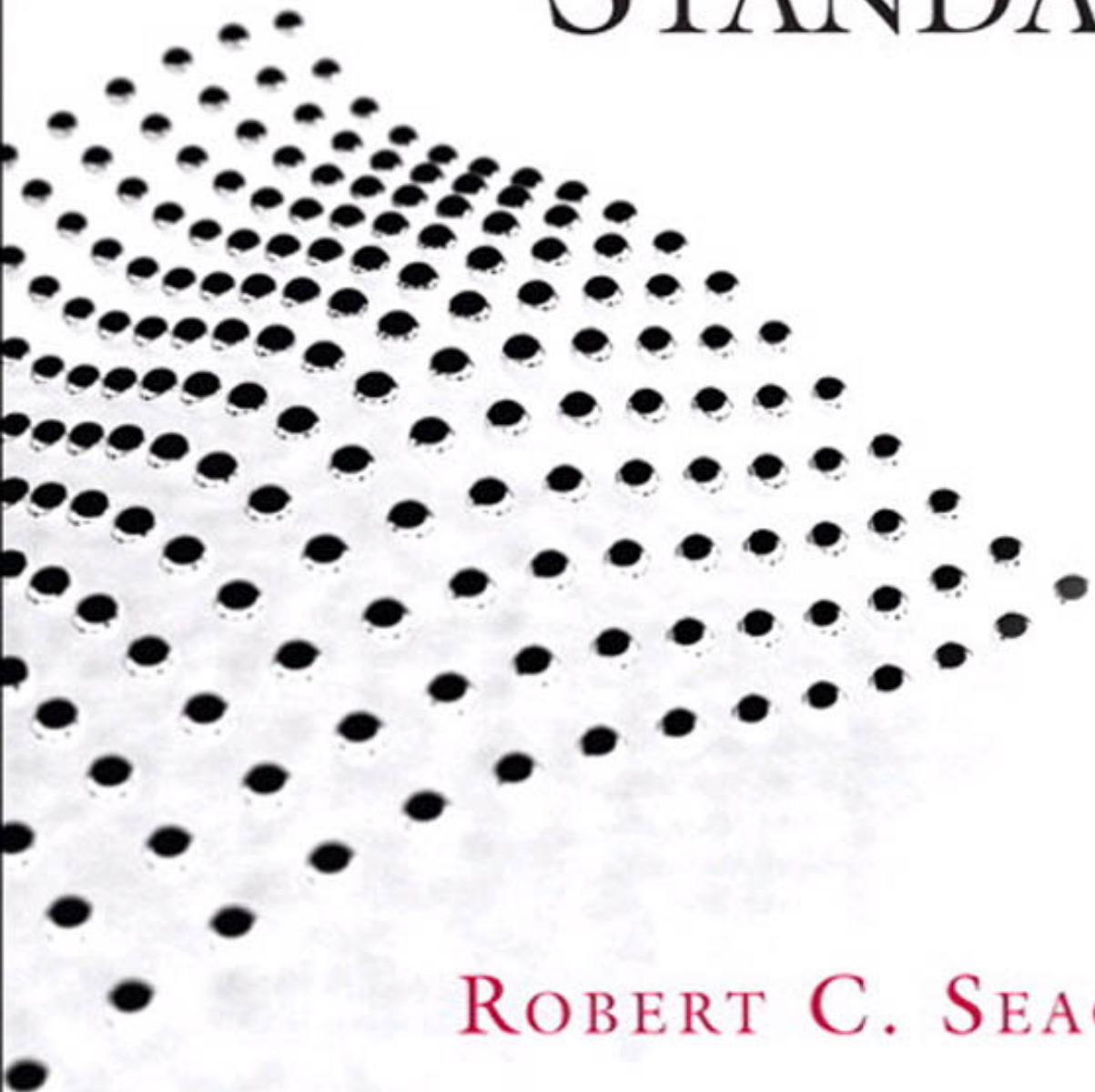
SEI SERIES • A CERT® BOOK

The CERT® C Secure Coding Standard



Robert C. Seacord

Carnegie Mellon
Software Engineering Institute

The SEI Series in Software Engineering

Library of Congress Cataloging-in-Publication Data

Seacord, Robert C.
The CERT C secure coding standard / Robert C. Seacord.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-56321-2 (pbk. : alk. paper) 1. C (Computer program language) 2. Computer security. I. Title.

QA76.73.C15S4155 2008
005.8—dc22 2008030261

ISBN 13: 978-0-321-56321-7
ISBN 10: 0-321-56321-2
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, October 2008.

Preface

An essential element of secure coding in the C programming language is a well-documented and enforceable coding standard. Coding standards encourage programmers to follow a uniform set of guidelines determined by the requirements of the project and organization rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes).

The CERT® C Secure Coding Standard provides guidelines for secure coding in the C programming language. The goal of these guidelines is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. Developing code in compliance with this coding standard will result in higher quality systems that are robust and more resistant to attack.

This standard is supported by training available from the Software Engineering Institute (SEI) and other licensed partners and is a basis for the Global Information Assurance Certification (GIAC) Secure Software Programmer–C (GSSP-C) exam and certification.

The Demand for Secure Software

The Morris worm incident, which brought 10 percent of Internet systems to a halt in November 1988, resulted in a new and acute awareness of the need for secure software systems. Twenty years later, many security analysts, software developers, software users, and policymakers are asking the question, Why isn't software more secure?

The first problem is that the term *software security*, as it is used today, is meaningless. I have attempted to define this term, as have others, but there is no generally accepted definition. Why does this matter?

A variety of reasons are given for why software is not more secure: for example, the tools are inadequate, programmers lack sufficient training, and schedules are too short. But these are all solvable problems. The root cause of the issue lies elsewhere.

The reason software systems are not more secure is that there is no *demand* for secure software. In simple terms, if one vendor offers a product that has more features and better performance and is available today and another vendor offers a *secure* product that has fewer features and lesser performance and will be available in 6 months, there is really no question as to which product customers will buy, and vendors *know* this.

So why don't customers buy secure products? Again, this is because the word *secure* is meaningless in this context. Why would a customer pass up tangible benefits to buy a product that has an ill-defined and intangible property?

The problem is addressed by this coding standard. While developing code in compliance with this coding standard does not guarantee the security of a software system, it does tell you a great deal about the quality and security of the code. It tells you that the software was developed to a set of industry standard rules and recommendations that were developed by the leading experts in the field. It tells you that a tremendous amount of attention and effort went into producing code that is free from the common coding errors that have resulted in numerous vulnerabilities that have been reported to and published by the CERT Coordination Center (CERT/CC) over the past two decades. It tells you that the software developers who produced the code have done so with a real knowledge of the types of vulnerabilities that can exist and the exploits that can be used against them, and consequently have developed the software with a real security mindset in place.

So, the *small* problem we have set out to address in this book is to change the market dynamic for developing and purchasing software systems. By producing an *actionable and measurable* definition of software security for C language programs—compliance with the rules and recommendations in this standard—we have defined a mechanism by which customers can demand secure software systems and vendors can comply. Furthermore, the concept of a secure system now has *value* because the word *secure* has meaning.

Community Development Process

The CERT® C Secure Coding Standard was developed over a period of two and a half years as a community effort involving 226 contributors and reviewers.

The following development process was followed:

1. Rules and recommendations for a coding standard were solicited from the communities involved in the development and application of the C programming language, including the formal standard bodies responsible for the C language standard and user groups.

2. These rules and recommendations were edited by members of the CERT technical staff and industry experts for content and style on the CERT Secure Coding Standards wiki at www.securecoding.cert.org.
3. The user community reviewed and commented on the publicly posted content using threaded discussions and other communication tools. Drafts of this standard were reviewed at the London and Kona meetings by ISO/IEC WG14 and subjected to the scrutiny of the public, including members of the Association of C and C++ Users (ACCU) and the `comp.lang.c` newsgroup.

