that attempts to provide software developers with an indication of the potential consequences of not addressing violations of a particular rule in their code (along with some indication of expected remediation costs). This information may be used to prioritize the repair of rule violations by a development team. The metric is designed primarily for remediation projects. It is generally assumed that new code will be developed to be compliant with the entire coding standard and applicable recommendations.

Each rule and recommendation has an assigned *priority*. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [IEC 60812]. Three values are assigned for each rule on a scale of 1 to 3 for severity, likelihood, and remediation cost.

- **Severity**—How serious are the consequences of the rule being ignored?

Value	Meaning	Examples of Vulnerability
1	Low	Denial-of-service attack, abnormal termination
2	Medium	Data integrity violation, unintentional information disclosure
3	High	Run arbitrary code

- **Likelihood**—How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

Value	Meaning
1	Unlikely
2	Probable
3	Likely

- **Remediation Cost**—How expensive is it to comply with the rule?

Value	Meaning	Detection	Correction
1	High	Manual	Manual
2	Medium	Automatic	Manual
3	Low	Automatic	Automatic

The three values are then multiplied together for each rule. This product provides a measure that can be used in prioritizing the application of the rules. The products range from 1 to 27, although only the following 10 distinct values are possible: 1, 2, 3, 4, 6, 8, 9, 12, 18, and 27. Rules and recommendations with a priority in the range of 1 to 4 are Level 3 rules, 6 to 9 are Level 2, and 12 to 27 are Level 1. The following are possible interpretations of the priorities and levels:

Level	Priorities	Possible Interpretation
L1	12, 18, 27	High severity, likely, inexpensive to repair
L2	6, 8, 9	Medium severity, probable, medium cost to repair
L3	1, 2, 3, 4	Low severity, unlikely, expensive to repair

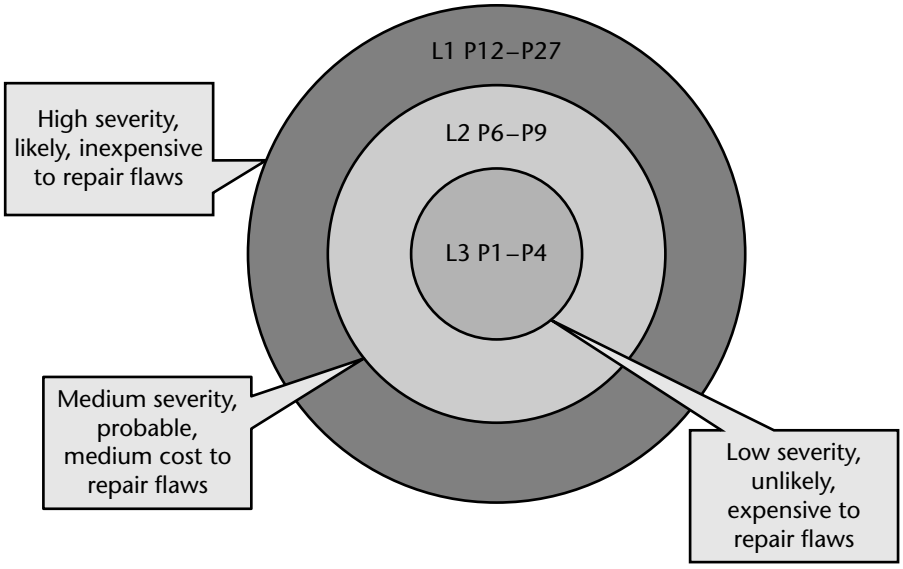

Figure P–3. Levels of compliance

Specific projects may begin remediation by implementing all rules at a particular level before proceeding to lower priority rules, as shown in Figure P–3.

Automated Detection

On the wiki, both rules and recommendations frequently have sections that describe automated detection. These sections provide additional information on analyzers that can automatically diagnose violations of coding guidelines. Most automated analyses for the C programming language are neither sound nor complete, so the inclusion of a tool in this section typically means that the tool can diagnose some violations of this particular rule. Although the Secure Coding Validation Suite can be used to test the ability of analyzers to diagnose violations of rules from ISO/IEC TS 17961, no currently available conformance test suite can assess the ability of analyzers to diagnose violations of the rules in this book. Consequently, the information in automated detection sections on the wiki may be

- Provided by the vendors
- Determined by CERT by informally evaluating the analyzer
- Determined by CERT by reviewing the vendor documentation

Where possible, we try to reference the exact version of the tool for which the results were obtained. Because these tools evolve continuously, this information can rapidly become dated and obsolete. Consequently, this information has been omitted from this book and is maintained only on the wiki.

Related Vulnerabilities

The related vulnerabilities sections on the wiki contain a link to search for related vulnerabilities on the CERT Web site. Whenever possible, CERT Vulnerability Notes are tagged with a keyword corresponding to the unique ID of the coding guideline. This search provides you with an up-to-date list of real-world vulnerabilities that have been determined to be at least partially caused by a violation of this specific guideline. These vulnerabilities are labeled as such only when the vulnerability analysis team at the CERT/CC is able to evaluate the source code and precisely determine the cause of the vulnerability. Because many vulnerability notes refer to vulnerabilities in closed-source software systems, it is not always possible to provide this additional analysis. Consequently, the related vulnerabilities field tends to be somewhat sparsely populated.

To find the latest list of related vulnerabilities, enter the following URL:

https://www.kb.cert.org/vulnotes/bymetric?searchview&query=FIELD+KEYWORDS+contains+XXXNN-X

where XXXNN-X is the ID of the rule or recommendation for which you are searching.

Specific vulnerability (VU) identifiers and common vulnerabilities and exposures (CVE) identifiers are referenced throughout this book. You can create a unique URL to get more information on specific vulnerabilities by appending the relevant ID to the end of a fixed string. For example, to find more information about

- *VU#551436*, "Mozilla Firefox SVG viewer vulnerable to integer overflow," you can append 551436 to https://www.kb.cert.org/vulnotes/id/ and enter the resulting URL in your browser: https://www.kb.cert.org/vulnotes/id/551436
- *CVE-2006-1174*, you can append CVE-2006-1174 to http://cve.mitre.org/cgi-bin/cvename.cgi?name= and enter the resulting URL in your browser: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1174

Related vulnerability sections are included only for specific rules in this book, when the information is both relevant and interesting.

Related Guidelines

This section contains links to guidelines in related standards, technical specifications, and guideline collections such as *Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules* [ISO/IEC TS 17961:2013]; *Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use* [ISO/IEC TR 24772:2013]; *MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems* [MISRA C:2012]; and CWE IDs in MITRE's Common Weakness Enumeration (CWE) [MITRE 2013].