The Wiki versus This Book

Developing a secure coding standard on a wiki has many advantages. However, one disadvantage is that the content is constantly evolving. This is ideal 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 final document before they can commit to complying with a (fixed) set of rules and recommendations. This book serves that purpose as Version 1.0 of *The CERT® C Secure Coding Standard*.

Starting with the production of this book in June 2008, Version 1.0 and the wiki versions of the Secure Coding Standard began to diverge. Because both the C programming language and our knowledge of how to use it securely are still evolving, CERT will continue to evolve *The CERT® C Secure Coding Standard* on the Secure Coding wiki. These changes may then be incorporated into future, officially released versions of this standard.

Purpose

This book provides developers with *guidelines* for secure coding in the C programming language. These guidelines serve a variety of purposes. First, they enumerate common errors in C language programming that can lead to software defects, security flaws, and software vulnerabilities. These are all errors for which a conforming compiler is not required by the standard to issue a fatal diagnostic. In other words, the compiler will generate an executable, frequently without issuing any warnings, which can be shipped and deployed, and the resulting program may still contain flaws that make it vulnerable to attack.

Second, this coding standard provides recommendations for how to produce secure code. Failure to comply with these recommendations does not necessarily mean that the software is insecure, but if followed, these recommendations can be powerful tools in eliminating vulnerabilities from software.

Third, this coding standard identifies nonportable coding practices. Portability is not a strict requirement of security, but nonportable assumptions in code often result in vulnerabilities when code is ported to platforms for which these assumptions are no longer valid.

Rules

Guidelines are classified as either *rules* or *recommendations*. Guidelines are defined to be rules when all of the following conditions are met:

1. Violation of the coding practice is likely to result in a security flaw that may result in an exploitable vulnerability.
2. There is a denumerable set of conditions for which violating the coding practice is necessary to ensure correct behavior.
3. Conformance to the coding practice can be determined through automated analysis, formal methods, or manual inspection techniques.

Implementation of the secure coding rules defined in this standard are necessary (but not sufficient) to ensure the security of software systems developed in the C programming language. Figure P–1 shows how the 89 rules in this secure coding standard are categorized.

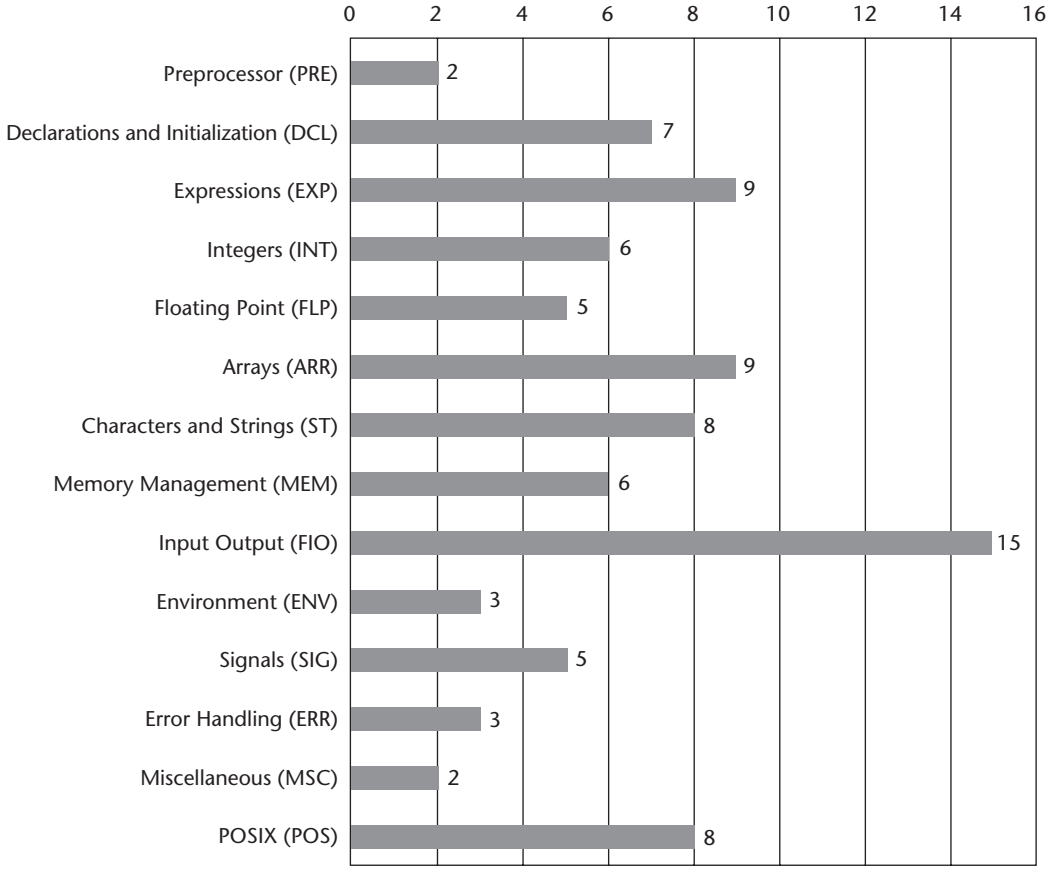


Figure P–1. CERT C Secure Coding rules

Recommendations

Guidelines are defined to be recommendations when all of the following conditions are met:

1. Application of the coding practice is likely to improve system security.
2. One or more of the requirements necessary for a coding practice to be considered a rule cannot be met.

The set of recommendations that a particular development effort adopts depends on the security requirements of the final software product. Projects with high-security requirements can dedicate more resources to security and consequently are likely to adopt a larger set of recommendations.

Figure P–2 shows how the 132 recommendations in this secure coding standard are categorized.

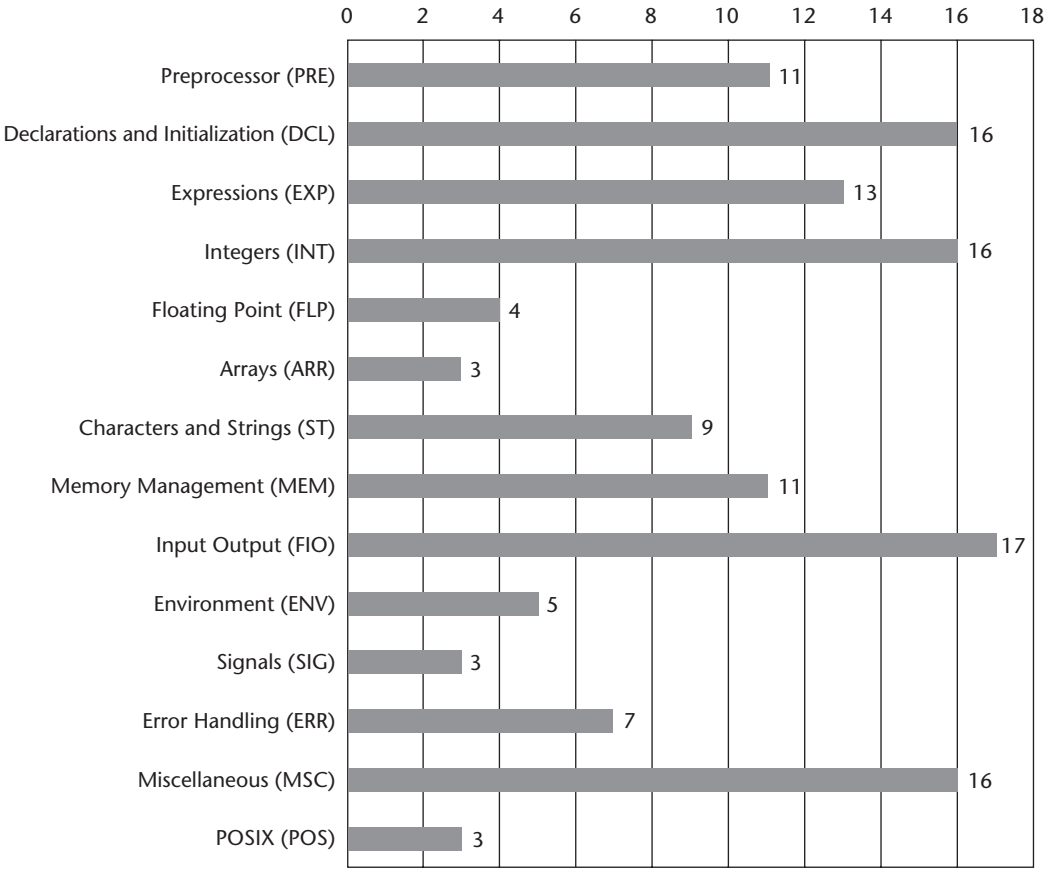


Figure P–2. CERT C Secure Coding recommendations

To ensure that the source code conforms to this secure coding standard, it is necessary to have measures in place that check for rules violations. The most effective means of achieving this is to use one or more static analysis tools. Where a rule cannot be checked by a tool, a manual review is required.

Both freely available and commercial source code analysis tools are available to automatically detect violations of CERT C Secure Coding Standard rules and recommendations, including Compass/ROSE, which has been developed by Lawrence Livermore National Laboratory and extended by CERT (www.rosecompiler.org).

Scope

The CERT® C Secure Coding Standard was developed specifically for versions of the C programming language defined in these publications:

- ISO/IEC 9899:1999, *Programming Languages—C*, Second Edition [ISO/IEC 9899:1999]
- Technical corrigenda TC1, TC2, and TC3
- ISO/IEC TR 24731-1, *Extensions to the C Library, Part I: Bounds-Checking Interfaces* [ISO/IEC TR 24731-1:2007]
- ISO/IEC PDTR 24731-2, *Extensions to the C Library, Part II: Dynamic Allocation Functions* [ISO/IEC PDTR 24731-2]

Most of the material included in this standard can also be applied to earlier versions of the C programming language.

Rules and recommendations included in this standard are designed to be operating system and platform independent. However, the best solutions to secure coding problems are often platform specific. In most cases, this standard provides appropriate compliant solutions for POSIX-compliant and Windows operating systems. In many cases, compliant solutions have also been provided for specific platforms such as Linux and OpenBSD. Occasionally, we also point out implementation-specific behaviors when these behaviors are of interest.

Rationale

A secure coding standard for the C programming language can create the highest value for the longest period of time by focusing on C99 and the relevant post-C99 technical reports.

In addition, because considerably more money and effort is devoted to developing new code than maintaining existing code, the highest return on investment comes from influencing programmers who are developing new code [Seacord 03]. Maintaining existing code is still an important concern, however.

The C standard (C99) documents existing practice where possible [ISO/IEC 9899:1999]. That is, most features must be tested in an implementation before being included in the standard. *The CERT® C Secure Coding Standard* has a different purpose. When existing practice serves this purpose, that is fine, but the goal is to create a new set of best practices, and that includes introducing some concepts that are not yet widely known. To put it a different way, the CERT C secure coding guidelines are attempting to drive change rather than just document it.

For example, the C library technical report, part 1 (TR 24731-1), is gaining support, but at present it 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 such functions simply because they are not yet widely implemented.

C99 is more widely implemented than TR 24731-1, but even if it were not yet, 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 and will be supported into the future.

Some vendors have extensions to C, and some have implemented only part of the C standard before stopping development. Consequently, it is not possible to back up and only discuss 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, a recent demarcation point was selected so that the rules and recommendations 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 C90. This is one of many tradeoffs 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 C99 and the post-C99 technical reports. A close-second priority is supporting remediation of old code using C99 and the technical reports.

This standard does try to 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 standard but only a few important ones.

Issues Not Addressed

There are a number of issues not addressed by this secure coding standard.

- **Coding Style.** Coding style issues are subjective, and it has proven impossible to develop a consensus on appropriate style guidelines. Consequently, this standard does not require any particular coding style to be enforced but only that the user define style guidelines and apply those guidelines consistently. The easiest way to consistently apply a coding style is with the use of a code formatting tool. Many interactive development environments (IDEs) provide such capabilities.
- **Tools.** As a federally funded research and development center (FFRDC), the SEI is not in a position to recommend particular vendors or tools to enforce these guidelines. The user of this document is free to choose tools, and vendors are encouraged to provide tools to enforce this standard.
- **Controversial Rules.** In general, the CERT secure coding standards try to avoid the inclusion of controversial rules that lack a broad consensus.

Who Should Read This Book

The CERT® C Secure Coding Standard is primarily intended for developers of C language programs. While security is important for Internet-facing systems, for example, it is also important for any software component that may be included or deployed as part of a secure software system. With systems increasingly being composed of software components, or even other systems, it is difficult to identify situations in which software is guaranteed not to be used in another context, which perhaps has more stringent security requirements.

This book is also useful for C language programmers who don't realize they are interested in security, as most of these guidelines have practical applications for achieving other quality attributes such as safety, reliability, dependability, robustness, availability, and maintainability.

While not intended for C++ programmers, this book may 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.

How This Book Is Organized

This book is organized into an introductory chapter, thirteen chapters containing guidelines in specific topic areas, and an appendix containing POSIX guidelines to demonstrate

how this secure coding standard can be customized for particular environments. The POSIX appendix is nonnormative and not a prescriptive part of the standard.

Most guidelines have a consistent structure. Each guideline in this standard has a unique *identifier,* which is included in the title. The title of the guidelines and the introductory paragraphs define the rule or recommendation. This is typically followed by one or more pairs of *noncompliant code examples* and *compliant solutions*. Each guideline also includes a *risk assessment* and a list of appropriate *references* (where applicable). Guidelines may also include a table of *related vulnerabilities*.

Guideline Identifiers

Guideline identifiers consist 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 can be used to group similar guidelines and to indicate to which category a guideline belongs.

The numeric value is used to give each guideline a unique identifier. Numeric values in the range of 00 to 29 are reserved for recommendations, while values in the range of 30 to 99 are reserved for rules.

Noncompliant Code Examples and Compliant Solutions

Noncompliant code examples are examples of insecure code that violate 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 your code is now compliant with the guideline.

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

Risk Assessment

Each guideline contains a risk assessment section, which attempts to quantify and qualify the risk of violating each guideline. This information is intended primarily for remediation projects to help prioritize repairs, as it is assumed that new code will be developed in conformance with the entire standard.

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: How serious are the consequences of the rule being ignored?
  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 could lead to an exploitable vulnerability?
  1 = unlikely
  2 = probable
  3 = likely
- Remediation cost: How expensive is it to comply with the rule?
  1 = high (manual detection and correction)
  2 = medium (automatic detection and manual correction)
  3 = low (automatic detection and correction)

The three values are multiplied together for each rule. This product provides a measure that can be used in prioritizing the application of the rules. These products range from 1 to 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. As a result, it is possible to claim level 1, level 2, or complete compliance (level 3) with a standard by implementing all rules in a level, as shown in Figure P–3.

Recommendations are not compulsory, and risk assessments are provided for information purposes only.

References

Guidelines include frequent references to the vulnerability notes in the CERT/CC Vulnerability Notes Database [CERT/CC VND], CWE IDs in MITRE's Common Weakness Enumeration (CWE) [MITRE 07], and CVE numbers from MITRE's Common Vulnerabilities and Exposures (CVE) [CVE].