You can create a unique URL to get more information on CWEs by appending the relevant ID to the end of a fixed string. For example, to find more information about CWE-192, "Integer Coercion Error," you can append 192.html to http://cwe.mitre.org/data/definitions/ and enter the resulting URL in your browser: http://cwe.mitre.org/data/definitions/192.html.

The other referenced technical specifications, technical reports, and guidelines are commercially available.

Bibliography

Most rules have a small bibliography section that lists documents and sections in these documents that provide information relevant to the rule.

Automatically Generated Code

If a code-generating tool is to be used, it is necessary to select an appropriate tool and undertake validation. Adherence to the requirements of this document may provide one criterion for assessing a tool.

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-coded 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 have the following properties:

- Readability
- Program comprehension

These requirements 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 do not apply to code that is tool generated and tool maintained. Tool-generated, tool-maintained code can impose consistent constraints that ensure the safety of some constructs that are risky in hand-generated code.

Government Regulations

Developing software to secure coding rules is a good idea and is increasingly a requirement. The National Defense Authorization Act for Fiscal Year 2013, Section 933, "Improvements in Assurance of Computer Software Procured by the Department of Defense," requires evidence that government software development and maintenance organizations and contractors are conforming, in computer software coding, to approved secure coding standards of the Department of Defense (DoD) during software development, upgrade, and maintenance activities, including through the use of inspection and appraisals.

DoD acquisition programs are specifying *The Application Security and Development Security Technical Implementation Guide (STIG)*, Version 2, Release 1 [DISA 2008] in requests for proposal (RFPs). Section 2.1.5, "Coding Standards," requires that "the Program Manager will ensure the development team follows a set of coding standards."

The proper application of this standard would enable a system to comply with the following requirements from the *Application Security and Development STIG* [DISA 2008]:

- **(APP2060.1: CAT II)** The Program Manager will ensure the development team follows a set of coding standards.
- **(APP2060.2: CAT II)** The Program Manager will ensure the development team creates a list of unsafe functions to avoid and document this list in the coding standards.
- **(APP3550: CAT I)** The Designer will ensure the application is not vulnerable to integer arithmetic issues.
- **(APP3560: CAT I)** The Designer will ensure the application does not contain format string vulnerabilities.
- **(APP3570: CAT I)** The Designer will ensure the application does not allow Command Injection.
- **(APP3590.1: CAT I)** The Designer will ensure the application does not have buffer overflows.
- **(APP3590.2: CAT I)** The Designer will ensure the application does not use functions known to be vulnerable to buffer overflows.
- **(APP3590.3: CAT II)** The Designer will ensure the application does not use signed values for memory allocation where permitted by the programming language.
- **(APP3600: CAT II)** The Designer will ensure the application has no canonical representation vulnerabilities.
- **(APP3630.1: CAT II)** The Designer will ensure the application is not vulnerable to race conditions.
- **(APP3630.2: CAT III)** The Designer will ensure the application does not use global variables when local variables could be used.

Training programmers and software testers will satisfy the following requirements:

- **(APP2120.3: CAT II)** The Program Manager will ensure developers are provided with training on secure design and coding practices on at least an annual basis.
- **(APP2120.4: CAT II)** The Program Manager will ensure testers are provided annual training.

- (**APP2060.3: CAT II**) The Designer will follow the established coding standards established for the project.
- (**APP2060.4: CAT II**) The Designer will not use unsafe functions documented in the project coding standards.
- (**APP5010: CAT III**) The Test Manager will ensure at least one tester is designated to test for security flaws in addition to functional testing.

Chapter 3

Expressions (EXP)

EXP30-C. Do not depend on the order of evaluation for side effects

Evaluation of an expression may produce side effects. At specific points during execution, known as **sequence points**, all side effects of previous evaluations are complete, and no side effects of subsequent evaluations have yet taken place. Do not depend on the order of evaluation for side effects unless there is an intervening sequence point.

The C Standard, 6.5, paragraph 2 [ISO/IEC 9899:2011], states:

> If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

This requirement must be met for each allowable ordering of the subexpressions of a full expression; otherwise, the behavior is undefined (see undefined behavior 35 in Appendix B.)

The following sequence points are defined in the C Standard, Annex C [ISO/IEC 9899:2011]:

- Between the evaluations of the function designator and actual arguments in a function call and the actual call
- Between the evaluations of the first and second operands of the following operators:
  - Logical AND: `&&`
  - Logical OR: `||`
  - Comma: `,`
- Between the evaluations of the first operand of the conditional `?:` operator and whichever of the second and third operands is evaluated
- The end of a full declarator
- Between the evaluation of a full expression and the next full expression to be evaluated; the following are full expressions:
  - An initializer that is not part of a compound literal
  - The expression in an expression statement
  - The controlling expression of a selection statement (`if` or `switch`)
  - The controlling expression of a `while` or `do` statement
  - Each of the (optional) expressions of a `for` statement
  - The (optional) expression in a `return` statement
- Immediately before a library function returns
- After the actions associated with each formatted input/output function conversion specifier
- Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call

This rule means that statements such as

```
i = i + 1;
a[i] = i;
```

have defined behavior, and statements such as the following do not:

```
/* i is modified twice between sequence points */
i = ++i + 1;

/* i is read other than to determine the value to be stored */
a[i++] = i;
```

Note that not all instances of a comma in C code denote a usage of the comma operator. For example, the comma between arguments in a function call is not a sequence point. However, according to the C Standard, 6.5.2.2, paragraph 10 [ISO/IEC 9899:2011]:

> Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.

This rule means that the order of evaluation for function call arguments is unspecified and can happen in any order.

Noncompliant Code Example

Programs cannot safely rely on the order of evaluation of operands between sequence points. In this noncompliant code example, `i` is evaluated twice without an intervening sequence point, and so the behavior of the expression is undefined:

```
#include <stdio.h>

void func(int i, int *b) {
  int a = i + b[++i];
  printf("%d, %d", a, i);
}
```

Compliant Solution

These examples are independent of the order of evaluation of the operands and can be interpreted in only one way:

```
#include <stdio.h>

void func(int i, int *b) {
  int a;
  ++i;
  a = i + b[i];
  printf("%d, %d", a, i);
}
```

Alternatively:

```
#include <stdio.h>

void func(int i, int *b) {
```

```
  int a = i + b[i + 1];
  ++i;
  printf("%d, %d", a, i);
}
```

Noncompliant Code Example

The call to `func()` in this noncompliant code example has undefined behavior because there is no sequence point between the argument expressions:

```
extern void func(int i, int j);

void f(int i) {
  func(i++, i);
}
```

The first (left) argument expression reads the value of `i` (to determine the value to be stored) and then modifies `i`. The second (right) argument expression reads the value of `i` between the same pair of sequence points as the first argument, but not to determine the value to be stored in `i`. This additional attempt to read the value of `i` has undefined behavior.