You can create a unique URL to get more information on any of these topics 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

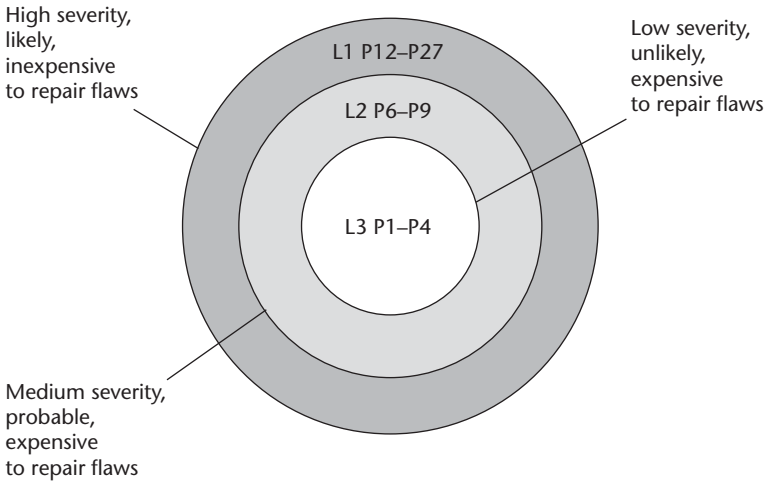


Figure P–3. Priorities and levels

- CWE ID 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
- 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

Guidelines are frequently correlated with language vulnerabilities in *Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use* [ISO/IEC PDTR 24772].

Related Vulnerabilities

Rules and recommendations linked to violations of actual vulnerabilities published in the CERT/CC Vulnerability Notes Database are shown in sections marked "Related Vulnerabilities" and are presented in table format, as in this example:

Metric	ID	Date Public	Name
1.62	VU#606700	03/19/2007	file integer overflow vulnerability
2.06	VU#559444	03/13/2007	Apple Mac OS X ImageIO integer overflow vulnerability

Metric	ID	Date Public	Name
22.22	VU#551436	02/23/2007	Mozilla Firefox SVG viewer vulnerable to integer overflow
0	VU#162289	04/17/2006	C compilers may silently discard some wraparound checks

New links are continually added. 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.

These tables consist of four fields: *metric*, *ID*, *date public*, and *name*.

Vulnerability Metric. The CERT vulnerability metric value is a number between 0 and 180 that assigns an approximate severity to the vulnerability. This value incorporates several elements:

- Is information about the vulnerability widely available or known?
- Is the vulnerability being exploited in incidents reported to CERT or other incident response teams?
- Is the Internet infrastructure (e.g., routers, name servers, critical Internet protocols) at risk because of this vulnerability?
- How many systems on the Internet are at risk from this vulnerability?
- What is the impact of exploiting the vulnerability?
- How easy is it to exploit the vulnerability?
- What are the preconditions required to exploit the vulnerability?

Because the questions are answered with approximate values based on the judgment of vulnerability analysts and may differ significantly from one site to another, you should not rely too heavily on the metric for prioritizing response to vulnerabilities. Rather, this metric may be useful for separating the serious vulnerabilities from the larger number of less severe vulnerabilities described in the database. Because the questions are not all weighted equally, the resulting score is not linear (that is, a vulnerability with a metric of 40 is not twice as severe as one with a metric of 20).

An alternative vulnerability severity metric is the Common Vulnerability Scoring System (CVSS) [Mell 07].

Vulnerability ID. Vulnerability ID numbers are assigned at random to uniquely identify a vulnerability. These IDs are four to six digits long and are usually prefixed with VU# to mark them as vulnerability IDs.

Date Public. This is the date on which the vulnerability was first publicly disclosed. Usually this date is when the vulnerability note was first published, when an exploit was first discovered, when the vendor first distributed a patch publicly, or when a description of the vulnerability was posted to a public mailing list. By default, this date is set to the vulnerability note publication date.

Vulnerability Name. The vulnerability name is a short description that summarizes the nature of the problem and the affected software product. While the name may include a clause describing the impact of the vulnerability, most names are focused on the nature of the defect that caused the problem to occur.

Type-generic macros may also be used, for example, to swap two variables of any type, provided they are of the same type.

PRE00-EX5: Macro parameters exhibit call-by-name semantics, whereas functions are call-by-value. Macros must be used in cases where call-by-name semantics are required.

Risk Assessment

Improper use of macros may result in undefined behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE00-C	medium	unlikely	medium	P4	L3

References

- [FSF 05] Section 5.34, "An Inline Function Is as Fast as a Macro"
- [Dewhurst 02] Gotcha #26, "#define Pseudofunctions"
- [ISO/IEC 9899:1999] Section 6.7.4, "Function Specifiers"
- [ISO/IEC PDTR 24772] "NMP Pre-processor Directives"
- [Kettlewell 03]
- [MISRA 04] Rule 19.7
- [Summit 05] Question 10.4

PRE01-C. Use parentheses within macros around parameter names

Parenthesize all parameter names found in macro definitions. See also PRE00-C, "Prefer inline or static functions to function-like macros," and PRE02-C, "Macro replacement lists should be parenthesized."

Noncompliant Code Example

This `CUBE()` macro definition is noncompliant because it fails to parenthesize the parameter names.

```
#define CUBE(I) (I * I * I)
```

As a result, the invocation

```
int a = 81 / CUBE(2 + 1);
```

expands to

```
int a = 81 / (2 + 1 * 2 + 1 * 2 + 1); /* evaluates to 11 */
```

which is clearly not the desired result.

Compliant Solution

Parenthesizing all parameter names in the `CUBE()` macro allows it to expand correctly (when invoked in this manner).

```
#define CUBE(I) ( (I) * (I) * (I) )
int a = 81 / CUBE(2 + 1);
```

Exceptions

PRE01-EX1: When the parameter names are surrounded by commas in the replacement text, regardless of how complicated the actual arguments are, there is no need for parenthesizing the macro parameters. Because commas have lower precedence than any other operator, there is no chance of the actual arguments being parsed in a surprising way. Comma separators, which separate arguments in a function call, also have lower precedence than other operators, although they are technically different from comma operators.

```
#define FOO(a, b, c) bar(a, b, c)
/* ... */
FOO(arg1, arg2, arg3);
```

PRE01-EX2: Macro parameters cannot be individually parenthesized when concatenating tokens using the `##` operator, converting macro parameters to strings using the `#` operator, or concatenating adjacent string literals. The `JOIN()` macro below concatenates both arguments to form a new token. The `SHOW()` macro converts the single argument into a string literal, which is then concatenated with the adjacent string literal to form the format specification in the call to `printf()`.

```
#define JOIN(a, b) (a ## b)
#define SHOW(a) printf(#a " = %d\n", a)
```

See PRE05-C, "Understand macro replacement when concatenating tokens or performing stringification," for more information on using the `##` operator to concatenate tokens.

Risk Assessment

Failing to parenthesize the parameter names in a macro can result in unintended program behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE01-C	medium	probable	low	P12	L1

References

- [ISO/IEC 9899:1999] Section 6.10, "Preprocessing Directives," and Section 5.1.1, "Translation Environment"
- [ISO/IEC PDTR 24772] "JCW Operator Precedence/Order of Evaluation"
- [MISRA 04] Rule 19.1
- [Plum 85]
- [Summit 05] Question 10.1

PRE02-C. Macro replacement lists should be parenthesized

Macro replacement lists should be parenthesized to protect any lower-precedence operators from the surrounding expression. See also PRE00-C, "Prefer inline or static functions to function-like macros," and PRE01-C, "Use parentheses within macros around parameter names."

Noncompliant Code Example

This `CUBE()` macro definition is noncompliant because it fails to parenthesize the replacement list.

```
#define CUBE(X) (X) * (X) * (X)
int i = 3;
int a = 81 / CUBE(i);
```

As a result, the invocation

```
int a = 81 / CUBE(i);
```

expands to

```
int a = 81 / i * i * i;
```

which evaluates as

```
int a = ((81 / i) * i) * i);  /* evaluates to 243 */
```

which is not the desired behavior.

existing destination file in FIO10-C, "Take care when using the `rename()` function," for an example of this exception.

EXP12-EX2: If a function cannot fail or if the return value cannot signify an error condition, the return value may be ignored. Such functions should be added to a white list when automatic checkers are used.

```
strcpy(dst, src);
```

Risk Assessment

Failure to handle error codes or other values returned by functions can lead to incorrect program flow and violations of data integrity.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP12-C	medium	unlikely	medium	P4	L3

References

- [ISO/IEC 9899:1999] Section 6.8.3, "Expression and Null Statements"
- [ISO/IEC PDTR 24772] "CSJ Passing Parameters and Return Values"

EXP30-C. Do not depend on order of evaluation between sequence points

Evaluation of an expression may produce side effects. At specific points during execution called sequence points, all side effects of previous evaluations have completed and no side effects of subsequent evaluations have yet taken place.

According to C99, Section 6.5:

> Between the previous and next sequence point an object can only have its stored value modified once by the evaluation of an expression. Additionally, the prior value can be read only to determine the value to be stored.

This requirement must be met for each allowable ordering of the subexpressions of a full expression; otherwise the behavior is undefined.

This rule means that statements such as

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

are allowed, while statements like

```
/* 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;
```

are not.

Noncompliant Code Example

Programs cannot safely rely on the order of evaluation of operands between sequence points. In this noncompliant code example, the order of evaluation of the operands to the + operator is unspecified.

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

If `i` was equal to 0 before the statement, the statement may result in the following outcome:

```
a = 0 + b[1];
```

Or it may result in the following outcome:

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

Compliant Solution

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

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

Or alternatively:

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

Noncompliant Code Example

The order of evaluation for function arguments is unspecified.

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

The call to `func()` has undefined behavior because there are no sequence points between the argument expressions. 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 solution is appropriate when the programmer intends for both arguments to `func()` to be equivalent.

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

This solution is appropriate when the programmer intends for the second argument to be one greater than the first.

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

Risk Assessment

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP30-C	medium	probable	medium	P8	L2

References

- [ISO/IEC 9899:1999] Section 5.1.2.3, "Program Execution," Section 6.5, "Expressions," and Annex C, "Sequence Points"
- [ISO/IEC PDTR 24772] "JCW Operator Precedence/Order of Evaluation" and "Side-Effects and Order of Evaluation [SAM]"
- [MISRA 04] Rule 12.1
- [Summit 05] Questions 3.1, 3.2, 3.3, 3.3b, 3.7, 3.8, 3.9, 3.10a, 3.10b, and 3.11
- [Saks 07a]

Exceptions

INT34-EX1: Unsigned integers can exhibit modulo behavior as long as the variable declaration is clearly commented as supporting modulo behavior, and each operation on that integer is also clearly commented as supporting modulo behavior.

If the integer exhibiting modulo behavior contributes to the value of an integer not marked as exhibiting modulo behavior, the resulting integer must obey this rule.

Risk Assessment

Improper range checking can lead to buffer overflows and the execution of arbitrary code by an attacker.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT34-C	high	probable	medium	**P12**	**L1**

References

- [Dowd 06] Chapter 6, "C Language Issues"
- [ISO/IEC 03] Section 6.5.7, "Bitwise Shift Operators"
- [ISO/IEC 9899:1999] Section 6.5.7, "Bitwise Shift Operators"
- [ISO/IEC PDTR 24772] "XYY Wrap-around Error"
- [Seacord 05a] Chapter 5, "Integers"
- [Viega 05] Section 5.2.7, "Integer Overflow"
- A test program for this rule is available at www.securecoding.cert.org/confluence/download/attachments/4385/leftshift.cpp.

INT35-C. Evaluate integer expressions in a larger size before comparing or assigning to that size

If an integer expression is compared to, or assigned to, a larger integer size, that integer expression should be evaluated in that larger size by explicitly casting one of the operands.

Noncompliant Code Example

This noncomplient code example is noncompliant on systems where `size_t` is an unsigned 32-bit value and `long long` is a 64-bit value. In this example, the programmer tests for wrapping by comparing `SIZE_MAX` to `length + BLOCK_HEADER_SIZE`. Because

length is declared as size_t, however, the addition is performed as a 32-bit operation and can result in wrapping. The comparison with SIZE_MAX will always test false. If wrapping occurs, malloc() will allocate insufficient space for mBlock, which can lead to a subsequent buffer overflow.

```
enum { BLOCK_HEADER_SIZE = 16 };

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

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

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

  return mBlock;
}
```

Some compilers will diagnose this condition.

Compliant Solution (Upcast)

In this compliant solution, the length operand is upcast to unsigned long long, ensuring that the addition takes place in this size.

```
enum { BLOCK_HEADER_SIZE = 16 };

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

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

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

  return mBlock;
}
```

This test for wrapping is effective only when the `sizeof(unsigned long long) > sizeof(size_t)`. If both `size_t` and `unsigned long long` types are represented as a 64-bit unsigned value, the result of the addition operation may not be representable as an `unsigned long long` value.

Compliant Solution (Rearrange Expression)

In this compliant solution, `length` is subtracted from `SIZE_MAX`, ensuring that wrapping cannot occur (see INT30-C, "Ensure that unsigned integer operations do not wrap").

```
enum { BLOCK_HEADER_SIZE = 16 };

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

  if (SIZE_MAX - length < BLOCK_HEADER_SIZE) return NULL;
  mBlock = (struct memBlock *)malloc(
    length + BLOCK_HEADER_SIZE
  );
  if (!mBlock) return NULL;

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

  return mBlock;
}
```

Noncompliant Code Example

In this noncompliant code example, the programmer attempts to prevent wrapping by allocating an `unsigned long long` integer called `alloc` and assigning it the result from `cBlocks * 16`.

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

There are two separate problems with this noncompliant code example. The first problem is that this code assumes an implementation where `unsigned long long` has a least four more bits than `size_t`. The second problem, assuming an implementation where `size_t` is a 32-bit value and `unsigned long long` is represented by a 64-bit value, is that to be compliant with C99, multiplying two 32-bit numbers in this context must yield a 32-bit result. Any wrapping resulting from this multiplication will remain undetected by this code, and the expression `alloc < UINT_MAX` will always be true.