Compliant Solution

This compliant solution is appropriate when the programmer intends for both arguments to `func()` to be equivalent:

```
extern void func(int i, int j);

void f(int i) {
  i++;
  func(i, i);
}
```

This compliant solution is appropriate when the programmer intends for the second argument to be 1 greater than the first:

```
extern void func(int i, int j);

void f(int i) {
  int j = i++;
  func(j, i);
}
```

Noncompliant Code Example

The order of evaluation for function arguments is unspecified. This noncompliant code example exhibits **unspecified behavior** but not undefined behavior:

```
extern void c(int i, int j);
int glob;

int a(void) {
  return glob + 10;
}

int b(void) {
  glob = 42;
  return glob;
}

void func(void) {
  c(a(), b());
}
```

It is unspecified what order `a()` and `b()` are called in; the only guarantee is that both `a()` and `b()` will be called before `c()` is called. If `a()` or `b()` rely on shared state when calculating their return value, as they do in this example, the resulting arguments passed to `c()` may differ between compilers or architectures.

Compliant Solution

In this compliant solution, the order of evaluation for `a()` and `b()` is fixed, and so no unspecified behavior occurs:

```
extern void c(int i, int j);
int glob;

int a(void) {
  return glob + 10;
}

int b(void) {
  glob = 42;
  return glob;
}

void func(void) {
  int a_val, b_val;
```

```
  a_val = a();
  b_val = b();

  c(a_val, b_val);
}
```

Risk Assessment

Attempting to modify an object multiple times between sequence points may cause that object to take on an unexpected value, which can lead to unexpected program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP30-C	Medium	Probable	Medium	P8	L2

Related Guidelines

ISO/IEC TR 24772:2013	Operator Precedence/Order of Evaluation [JCW] Side-effects and Order of Evaluation [SAM]
MISRA C:2012	Rule 12.1 (advisory)

Bibliography

[ISO/IEC 9899:2011]	6.5, "Expressions" 6.5.2.2, "Function Calls" Annex C, "Sequence Points"
[Saks 2007]	
[Summit 2005]	Questions 3.1, 3.2, 3.3, 3.3b, 3.7, 3.8, 3.9, 3.10a, 3.10b, and 3.11

■ EXP35-C. Do not modify objects with temporary lifetime

The C11 Standard [ISO/IEC 9899:2011] introduced a new term: *temporary lifetime*. Modifying an object with temporary lifetime is undefined behavior. According to subclause 6.2.4, paragraph 8:

> A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all

> contained structures and unions) refers to an object with automatic storage duration and *temporary* lifetime. Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression or full declarator ends. Any attempt to modify an object with temporary lifetime results in undefined behavior.

This definition differs from the C99 Standard (which defines modifying the result of a function call or accessing it after the next sequence point as undefined behavior) because a temporary object's lifetime ends when the evaluation containing the full expression or full declarator ends, so the result of a function call can be accessed. This extension to the lifetime of a temporary also removes a quiet change to C90 and improves compatibility with C++.

C functions may not return arrays; however, functions can return a pointer to an array or a `struct` or `union` that contains arrays. Consequently, if a function call returns by value a `struct` or `union` containing an array, do not modify those arrays within the expression containing the function call. Do not access an array returned by a function after the next sequence point or after the evaluation of the containing full expression or full declarator ends.

Noncompliant Code Example (C99)

This noncompliant code example conforms to the C11 Standard; however, it fails to conform to C99. If compiled with a C99-conforming implementation, this code has undefined behavior because the sequence point preceding the call to `printf()` comes between the evaluation of its arguments and the access by `printf()` of the string in the returned object.

```
#include <stdio.h>

struct X { char a[8]; };

struct X salutation(void) {
  struct X result = { "Hello" };
  return result;
}

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

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

Compliant Solution

This compliant solution stores the structures returned by the call to `addressee()` before calling the `printf()` function. Consequently, this program conforms to C99 and C11.

```
#include <stdio.h>

struct X { char a[8]; };

struct X salutation(void) {
  struct X result = { "Hello" };
  return result;
}

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

int main(void) {
  struct X my_salutation = salutation();
  struct X my_addressee = addressee();

  printf("%s, %s!\n", my_salutation.a, my_addressee.a);
  return 0;
}
```

Noncompliant Code Example

This noncompliant code example attempts to retrieve an array and increment the array's first element. The array is part of a `struct` that is returned by a function call. Consequently, the array has temporary lifetime, and modifying the array is undefined behavior.

```
#include <stdio.h>

struct X { int a[6]; };

struct X addressee(void) {
  struct X result = { { 1, 2, 3, 4, 5, 6 } };
  return result;
}

int main(void) {
  printf("%x", ++(addressee().a[0]));
  return 0;
}
```

Compliant Solution

This compliant solution stores the structure returned by the call to `addressee()` as `my_x` before calling the `printf()` function. When the array is modified, its lifetime is no longer temporary but matches the lifetime of the block in `main()`.

```
#include <stdio.h>

struct X { int a[6]; };

struct X addressee(void) {
  struct X result = { { 1, 2, 3, 4, 5, 6 } };
  return result;
}

int main(void) {
  struct X my_x = addressee();
  printf("%x", ++(my_x.a[0]));
  return 0;
}
```

Risk Assessment

Attempting to modify an array or access it after its lifetime expires may result in erroneous program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP35-C	Low	Probable	Medium	**P4**	**L3**

Related Guidelines

ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM] Side-effects and Order of Evaluation [SAM]

Bibliography

[ISO/IEC 9899:2011]	6.2.4, “Storage Durations of Objects”

Chapter 7

Characters and Strings (STR)

STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013b]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character (see "STR03-C. Do not inadvertently truncate a string").

When strings live on the heap, this rule is a specific instance of "MEM35-C. Allocate sufficient memory for an object." Because strings are represented as arrays of characters, this rule is related to both "ARR30-C. Do not form or use out-of-bounds pointers or array subscripts" and "ARR38-C. Guarantee that library functions do not form invalid pointers."

Noncompliant Code Example (Off-by-One Error)

This noncompliant code example demonstrates an *off-by-one* error [Dowd 2006]. The loop copies data from `src` to `dest`. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of `dest`.

```
#include <stddef.h>

enum { ARRAY_SIZE = 32 };
```

```
void func(void) {
  char dest[ARRAY_SIZE];
  char src[ARRAY_SIZE];
  size_t i;

  for (i = 0; src[i] && (i < sizeof(dest)); ++i) {
    dest[i] = src[i];
  }
  dest[i] = '\0';
}
```

Compliant Solution (Off-by-One Error)

In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to `dest`:

```
#include <stddef.h>

enum { ARRAY_SIZE = 32 };

void func(void) {
  char dest[ARRAY_SIZE];
  char src[ARRAY_SIZE];
  size_t i;

  for (i = 0; src[i] && (i < sizeof(dest) - 1); ++i) {
    dest[i] = src[i];
  }
  dest[i] = '\0';
}
```

Noncompliant Code Example (`gets()`)

The `gets()` function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from `stdin`. This noncompliant code example assumes that `gets()` will not read more than `BUFFER_SIZE - 1` characters from `stdin`. This is an invalid assumption, and the resulting operation can result in a buffer overflow.

The `gets()` function reads characters from `stdin` into a destination array until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

```
#include <stdio.h>

#define BUFFER_SIZE 1024

void func(void) {
  char buf[BUFFER_SIZE];
  if (gets(buf) == NULL) {
    /* Handle error */
  }
}
```

See also "MSC24-C. Do not use deprecated or obsolescent functions."

Compliant Solution (fgets())

The `fgets()` function reads, at most, one less than the specified number of characters from a stream into an array. This solution is compliant because the number of characters copied from `stdin` to `buf` cannot exceed the allocated memory:

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

enum { BUFFERSIZE = 32 };

void func(void) {
  char buf[BUFFERSIZE];
  int ch;

  if (fgets(buf, sizeof(buf), stdin)) {
    /* fgets() succeeded; scan for new-line character */
    char *p = strchr(buf, '\n');
    if (p) {
      *p = '\0';
    } else {
      /* New-line not found; flush stdin to end of line */
      while ((ch = getchar()) != '\n' && ch != EOF)
        ;
      if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
         /* Character resembles EOF; handle error */
      }
    }
  } else {
    /* fgets() failed; handle error */
  }
}
```

The fgets() function is not a strict replacement for the gets() function because fgets() retains the new-line character (if read) and may also return a partial line. It is possible to use fgets() to safely process input lines too long to store in the destination array, but this is not recommended for performance reasons. Consider using one of the following compliant solutions when replacing gets().

Compliant Solution (gets_s())

The gets_s() function reads, at most, one less than the number of characters specified from the stream pointed to by stdin into an array.

The C Standard, Annex K [ISO/IEC 9899:2011], states:

> No additional characters are read after a new-line character (which is discarded) or after end-of-file. The discarded new-line character does not count towards number of characters read. A null character is written immediately after the last character read into the array.

If end-of-file is encountered and no characters have been read into the destination array, or if a read error occurs during the operation, then the first character in the destination array is set to the null character and the other elements of the array take unspecified values:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>

enum { BUFFERSIZE = 32 };

void func(void) {
  char buf[BUFFERSIZE];

  if (gets_s(buf, sizeof(buf)) == NULL) {
    /* Handle error */
  }
}
```

Compliant Solution (getline(), POSIX)

The getline() function is similar to the fgets() function but can dynamically allocate memory for the input buffer. If passed a null pointer, getline() dynamically allocates a buffer of sufficient size to hold the input. If passed a pointer to dynamically allocated storage that is too small to hold the contents of the string, the getline() function resizes the buffer, using realloc(), rather than truncating the input. If successful, the getline() function returns

the number of characters read, which can be used to determine if the input has any null characters before the new-line. The `getline()` function works only with dynamically allocated buffers. Allocated memory must be explicitly deallocated by the caller to avoid memory leaks (see "MEM31-C. Free dynamically allocated memory when no longer needed").

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

void func(void) {
  int ch;
  size_t buffer_size = 32;
  char *buffer = malloc(buffer_size);

  if (!buffer) {
    /* Handle error */
    return;
  }

  if ((ssize_t size = getline(&buffer, &buffer_size, stdin))
        == -1) {
    /* Handle error */
  } else {
    char *p = strchr(buffer, '\n');
    if (p) {
      *p = '\0';
    } else {
      /* New-line not found; flush stdin to end of line */
      while ((ch = getchar()) != '\n' && ch != EOF)
        ;
      if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
        /* Character resembles EOF; handle error */
      }
    }
  }
  free (buffer);
}
```

Note that the `getline()` function uses an **in-band error indicator**, in violation of "ERR02-C. Avoid in-band error indicators."

Noncompliant Code Example (`getchar()`)

Reading one character at a time provides more flexibility in controlling behavior, though with additional performance overhead. This noncompliant

code example uses the getchar() function to read one character at a time from stdin instead of reading the entire line at once. The stdin stream is read until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array. Similar to the noncompliant code example that invokes gets(), there are no guarantees that this code will not result in a buffer overflow.

```
#include <stdio.h>

enum { BUFFERSIZE = 32 };

void func(void) {
  char buf[BUFFERSIZE];
  char *p;
  int ch;
  p = buf;
  while ((ch = getchar()) != '\n' && ch != EOF) {
    *p++ = (char)ch;
  }
  *p++ = 0;
  if (ch == EOF) {
      /* Handle EOF or error */
  }
}
```

After the loop ends, if ch == EOF, the loop has read through to the end of the stream without encountering a new-line character, or a read error occurred before the loop encountered a new-line character. To conform to "FIO34-C. Distinguish between characters read from a file and EOF or WEOF," the error-handling code must verify that an end-of-file or error has occurred by calling feof() and ferror().

Compliant Solution (getchar())

In this compliant solution, characters are no longer copied to buf once index == BUFFERSIZE - 1, leaving room to null-terminate the string. The loop continues to read characters until the end of the line, the end of the file, or an error is encountered. When chars_read > index, the input string has been truncated.

```
#include <stdio.h>

enum { BUFFERSIZE = 32 };
```

```
void func(void) {
  char buf[BUFFERSIZE];
  int ch;
  size_t index = 0;
  size_t chars_read = 0;

  while ((ch = getchar()) != '\n' && ch != EOF) {
    if (index < sizeof(buf) - 1) {
      buf[index++] = (char)ch;
    }
    chars_read++;
  }
  buf[index] = '\0';  /* Terminate string */
  if (ch == EOF) {
    /* Handle EOF or error */
  }
  if (chars_read > index) {
    /* Handle truncation */
  }
}
```

Noncompliant Code Example (fscanf())

In this noncompliant example, the call to `fscanf()` can result in a write outside the character array `buf`:

```
#include <stdio.h>

enum { BUF_LENGTH = 1024 };

void get_data(void) {
  char buf[BUF_LENGTH];
  if (1 != fscanf(stdin, "%s", buf)) {
    /* Handle error */
  }

  /* Rest of function */
}
```

Compliant Solution (fscanf())

In this compliant solution, the call to `fscanf()` is constrained not to overflow `buf`:

```
#include <stdio.h>

enum { BUF_LENGTH = 1024 };
```

```
void get_data(void) {
  char buf[BUF_LENGTH];
  if (1 != fscanf(stdin, "%1023s", buf)) {
    /* Handle error */
  }

  /* Rest of function */
}
```

Noncompliant Code Example (argv)

In a **hosted environment**, arguments read from the command line are stored in process memory. The function `main()`, called at program startup, is typically declared as follows when the program accepts command-line arguments:

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

Command-line arguments are passed to `main()` as pointers to strings in the array members `argv[0]` through `argv[argc - 1]`. If the value of `argc` is greater than 0, the string pointed to by `argv[0]` is, by convention, the program name. If the value of `argc` is greater than 1, the strings referenced by `argv[1]` through `argv[argc - 1]` are the program arguments.