Compliant Solution

In this compliant solution, the `cBlocks` operand is upcast to `unsigned long long`, ensuring that the multiplication takes place in this size.

```
static_assert(
  CHAR_BIT * sizeof(unsigned long long) >=
  CHAR_BIT * sizeof(size_t) + 4,
  "Unable to detect wrapping after multiplication"
);

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

Note that this code does not prevent wrapping unless the `unsigned long long` type is at least four bits larger than `size_t`.

Risk Assessment

Failure to cast integers before comparing or assigning them to a larger integer size can result in software vulnerabilities that can allow the execution of arbitrary code by an attacker with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT35-C	high	likely	medium	**P18**	**L1**

References

- [Dowd 06] Chapter 6, "C Language Issues"
- [ISO/IEC 9899:1999] Section 6.3.1, "Arithmetic Operands"
- [ISO/IEC PDTR 24772] "FLC Numeric Conversion Errors"
- [MITRE 07] CWE ID 681, "Incorrect Conversion between Numeric Types," and CWE ID 190, "Integer Overflow (Wrap or Wraparound)"
- [Seacord 05a] Chapter 5, "Integer Security"

References

- [Banahan 03] Section 5.3, "Pointers," and Section 5.7, "Expressions Involving Pointers"
- [ISO/IEC 9899:1999] Section 6.5.6, "Additive Operators"
- [MITRE 07] CWE ID 469, "Use of Pointer Subtraction to Determine Size"
- [VU#162289]

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

Do not add or subtract an integer to a pointer if the resulting value does not refer to an element within the array (or to the nonexistent element just after the last element of the array). According to C99, Section 6.5.6:

> If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

If the pointer resulting from the addition (or subtraction) is outside of the bounds of the array, an overflow has occurred and the result is undefined.

Noncompliant Code Example

In this noncompliant code example, a pointer is set to reference the start of an array. Array elements are accessed sequentially within the for loop. The array pointer `ip` is incremented on each iteration.

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

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

C99 guarantees that it is permissible to use the address of `ar[20]` even though no such element exists. However, in this noncompliant code example, the bound of the array is incorrectly specified, and consequently, the reference to `&ar[21]` constitutes undefined behavior. On the final iteration of the loop, the expression `ip++` (which adds 1 to `ip`) will also overflow.

This code also suffers from using "magic numbers," described in DCL06-C, "Use meaningful symbolic constants to represent literal values in program logic." When replacing the numbers with constants, a developer is likely to catch the invalid array bounds in the `for` statement.

Compliant Solution

This compliant solution fixes the problem from the previous noncompliant code example by using the common idiom `sizeof(ar)/sizeof(ar[0])` to determine the actual number of elements in the array. This idiom works only when the definition of the array is visible (see ARR01-C, "Do not apply the `sizeof` operator to a pointer when taking the size of an array").

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

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

C99 guarantees that it is permissible to use the address of `ar[sizeof(ar)/sizeof(ar[0])]` even though no such element exists. This allows you to use this address for checks in loops like the one above. The guarantee extends only to one element beyond the end of an array and no further [Banahan 03].

Noncompliant Code Example

Pointer arithmetic can result in undefined behavior if the pointer operand and the resulting pointer do not refer to the same array object (or one past the last element of the array object). Compiler implementations are provided broad latitude by the standard in how to deal with undefined behavior (see MSC15-C, "Do not depend on undefined behavior"), including ignoring the situation completely with unpredictable results.

In this noncompliant code example, the programmer is trying to determine if a pointer added to a length will wrap around the end of memory.

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

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

This code resembles the test for wraparound from the `sprint()` function as implemented for the Plan 9 operating system. If `buf + len < buf` evaluates to true, `len` is assigned the remaining space minus 1 byte. However, because the expression `buf + len < buf` constitutes undefined behavior, compilers can assume this condition will never occur and optimize out the entire conditional statement. In gcc version 4.2 and later, for example, code that performs checks for wrapping that depend on undefined behavior (such as the code in this noncompliant code example) are optimized away; no object code to perform the check appears in the resulting executable program [VU#162289]. This is of special concern because it often results in the silent elimination of code that was inserted to provide a safety or security check. For gcc version 4.2.4 and later, this optimization may be disabled with the `-fno-strict-overflow` option.

Compliant Solution (Linear Address Space)

In this compliant solution, references to `buf` are cast to `uintptr_t`. The `uintptr_t` type is an unsigned integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer. Because it is an unsigned type, C99 guarantees that it has modulo behavior. Alternatively, developers can use `size_t` on platforms that do not provide the `uintptr_t` type.

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

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

This compliant solution works on architectures that provide a linear address space and the `uintptr_t` value is a byte address. The latter is not guaranteed, although it is usually the case. Some word-oriented machines are likely to produce a word address with the high-order bits used as a byte selector, in which case this solution will fail. Consequently, this is not a portable solution.

This same compliant solution can be implemented without wrapping:

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

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

Noncompliant Code Example

Another interesting case is shown in this noncompliant code example. The expression `buf + n` may wrap for large values of `n`, resulting in undefined behavior.

```
int process_array(char *buf, size_t n) {
  return buf + n < buf + 100;
}
```

This is an example of how optimization may actually help improve security. When compiled using GCC 4.3.0 with the `-O2` option, for example, the expression `buf + n < buf + 100` is optimized to `n < 100`, eliminating the possibility of wrapping. This code example is still noncompliant because it is not safe to rely on compiler optimizations for security.

Compliant Solution

In this compliant solution, the "optimization" is performed by hand.

```
int process_array(char *buf, size_t n) {
  return n < 100;
}
```

Risk Assessment

If adding or subtracting an integer to a pointer results in a reference to an element outside the array or one past the last element of the array object, the behavior is undefined but frequently leads to a buffer overflow, which can often be exploited to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR38-C	high	likely	medium	P18	L1

Related Vulnerabilities

The following vulnerability resulting from a violation of this rule is documented in the CERT Coordination Center Vulnerability Notes Database [CERT/CC VND].

Metric	ID	Date Public	Name
0	VU#162289	04/17/2006	C compilers may silently discard some wrap-around checks

References

- [Banahan 03] Section 5.3, "Pointers," and Section 5.7, "Expressions Involving Pointers"
- [ISO/IEC 9899:1999] Section 6.5.6, "Additive Operators"
- [ISO/IEC PDTR 24772] "XYX Boundary Beginning Violation" and "XYZ Unchecked Array Indexing"
- [MITRE 07] CWE ID 129, "Unchecked Array Indexing"
- [VU#162289]

Risk Assessment

Failure to correctly determine the size of a wide character string can lead to buffer overflows and the execution of arbitrary code by an attacker.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR33-C	high	likely	medium	P18	L1

References

- [Viega 05] Section 5.2.15, "Improper String Length Checking"
- [ISO/IEC 9899:1999] Section 7.21, "String Handling <string.h>"
- [MITRE 07] CWE ID 119, "Failure to Constrain Operations within the Bounds of an Allocated Memory Buffer," and CWE ID 135, "Incorrect Calculation of Multi-Byte String Length"
- [Seacord 05a] Chapter 2, "Strings"

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

Signed character data must be converted to an unsigned type before being assigned or converted to a larger signed type. Because compilers have the latitude to define `char` to have the same range, representation, and behavior as either `signed char` or `unsigned char`, this rule should be applied to both `signed char` and (plain) `char` characters.

This rule is applicable only in cases where the character data may contain values that can be interpreted as negative values. 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.

Noncompliant Code Example

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

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

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

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

The `string` 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 compilers in which the `char` type defaults to `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`.

This problem was repaired by explicitly declaring the `string` variable as `unsigned char`.

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

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

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

This solution, however, is in violation of STR04-C, "Use plain `char` for characters in the basic character set."

Compliant Solution

In this compliant solution, the result of the expression `*string++` is cast to `(unsigned char)` before assignment to the `int` variable `c`.

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

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

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

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

Risk Assessment

This is a subtle error that results in a disturbingly broad range of potentially severe vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR34-C	medium	probable	medium	P8	L2

References

- [ISO/IEC 9899:1999] Section 6.2.5, "Types"
- [MISRA 04] Rule 6.1, "The plain `char` type shall be used only for the storage and use of character values"
- [MITRE 07] CWE ID 704, "Incorrect Type Conversion or Cast"

STR35-C. Do not copy data from an unbounded source to a fixed-length array

Functions that perform unbounded copies often rely on external input to be a reasonable size. Such assumptions may prove to be false, causing a buffer overflow to occur. For this reason, care must be taken when using functions that may perform unbounded copies.

Noncompliant Code Example (`gets()`)

The `gets()` function 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 compliant code example assumes that `gets()` will not read more than `BUFSIZ - 1` characters from `stdin`. This is an invalid assumption, and the resulting operation can cause a buffer overflow.

According to Section 7.19.7.7 of C99, the `gets()` function reads characters from the `stdin` into a destination array until end-of-file is encountered or a new-line character is

References

- [ISO/IEC 9899:1999] Section 7.20.3.2, "The `free` Function"
- [ISO/IEC PDTR 24772] "DCM Dangling References to Stack Frames" and "XYK Dangling Reference to Heap"
- [Kernighan 88] Section 7.8.5, "Storage Management"
- [MISRA 04] Rule 17.6
- [MITRE 07] CWE ID 416, "Use After Free"
- [OWASP Freed Memory]
- [Seacord 05a] Chapter 4, "Dynamic Memory Management"
- [Viega 05] Section 5.2.19, "Using Freed Memory"

MEM31-C. Free dynamically allocated memory exactly once

Freeing memory multiple times has similar consequences to accessing memory after it is freed. The underlying data structures that manage the heap can become corrupted in a way that can introduce security vulnerabilities into a program. These types of issues are referred to as double-free vulnerabilities. In practice, double-free vulnerabilities can be exploited to execute arbitrary code. VU#623332, which describes a double-free vulnerability in the MIT Kerberos 5 function `krb5_recvauth()` [MIT 05] is one example.

To eliminate double-free vulnerabilities, it is necessary to guarantee that dynamic memory is freed exactly one time. Programmers should be wary when freeing memory in a loop or conditional statement; if coded incorrectly, these constructs can lead to double-free vulnerabilities. It is also a common error to misuse the `realloc()` function in a manner that results in double-free vulnerabilities (see MEM04-C, "Do not perform zero-length allocations").

Noncompliant Code Example

In this noncompliant code example, the memory referred to by `x` may be freed twice: once if `error_condition` is true and again at the end of the code.

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

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

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

Compliant Solution

In this compliant solution, the memory referenced by `x` is freed only once. This is accomplished by eliminating the call to `free()` when `error_condition` is equal to 1.

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

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

Note that this solution checks for numeric overflow (see INT32-C, "Ensure that operations on signed integers do not result in overflow").

Risk Assessment

Freeing memory multiple times can result in an attacker executing arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM31-C	high	probable	medium	**P12**	**L1**

References

- [ISO/IEC PDTR 24772] "XYK Dangling Reference to Heap" and "XYL Memory Leak"
- [MIT 05]
- [MITRE 07] CWE ID 415, "Double Free"

- [OWASP, Double Free]
- [Viega 05] "Doubly Freeing Memory"
- [VU#623332]

MEM32-C. Detect and handle memory allocation errors

The return values for memory allocation routines indicate the failure or success of the allocation. According to C99, `calloc()`, `malloc()`, and `realloc()` return null pointers if the requested memory allocation fails [ISO/IEC 9899:1999]. Failure to detect and properly handle memory management errors can lead to unpredictable and unintended program behavior. As a result, it is necessary to check the final status of memory management routines and handle errors appropriately.

Table 9–1 shows the possible outcomes of the standard memory allocation functions.

Noncompliant Code Example

In this noncompliant code example, `input_string` is copied into dynamically allocated memory referenced by `str`. However, the result of `malloc()` is not checked before `str` is referenced. Consequently, if `malloc()` fails, the program abnormally terminates.