Vulnerabilities can occur when inadequate space is allocated to copy a command-line argument or other program input. In this noncompliant code example, an attacker can manipulate the contents of `argv[0]` to cause a buffer overflow:

```
#include <string.h>

int main(int argc, char *argv[]) {
  /* Ensure argv[0] is not null */
  const char *const name = (argc && argv[0]) ? argv[0] : "";
  char prog_name[128];
  strcpy(prog_name, name);

  return 0;
}
```

Compliant Solution (argv)

The `strlen()` function can be used to determine the length of the strings referenced by `argv[0]` through `argv[argc - 1]` so that adequate memory can be dynamically allocated.

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

```
int main(int argc, char *argv[]) {
  /* Ensure argv[0] is not null */
  const char *const name = (argc && argv[0]) ? argv[0] : "";
  char *prog_name = (char *)malloc(strlen(name) + 1);
  if (prog_name != NULL) {
    strcpy(prog_name, name);
  } else {
    /* Handle error */
  }
  free(prog_name);
  return 0;
}
```

Remember to add a byte to the destination string size to accommodate the null-termination character.

Compliant Solution (argv)

The `strcpy_s()` function provides additional safeguards, including accepting the size of the destination buffer as an additional argument (see "STR07-C. Use the bounds-checking interfaces for remediation of existing string manipulation code").

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
  /* Ensure argv[0] is not null */
  const char *const name = (argc && argv[0]) ? argv[0] : "";
  char *prog_name;
  size_t prog_size;

  prog_size = strlen(name) + 1;
  prog_name = (char *)malloc(prog_size);

  if (prog_name != NULL) {
    if (strcpy_s(prog_name, prog_size, name)) {
      /* Handle  error */
    }
  } else {
    /* Handle error */
  }
  /* ... */
  free(prog_name);
  return 0;
}
```

The strcpy_s() function can be used to copy data to or from dynamically allocated memory or a statically allocated array. If insufficient space is available, strcpy_s() returns an error.

Compliant Solution (argv)

If an argument will not be modified or concatenated, there is no reason to make a copy of the string. Not copying a string is the best way to prevent a buffer overflow and is also the most efficient solution. Care must be taken to avoid assuming that argv[0] is non-null.

```
int main(int argc, char *argv[]) {
  /* Be prepared for argv[0] to be null */
  const char * const prog_name = (argc && argv[0]) ? argv[0] : "";
  /* ... */
  return 0;
}
```

Noncompliant Code Example (getenv())

According to the C Standard, 7.22.4.6, paragraph 2 [ISO/IEC 9899:2011]:

> The getenv function searches an environment list, provided by the host environment, for a string that matches the string pointed to by name. The set of environment names and the method for altering the environment list are implementation-defined.

Environment variables can be arbitrarily large, and copying them into fixed-length arrays without first determining the size and allocating adequate storage can result in a buffer overflow.

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

void func(void) {
  char buff[256];
  char *editor = getenv("EDITOR");
  if (editor == NULL) {
    /* EDITOR environment variable not set */
  } else {
    strcpy(buff, editor);
  }
}
```

Compliant Solution (getenv())

Environmental variables are loaded into process memory when the program is loaded. As a result, the length of these strings can be determined by calling the strlen() function, and the resulting length can be used to allocate adequate dynamic memory:

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

void func(void) {
  char *buff;
  char *editor = getenv("EDITOR");
  if (editor == NULL) {
    /* EDITOR environment variable not set */
  } else {
    size_t len = strlen(editor) + 1;
    buff = (char *)malloc(len);
    if (buff == NULL) {
      /* Handle error */
    }
    memcpy(buff, editor, len);
    free(buff);
  }
}
```

Noncompliant Code Example (sprintf())

In this noncompliant code example, name refers to an external string; it could have originated from user input, from the file system, or from the network. The program constructs a file name from the string in preparation for opening the file.

```
#include <stdio.h>

void func(const char *name) {
  char filename[128];
  sprintf(filename, "%s.txt", name);
}
```

Because the sprintf() function makes no guarantees regarding the length of the generated string, a sufficiently long string in name could generate a buffer overflow.

Compliant Solution (sprintf())

The buffer overflow in the preceding noncompliant example can be prevented by adding a precision to the `%s` conversion specification. If the precision is specified, no more than that many bytes are written. The precision `123` in this compliant solution ensures that `filename` can contain the first 123 characters of `name`, the `.txt` extension, and the null terminator.

```
#include <stdio.h>

void func(const char *name) {
  char filename[128];
  sprintf(filename, "%.123s.txt", name);
}
```

Compliant Solution (snprintf())

A more general solution is to use the `snprintf()` function:

```
#include <stdio.h>

void func(const char *name) {
  char filename[128];
  snprintf(filename, sizeof(filename), "%s.txt", name);
}
```

Risk Assessment

Copying string data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can exploit this condition to execute arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR31-C	High	Likely	Medium	P18	L1

Related Vulnerabilities

CVE-2009-1252 results from a violation of this rule. The Network Time Protocol daemon (NTPd), before versions 4.2.4p7 and 4.2.5p74, contained calls to `sprintf` that allow an attacker to execute arbitrary code by overflowing a character array [xorl 2009].

CVE-2009-0587 results from a violation of this rule. Before version 2.24.5, Evolution Data Server performed unchecked arithmetic operations on the length of a user-input string and used the value to allocate space for a new

buffer. An attacker could thereby execute arbitrary code by inputting a long string, resulting in incorrect allocation and buffer overflow [xorl 2009].

Related Guidelines

ISO/IEC TR 24772:2013	String Termination [CJM] Buffer Boundary Violation (Buffer Overflow) [HCB] Unchecked Array Copying [XYW]
ISO/IEC TS 17961:2013	Using a tainted value to write to an object using a formatted input or output function [taintformatio] Tainted strings are passed to a string copying function [taintstrcpy]
MITRE CWE	CWE-119, Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-120, Buffer Copy without Checking Size of Input ("Classic Buffer Overflow") CWE-193, Off-by-one Error

Bibliography

[Dowd 2006]	Chapter 7, "Program Building Blocks" ("Loop Constructs," pp. 327–336)
[Drepper 2006]	Section 2.1.1, "Respecting Memory Bounds"
[ISO/IEC 9899:2011]	K.3.5.4.1, "The `gets_s` Function"
[Lai 2006]	
[NIST 2006]	SAMATE Reference Dataset Test Case ID 000-000-088
[Seacord 2013b]	Chapter 2, "Strings"
[xorl 2009]	FreeBSD-SA-09:11: NTPd Remote Stack Based Buffer Overflows

STR34-C. Cast characters to `unsigned char` before converting to larger integer sizes

Signed character data must be converted to `unsigned char` before being assigned or converted to a larger signed type. This rule applies to both `signed char` and (plain) `char` characters on implementations where `char` is defined to have the same range, representation, and behavior as `signed char`. However, this rule is applicable only in cases where the character data may contain

values that can be interpreted as negative numbers. For example, if the char type is represented by a two's complement 8-bit value, any character value greater than +127 is interpreted as a negative value.

This rule is a generalization of "STR37-C. Arguments to character handling functions must be representable as an unsigned char."

Noncompliant Code Example

This noncompliant code example is taken from a vulnerability in bash versions 1.14.6 and earlier that led to the release of CERT Advisory CA-1996-22. This vulnerability resulted from the sign extension of character data referenced by the c_str pointer in the yy_string_get() function in the parse.y module of the bash source code.

```
static int yy_string_get(void) {
  register char *c_str;
  register int c;

  c_str = bash_input.location.string;
  c = EOF;

  /* If the string doesn't exist or is empty, EOF found */
  if (c_str && *c_str) {
    c = *c_str++;
    bash_input.location.string = c_str;
  }
  return (c);
}
```

The c_str variable is used to traverse the character string containing the command line to be parsed. As characters are retrieved from this pointer, they are stored in a variable of type int. For implementations in which the char type is defined to have the same range, representation, and behavior as signed char, this value is sign-extended when assigned to the int variable. For character code 255 decimal (–1 in two's complement form), this sign extension results in the value –1 being assigned to the integer, which is indistinguishable from EOF.

Noncompliant Code Example

This problem can be repaired by explicitly declaring the c_str variable as unsigned char:

```
static int yy_string_get(void) {
  register unsigned char *c_str;
  register int c;
```

```
  c_str = bash_input.location.string;
  c = EOF;

  /* If the string doesn't exist or is empty, EOF found */
  if (c_str && *c_str) {
    c = *c_str++;
    bash_input.location.string = c_str;
  }
  return (c);
}
```

This example, however, violates "STR04-C. Use plain char for characters in the basic character set."

Compliant Solution

In this compliant solution, the result of the expression *c_str++ is cast to unsigned char before assignment to the int variable c:

```
static int yy_string_get(void) {
  register char *c_str;
  register int c;

  c_str = bash_input.location.string;
  c = EOF;

  /* If the string doesn't exist or is empty, EOF found */
  if (c_str && *c_str) {
    /* Cast to unsigned type */
    c = (unsigned char)*c_str++;

    bash_input.location.string = c_str;
  }
  return (c);
}
```

Noncompliant Code Example

In this noncompliant code example, the cast of *s to unsigned int can result in a value in excess of UCHAR_MAX because of integer promotions, a violation of "ARR30-C. Do not form or use out-of-bounds pointers or array subscripts":

```
#include <limits.h>
#include <stddef.h>

static const char table[UCHAR_MAX] = { 'a' /* ... */ };
```

```
ptrdiff_t first_not_in_table(const char *c_str) {
  for (const char *s = c_str; *s; ++s) {
    if (table[(unsigned int)*s] != *s) {
      return s - c_str;
    }
  }
  return -1;
}
```

Compliant Solution

This compliant solution casts the value of type `char` to `unsigned char` before the implicit promotion to a larger type:

```
#include <limits.h>
#include <stddef.h>

static const char table[UCHAR_MAX] = { 'a' /* ... */ };

ptrdiff_t first_not_in_table(const char *c_str) {
  for (const char *s = c_str; *s; ++s) {
    if (table[(unsigned char)*s] != *s) {
      return s - c_str;
    }
  }
  return -1;
}
```

Risk Assessment

Conversion of character data resulting in a value in excess of `UCHAR_MAX` is an often-missed error that can result in a disturbingly broad range of potentially severe vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR34-C	Medium	Probable	Medium	**P8**	**L2**

Related Vulnerabilities

CVE-2009-0887 results from a violation of this rule. In Linux PAM (up to version 1.0.3), the `libpam` implementation of `strtok()` casts a (potentially signed) character to an integer for use as an index to an array. An attacker can exploit this vulnerability by inputting a string with non-ASCII characters, causing the cast to result in a negative index and accessing memory outside of the array [xorl 2009].

Related Guidelines

ISO/IEC TS 17961:2013	Conversion of signed characters to wider integer types before a check for EOF [signconv]
MISRA-C	Rule 10.1 through Rule 10.4 (required)
MITRE CWE	CWE-704, Incorrect Type Conversion or Cast

Bibliography

[xorl 2009]	CVE-2009-0887: Linux-PAM Signedness Issue

Chapter 13

Concurrency (CON)

CON34-C. Declare objects shared between threads with appropriate storage durations

Accessing the automatic or thread-local variables of one thread from another thread is implementation-defined [ISO/IEC 9899:2011] and can cause invalid memory accesses because the execution of threads can be interwoven within the constraints of the synchronization model. As a result, the referenced stack frame or thread-local variable may no longer be valid when another thread tries to access it. Shared static variables can be protected by thread synchronization mechanisms. However, automatic (local) variables cannot be shared in the same manner because the referenced stack frame's thread would need to stop executing, or some other mechanism must be employed to ensure that the referenced stack frame is still valid. Do not access automatic or thread-local objects from a thread other than the one with which the object is associated. See "DCL30-C. Declare objects with appropriate storage durations" for information on how to declare objects with appropriate storage durations when data is not being shared between threads.

Noncompliant Code Example (Automatic Storage Duration)

This noncompliant code example passes the address of a variable to a child thread, which prints it out. The variable has automatic storage duration. Depending on the execution order, the child thread might reference the

variable after the variable's lifetime in the parent thread. This would cause the child thread to access an invalid memory location.

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

int child_thread(void *val) {
  int *res = (int *)val;
  printf("Result: %d\n", *res);
  return 0;
}

void create_thread(thrd_t *tid) {
  int val = 1;
  if (thrd_success != thrd_create(tid, child_thread, &val)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  create_thread(&tid);

  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

Noncompliant Code Example (Automatic Storage Duration)

One solution is to ensure that all objects with automatic storage duration shared between threads are declared such that their lifetime extends past the lifetime of the threads. This can be accomplished using a thread synchronization mechanism, such as `thrd_join()`. For example, in this compliant solution, `val` is declared in `main()`, where `thrd_join()` is called. Because the parent thread waits until the child thread completes before continuing its execution, the shared objects have a lifetime at least as great as the thread. However, this example relies on implementation-defined behavior and is nonportable.

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

int child_thread(void *val) {
  int *result = (int *)val;
  printf("Result: %d\n", *result);  /* Correctly prints 1 */
  return 0;
}
```

```
void create_thread(thrd_t *tid, int *val) {
  if (thrd_success != thrd_create(tid, child_thread, val)) {
    /* Handle error */
  }
}

int main(void) {
  int val = 1;
  thrd_t tid;
  create_thread(&tid, &val);
  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

Compliant Solution (Static Storage Duration)

This compliant solution stores the value in an object having static storage duration. The lifetime of this object is the entire execution of the program; consequently, it can be safely accessed by any thread.

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

int child_thread(void *v) {
  int *result = (int *)v;
  printf("Result: %d\n", *result);  /* Correctly prints 1 */
  return 0;
}

void create_thread(thrd_t *tid) {
  static int val = 1;
  if (thrd_success != thrd_create(tid, child_thread, &val)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  create_thread(&tid);
  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

Compliant Solution (Allocated Storage Duration)

This compliant solution stores the value passed to the child thread in a dynamically allocated object. Because this object will persist until explicitly freed, the child thread can safely access its value.

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

int child_thread(void *val) {
  int *result = (int *)val;
  printf("Result: %d\n", *result); /* Correctly prints 1 */
  return 0;
}

void create_thread(thrd_t *tid, int *value) {
  *value = 1;
  if (thrd_success != thrd_create(tid, child_thread,
                                  value)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int *value = (int *)malloc(sizeof(int));
  if (!value) {
    /* Handle error */
  }
  create_thread(&tid, value);
  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  free(value);
  return 0;
}
```

Noncompliant Code Example (Thread-Specific Storage)

In this noncompliant code example, the value is stored in thread-specific storage of the parent thread. However, because thread-specific data is available only to the thread that stores it, the `child_thread()` function will set `result` to a null value.

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

```
static tss_t key;

int child_thread(void *v) {
  int *result = tss_get(*(tss_t *)v);
  printf("Result: %d\n", *result);
  return 0;
}

int create_thread(void *thrd) {
  int *val = (int *)malloc(sizeof(int));
  if (val == NULL) {
    /* Handle error */
  }
  *val = 1;
  if (thrd_success != tss_set(key, val) {
    /* Handle error */
  }
  if (thrd_success != thrd_create((thrd_t *)thrd,
                                  child_thread, &key)) {
    /* Handle error */
  }
  return 0;
}

int main(void) {
  thrd_t parent_tid, child_tid;

  if (thrd_success != tss_create(&key, free)) {
    /* Handle error */
  }
  if (thrd_success != thrd_create(&parent_tid, create_thread,
                                  &child_tid)) {
    /* Handle error */
  }
  if (thrd_success != thrd_join(parent_tid, NULL)) {
    /* Handle error */
  }
  if (thrd_success != thrd_join(child_tid, NULL)) {
    /* Handle error */
  }
  if (thrd_success != tss_delete(key)) {
    /* Handle error */
  }
  return 0;
}
```

Compliant Solution (Thread-Specific Storage)

This compliant solution illustrates how thread-specific storage can be combined with a call to a thread synchronization mechanism, such as `thrd_join()`. Because the parent thread waits until the child thread completes before continuing its execution, the child thread is guaranteed to access a valid live object.

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

static tss_t key;

int child_thread(void *v) {
  int *result = v;
  printf("Result: %d\n", *result);  /* Correctly prints 1 */
  return 0;
}

int create_thread(void *thrd) {
  int *val = (int *)malloc(sizeof(int));
  if (val == NULL) {
    /* Handle error */
  }
  val = 1;
  if (thrd_success != tss_set(key, val)) {
    /* Handle error */
  }
  /* ... */
  void *v = tss_get(key);
  if (thrd_success != thrd_create((thrd_t *)thrd,
                                  child_thread, v)) {
    /* Handle error */
  }
  return 0;
}

int main(void) {
  thrd_t parent_tid, child_tid;

  if (thrd_success != tss_create(&key, free)) {
    /* Handle error */
  }
  if (thrd_success != thrd_create(&parent_tid, create_thread,
                                  &child_tid)) {
    /* Handle error */
  }
```

```
  if (thrd_success != thrd_join(parent_tid, NULL)) {
    /* Handle error */
  }
  if (thrd_success != thrd_join(child_tid, NULL)) {
    /* Handle error */
  }
  if (thrd_success != tss_delete(key)) {
    /* Handle error */
  }
  return 0;
}
```

This compliant solution uses pointer-to-integer and integer-to-pointer conversions, which have implementation-defined behavior (see "INT36-C. Converting a pointer to integer or integer to pointer").

Compliant Solution (Thread-Local Storage, Windows, Visual Studio)

Similar to the preceding compliant solution, this compliant solution uses thread-local storage combined with thread synchronization to ensure the child thread is accessing a valid live object. It uses the Visual Studio–specific `__declspec(thread)` language extension to provide the thread-local storage and the `WaitForSingleObject()` API to provide the synchronization.

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

DWORD WINAPI child_thread(LPVOID v) {
  int *result = (int *)v;
  printf("Result: %d\n", *result);  /* Correctly prints 1 */
  return NULL;
}

int create_thread(HANDLE *tid) {
  /* Declare val as a thread-local value */
  __declspec(thread) int val = 1;
  *tid = create_thread(NULL, 0, child_thread, &val, 0, NULL);
  return *tid == NULL;
}

int main(void) {
  HANDLE tid;

  if (create_thread(&tid)) {
    /* Handle error */
  }
```

```
  if (WAIT_OBJECT_0 != WaitForSingleObject(tid, INFINITE)) {
    /* Handle error */
  }
  CloseHandle(tid);

  return 0;
}
```

Noncompliant Code Example (OpenMP, parallel)

It is important to note that local data can be used securely with threads when using other thread interfaces, so the programmer need not always copy data into nonlocal memory when sharing data with threads. For example, the `shared` keyword in "The OpenMP® API Specification for Parallel Programming" [OpenMP] can be used in combination with OpenMP's threading interface to share local memory without having to worry about whether local automatic variables remain valid.

In this noncompliant code example, a variable `j` is declared outside a `parallel #pragma` and not listed as a private variable. In OpenMP, variables outside a `parallel #pragma` are shared unless designated as `private`.