```
char *input_string = /* initialize from untrusted data */;

size_t size = strlen(input_string) + 1;
char *str = (char *)malloc(size);
strcpy(str, input_string);
/* ... */
free(str);
str = NULL;
```

Table 9–1. Possible outcomes of standard memory allocation functions

Function	Successful Return	Error Return
`malloc()`	pointer to allocated space	null pointer
`calloc()`	pointer to allocated space	null pointer
`realloc()`	pointer to the new object	null pointer

FIO30-C. Exclude user input from format strings

Never call any formatted I/O function with a format string containing user input.

An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location and consequently execute arbitrary code with the permissions of the vulnerable process [Seacord 05a].

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

Noncompliant Code Example

This noncompliant code example shows the `incorrect_password()` function, which is called during identification and authentication if the specified user is not found, or the password is incorrect, to display an error message. The function accepts the name of the user as a null-terminated byte string referenced by `user`. This is an excellent example of data that originates from an untrusted, unauthenticated user. The function constructs an error message, which is then output to `stderr` using the C99 standard `fprintf()` function [ISO/IEC 9899:1999].

```
#define MSG_FORMAT "%s cannot be authenticated.\n"
void incorrect_password(const char *user) {
  /* user names are restricted to 256 characters or less */
  static const char *msg_format = MSG_FORMAT;
  size_t len = strlen(user) + sizeof(MSG_FORMAT);
  char *msg = (char *)malloc(len);
  if (!msg) {
    /* Handle error condition */
  }
  int ret = snprintf(msg, len, msg_format, user);
  if (ret < 0 || ret >= len) {\
    /* Handle error */
  }
  fprintf(stderr, msg);
  free(msg);
  msg = NULL;
}
```

The `incorrect_password()` function calculates the size of the message, allocates dynamic storage, and constructs the message in the allocated memory using the `snprintf()` function. The addition operations are not checked for integer overflow because the length of the string referenced by `user` is known to have a length of 256 or less. Because the `%s` characters are replaced by the string referenced by `user` in the call to `snprintf()`, one less

byte is required to store the resulting string and terminating NULL-byte character. This is a common idiom for displaying the same message in multiple locations or when the message is difficult to build. The resulting code contains a format-string vulnerability, however, because the msg includes untrusted user input and is passed as the format-string argument in the call to fprintf().

Compliant Solution (fputs())

This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which does not treat msg like a format string but outputs it to stderr as is.

```
#define MSG_FORMAT "%s cannot be authenticated.\n"
void incorrect_password(const char *user) {
  /* user names are restricted to 256 characters or less */
  static const char *msg_format = MSG_FORMAT;
  size_t len = strlen(user) + sizeof(MSG_FORMAT);
  char *msg = (char *) malloc(len);
  if (!msg) {
    /* Handle error condition */
  }
  int ret = snprintf(msg, len, msg_format, user);
  if (ret < 0 || ret >= len) {\
    /* Handle error */
  }
  if (fputs(msg, stderr) == EOF) {
    /* Handle error */
  }
  free(msg);
  msg = NULL;
}
```

Compliant Solution (fprintf())

This simpler compliant solution passes the untrusted user input as one of the variadic arguments to fprintf() and not as part of the format string, eliminating the possibility of a format-string vulnerability.

```
#define MSG_FORMAT "%s cannot be authenticated.\n"
void incorrect_password(char const *user) {
  fprintf(stderr, MSG_FORMAT user);
}
```

Noncompliant Code Example (POSIX)

This noncompliant code example is exactly the same as the first noncompliant code example but uses the POSIX function syslog() [Open Group 04] instead of the fprintf() function, which is also susceptible to format-string vulnerabilities.

```
#define MSG_FORMAT "%s cannot be authenticated.\n"
void incorrect_password(const char *user) {
  /* user names are restricted to 256 characters or less */
  static const char *msg_format = MSG_FORMAT;
  size_t len = strlen(user) + sizeof(MSG_FORMAT);
  char *msg = (char *)malloc(len);
  if (!msg) {
    /* Handle error condition */
  }
  int ret = snprintf(msg, len, msg_format, user);
  if (ret < 0 || ret >= len) {\
    /* Handle error */
  }
  syslog(LOG_INFO, msg);
  free(msg);
  msg = NULL;
}
```

The `syslog()` function first appeared in BSD 4.2 and is supported by Linux and other modern UNIX implementations. It is not available on Windows systems.

Compliant Solution (POSIX)

This compliant solution passes the untrusted user input as one of the variadic arguments to `syslog()` instead of including it in the format string.