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

int main(void) {
  int j = 0;
  #pragma omp parallel
  {
    int t = omp_get_thread_num();
    printf("Running thread - %d\n", t);
    for (int i = 0; i < 5050; i++) {
      j++;  /* j not private; could be a race condition */
    }
    printf("Just ran thread - %d\n", t);
    printf("loop count %d\n", j);
  }
  return 0;
}
```

Compliant Solution (OpenMP, parallel, private)

In this compliant solution, the variable `j` is declared outside of the `parallel #pragma` but is explicitly labeled as `private`:

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

```
int main(void) {
  int j = 0;
  #pragma omp parallel private(j)
  {
    int t = omp_get_thread_num();
    printf("Running thread - %d\n", t);
    for (int i = 0; i < 5050; i++) {
      j++;
    }
    printf("Just ran thread - %d\n", t);
    printf("loop count %d\n", j);
  }
  return 0;
}
```

Risk Assessment

Threads that reference the stack of other threads can potentially overwrite important information on the stack, such as function pointers and return addresses. The compiler may not generate warnings if the programmer allows one thread to access another thread's local variables, so a programmer may not catch a potential error at compile time. The remediation cost for this error is high because analysis tools have difficulty diagnosing problems with concurrency and race conditions.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
CON34-C	Medium	Probable	High	**P4**	**L3**

Bibliography

[ISO/IEC 9899:2011]	6.2.4, "Storage Durations of Objects"
[OpenMP]	The OpenMP® API Specification for Parallel Programming

■ CON40-C. Do not refer to an atomic variable twice in an expression

A consistent locking policy guarantees that multiple threads cannot simultaneously access or modify shared data. Atomic variables eliminate the need for locks by guaranteeing thread safety when certain operations are performed on them. The thread-safe operations on atomic variables are specified in the C Standard, subclauses 7.17.7 and 7.17.8 [ISO/IEC 9899:2011]. While atomic

operations can be combined, combined operations do not provide the thread safety provided by individual atomic operations.

Every time an atomic variable appears on the left-hand side of an assignment operator, including a compound assignment operator such as *=, an atomic write is performed on the variable. The use of the increment (++) or decrement (--) operators on an atomic variable constitutes an atomic read-and-write operation and is consequently thread-safe. Any reference of an atomic variable anywhere else in an expression indicates a distinct atomic read on the variable.

If the same atomic variable appears twice in an expression, then two atomic reads, or an atomic read and an atomic write, are required. Such a pair of atomic operations is not thread-safe, as another thread can modify the atomic variable between the two operations. Consequently, an atomic variable must not be referenced twice in the same expression.

Noncompliant Code Example (atomic_bool)

This noncompliant code example declares a shared `atomic_bool flag` variable and provides a `toggle_flag()` method that negates the current value of `flag`:

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag;

void init_flag(void) {
  atomic_init(&flag, false);
}

void toggle_flag(void) {
  bool temp_flag = atomic_load(&flag);
  temp_flag = !temp_flag;
  atomic_store(&flag, temp_flag);
}

bool get_flag(void) {
  return atomic_load(&flag);
}
```

Execution of this code may result in a data race because the value of `flag` is read, negated, and written back. This occurs even though the read and write are both atomic.

Consider, for example, two threads that call `toggle_flag()`. The expected effect of toggling `flag` twice is that it is restored to its original value. However, the scenario in Table 13–3 leaves `flag` in the incorrect state.

Table 13–3. `Toggle_Flag()` without Compare-and-Exchange

Time	flag	Thread	Action
1	`true`	t_1	Reads the current value of `flag`, `true`, into a cache
2	`true`	t_2	Reads the current value of `flag`, (still) `true`, into a different cache
3	`true`	t_1	Toggles the temporary variable in the cache to `false`
4	`true`	t_2	Toggles the temporary variable in the different cache to `false`
5	`false`	t_1	Writes the cache variable's value to `flag`
6	`false`	t_2	Writes the different cache variable's value to `flag`

As a result, the effect of the call by t_2 is not reflected in `flag`; the program behaves as if `toggle_flag()` were called only once, not twice.

Compliant Solution (`atomic_compare_exchange_weak()`)

This compliant solution uses a compare-and-exchange to guarantee that the correct value is stored in `flag`. All updates are visible to other threads. The call to `atomic_compare_exchange_weak()` is in a loop in conformance with "CON41-C. Wrap functions that can fail spuriously in a loop."

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag;

void init_flag(void) {
  atomic_init(&flag, false);
}

void toggle_flag(void) {
  bool old_flag = atomic_load(&flag);
  bool new_flag;
  do {
    new_flag = !old_flag;
  } while (!atomic_compare_exchange_weak(&flag, &old_flag, new_flag));
}

bool get_flag(void) {
  return atomic_load(&flag);
}
```

An alternative solution is to use the `atomic_flag` data type for managing Boolean values atomically. However, `atomic_flag` does not support a toggle operation.

Compliant Solution (Compound Assignment)

This compliant solution uses the `^=` assignment operation to toggle `flag`. This operation is guaranteed to be atomic, according to the C Standard, 6.5.16.2, paragraph 3. This operation performs a bitwise-exclusive-or between its arguments, but for Boolean arguments, this is equivalent to negation.

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag;

void toggle_flag(void) {
  flag ^= 1;
}

bool get_flag(void) {
  return flag;
}
```

Another alternative solution is to use a mutex to protect the atomic operation, but this solution loses the performance benefits of atomic variables.

Noncompliant Code Example

This noncompliant code example takes an atomic global variable `n` and computes `n + (n-1) + (n-2) + ... + 1`, using the formula `n * (n + 1) / 2`:

```
#include <stdatomic.h>

atomic_int n;

void compute_sum(void) {
  return n * (n + 1) / 2;
}
```

The value of `n` may change between the two atomic reads of `n` in the expression, yielding an incorrect result.

Compliant Solution

This compliant solution passes the atomic variable as a function parameter, forcing the variable to be copied, and guaranteeing a correct result:

```
#include <stdatomic.h>

void compute_sum(atomic_int n) {
  return n * (n + 1) / 2;
}
```

Risk Assessment

When operations on atomic variables are assumed to be atomic, but are not atomic, surprising data races can occur, leading to corrupted data and invalid control flow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON40-C	Medium	Probable	Medium	P8	L2

Related Guidelines

MITRE CWE	CWE-366, Race Condition within a Thread CWE-413, Improper Resource Locking CWE-567, Unsynchronized Access to Shared Data in a Multithreaded Context CWE-667, Improper locking

Bibliography

[ISO/IEC 14882:2011]	6.5.16.2, “Compound Assignment” 7.17, “Atomics”