```
#define MSG_FORMAT "%s cannot be authenticated.\n"
void incorrect_password(const char *user) {
  syslog(LOG_INFO, MSG_FORMAT user);
}
```

Risk Assessment

Failing to exclude user input from format specifiers may allow an attacker to crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location, and consequently execute arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO30-C	high	likely	medium	P18	L1

Two recent examples of format-string vulnerabilities resulting from a violation of this rule include Ettercap (ettercap.sourceforge.net/history.php) and Samba (samba.org/samba/security/CVE-2007-0454.html). In Ettercap v.NG-0.7.2, the ncurses user interface suffers from a format string defect. The `curses_msg()` function in `ec_curses.c` calls `wdg_scroll_print()`, which takes a format string and its parameters and passes it to

vw_printw(). The curses_msg() function uses one of its parameters as the format string. This input can include user data, allowing for a format string vulnerability. The Samba AFS ACL mapping VFS plug-in fails to properly sanitize user-controlled file names that are used in a format specifier supplied to snprintf(). This security flaw becomes exploitable when a user can write to a share that uses Samba's afsacl.so library for setting Windows NT access control lists on files residing on an AFS file system.

Related Vulnerabilities. The following vulnerabilities resulting from the violation of this rule are documented in the CERT Coordination Center Vulnerability Notes Database [CERT/CC VND].

Metric	ID	Date Public	Name
1.81	VU#649732	02/05/2007	Samba AFS ACL mapping VFS plug-in format string vulnerability
11.85	VU#794752	01/20/2007	Apple iChat AIM URI handler format string vulnerability
1.8	VU#512491	03/05/2008	GNOME Evolution format string vulnerability
8.11	VU#286468	05/31/2005	Ettercap contains a format string error in the "curses_msg()" function

References

- [ISO/IEC 9899:1999] Section 7.19.6, "Formatted Input/Output Functions"
- [ISO/IEC PDTR 24772] "RST Injection"
- [MITRE 07] CWE ID 134, "Uncontrolled Format String"
- [Open Group 04] syslog()
- [Seacord 05a] Chapter 6, "Formatted Output"
- [Viega 05] Section 5.2.23, "Format String Problem"

■ FIO31-C. Do not simultaneously open the same file multiple times

Simultaneously opening a file multiple times has implementation-defined behavior. While some platforms may forbid a file simultaneously being opened multiple times, platforms that allow it may facilitate dangerous race conditions.

Noncompliant Code Example

This noncompliant code example logs the program's state at runtime.

Compliant Solution

This compliant solution implements a strictly conforming test for unsigned overflow.

```
unsigned int ui1, ui2, sum;

if (UINT_MAX - ui1 < ui2) {
  /* Handle error condition */
}
sum = ui1 + ui2;
```

If the noncompliant form of this test is truly faster, talk to your compiler vendor, because if these tests are equivalent, optimization should occur. If both forms have the same performance, prefer the portable form.

Risk Assessment

Unnecessary platform dependencies are, by definition, unnecessary. Avoiding these dependencies can eliminate porting errors resulting from invalidated assumptions.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC14-C	low	unlikely	medium	**P2**	**L3**

References

- [Dowd 06] Chapter 6, "C Language Issues" (Arithmetic Boundary Conditions, pp. 211–223)
- [ISO/IEC 9899:1999] Section 3.4.1, "Implementation-Defined Behavior," Section 3.4.4, "Unspecified Behavior," Annex J.1, "Unspecified Behavior," and Annex J.3, "Implementation-Defined Behavior"
- [ISO/IEC PDTR 24772] "BQF Unspecified Behaviour"
- [Seacord 05a] Chapter 5, "Integers"

■ MSC15-C. Do not depend on undefined behavior

C99, Section 3.4.3, defines *undefined behavior* as

> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

C99, Section 4, explains how the standard identifies undefined behaviors:

> If a "shall" or "shall not" requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words "undefined behavior" or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe "behavior that is undefined."

C99, Annex J.2, "Undefined Behavior," contains a list of explicit undefined behaviors in C99.

Behavior can be classified as undefined by the C standards committee for the following reasons:

- to give the implementor license not to catch certain program errors that are difficult to diagnose.
- to identify areas of possible conforming language extension: the implementor may augment the language by providing a definition of the officially undefined behavior.

Conforming implementations can deal with undefined behavior in a variety of fashions, such as ignoring the situation completely, with unpredictable results; translating or executing the program in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message); or terminating a translation or execution (with the issuance of a diagnostic message). Because compilers are not obligated to generate code for undefined behavior, these behaviors are candidates for optimization. By assuming that undefined behaviors will not occur, compilers can generate code with better performance characteristics.

Unfortunately, undefined behaviors are coded. Optimizations make it difficult to determine how these systems will behave in the presence of undefined behaviors. This is particularly true when visually inspecting source code that relies on undefined behaviors; a code reviewer cannot be certain if the code will be compiled or if it will be optimized out. Furthermore, just because a compiler currently generates object code for an undefined behavior does not mean that future versions of the compiler are obligated to do the same; the behavior may be viewed as an opportunity for further optimization. Compilers are also not required to issue diagnostics for undefined behavior, so there is frequently no easy way to identify undefined behavior in code.

All of this puts the onus on the programmer to write strictly conforming code, with or without the help of the compiler. Because performance is a primary emphasis of the C language, this situation is likely to get worse before it gets better.

Noncompliant Code Example

An example of undefined behavior in C99 is the behavior on signed integer overflow. This noncompliant code example depends on this behavior to catch the overflow.

```
#include <assert.h>

int foo(int a) {
  assert(a + 100 > a);
  printf("%d %d\n", a + 100, a);
  return a;
}

int main(void) {
  foo(100);
  foo(INT_MAX);
}
```

This code tests for signed integer overflow by testing to see if `a + 100 > a`. This test cannot evaluate to false unless an integer overflow occurs. However, because a conforming implementation is not required to generate code for undefined behavior, and signed integer overflow is undefined behavior, this code may be compiled out. For example, GCC 4.1.1 optimizes out the assertion for all optimization levels, and GCC 4.2.3 optimizes out the assertion for programs compiled with `-O2`-level optimization and higher.

On some platforms, the integer overflow will cause the program to terminate (before it has an opportunity to test).

Compliant Solution

This compliant solution does not depend on undefined behavior.

```
#include <assert.h>

int foo(int a) {
  assert(a < (INT_MAX - 100));
  printf("%d %d\n", a + 100, a);
  return a;
}

int main(void) {
  foo(100);
  foo(INT_MAX);
}
```

Risk Assessment

While it is rare that the entire application can be strictly conforming, the goal should be that almost all the code is allowed for a strictly conforming program (which among other things means that it avoids undefined behavior), with the implementation-dependent parts confined to modules that the programmer knows he or she needs to adapt to the platform when it changes.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC15-C	high	likely	medium	P18	L1

Related Vulnerabilities. The following vulnerability resulting from the violation of this recommendation is documented in the CERT Coordination Center Vulnerability Notes Database [CERT/CC VND].

Metric	ID	Date Public	Name
0	VU#162289	04/17/2006	C compilers may silently discard some wraparound checks

References

- [ISO/IEC 9899:1999] Section 3.4.3, "Undefined Behavior," Section 4, "Conformance," and Annex J.2, "Undefined Behavior"
- [ISO/IEC PDTR 24772] "BQF Unspecified Behaviour," "EWF Undefined Behaviour," and "FAB Implementation-Defined Behaviour"
- [Seacord 05a] Chapter 5, "Integers"

MSC30-C. Do not use the rand() function for generating pseudorandom numbers

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.

The C Standard function `rand()` (available in `stdlib.h`) does not have good random number properties. The numbers generated by `rand()` have a comparatively short cycle, and the numbers may be predictable.

Index