

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

The C Standard identifies the following condition under which division and remainder operations result in [undefined behavior \(UB\)](#):

| UB | Description |
|----|---|
| 45 | The value of the second operand of the / or % operator is zero (6.5.5). |

Ensure that division and remainder operations do not result in 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 two's complement signed integer division when the dividend is equal to the minimum (most negative) value for the signed integer type and the divisor is equal to 1. (See [INT32-C. Ensure that operations on signed integers do not result in overflow.](#))

Noncompliant Code Example

This noncompliant code example prevents signed integer overflow in compliance with [INT32-C. Ensure that operations on signed integers do not result in overflow](#) but fails to prevent a divide-by-zero error during the division of the signed operands `s_a` and `s_b`:

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
    signed long result;
    if ((s_a == LONG_MIN) && (s_b == -1)) {
        /* Handle error */
    } else {
        result = s_a / s_b;
    }
    /* ... */
}
```

Compliant Solution

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

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
    signed long result;
    if ((s_b == 0) || ((s_a == LONG_MIN) && (s_b == -1))) {
        /* Handle error */
    } else {
        result = s_a / s_b;
    }
    /* ... */
}
```

Remainder

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

Noncompliant Code Example

This noncompliant code example prevents signed integer overflow in compliance with [INT32-C. Ensure that operations on signed integers do not result in overflow](#) but fails to prevent a divide-by-zero error during the remainder operation on the signed operands `s_a` and `s_b`:

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
    signed long result;
    if ((s_a == LONG_MIN) && (s_b == -1)) {
        /* Handle error */
    } else {
        result = s_a % s_b;
    }
    /* ... */
}
```

Compliant Solution

This compliant solution tests the remainder operand to guarantee there is no possibility of a divide-by-zero error or an overflow error:

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
    signed long result;
    if ((s_b == 0) || ((s_a == LONG_MIN) && (s_b == -1))) {
        /* Handle error */
    } else {
        result = s_a % s_b;
    }
    /* ... */
}
```

Risk Assessment

A divide-by-zero error can result in [abnormal program termination](#) and denial of service.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---------|----------|------------|------------------|----------|-------|
| INT33-C | Low | Likely | Medium | P6 | L2 |

Automated Detection

| Tool | Version | Checker | Description |
|-------------------------------------|---------|--|---|
| Astrée | 19.04 | int-division-by-zero int-modulo-by-zero | Fully checked |
| CodeSonar | 5.2p0 | LANG.ARITH.DIVZERO LANG.ARITH.FDIVZERO | Division by zero Float Division By Zero |
| Compass /ROSE | | | Can detect some violations of this rule (In particular, it ensures that all operations involving division or modulo are preceded by a check ensuring that the second operand is nonzero.) |
| Coverity | 2017.07 | DIVIDE_BY_ZERO | Fully implemented |
| Cppcheck | 1.66 | zerodiv zerodivcond | Context sensitive analysis of division by zero Not detected for division by struct member / array element / pointer data that is 0 Detected when there is unsafe division by variable before/after test if variable is zero |
| Klocwork | 2018 | DBZ.CONST DBZ.CONST.CALL DBZ.GENERAL DBZ.ITERATOR | |
| LDRA tool suite | 9.7.1 | 43 D, 127 D, 248 S, 629 S, 80 X | Partially implemented |
| Parasoft C /C++test | 10.4.2 | CERT_C-INT33-a | Avoid division by zero |

| | | | |
|-------------------------|--------|--|--|
| Parasoft Insure++ | | | Runtime analysis |
| Polyspace Bug Finder | R2019b | CERT C: Rule INT33-C | Checks for: <ul style="list-style-type: none"> Integer division by zero Tainted division operand Tainted modulo operand Rule fully covered. |
| PRQA QA-C | 9.7 | 2830 [C], 2831 [D], 2832 [A] 2833 [S] | Fully implemented |
| PRQA QA-C++ | 4.4 | 2831, 2832, 2833 | |
| SonarQube C /C++ Plugin | 3.11 | S3518 | |
| PVS-Studio | 6.23 | V609 | |
| TrustInSoft Analyzer | 1.38 | division_by_zero | Exhaustively verified (see one compliant and one non-compliant example). |

Related Vulnerabilities

Search for [vulnerabilities](#) resulting from the violation of this rule on the [CERT website](#).

Related Guidelines

[Key here](#) (explains table format and definitions)

| Taxonomy | Taxonomy item | Relationship |
|---|---|---|
| CERT C | INT32-C. Ensure that operations on signed integers do not result in overflow | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT Oracle Secure Coding Standard for Java | NUM02-J. Ensure that division and remainder operations do not result in divide-by-zero errors | Prior to 2018-01-12: CERT: Unspecified Relationship |
| ISO/IEC TS 17961 | Integer division errors [diverr] | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CWE 2.11 | CWE-369 , Divide By Zero | 2017-07-07: CERT: Exact |

CERT-CWE Mapping Notes

[Key here](#) for mapping notes

CWE-682 and INT33-C

CWE-682 = Union(INT33-C, list) where list =

- Incorrect calculations that do not involve division by zero

Bibliography

| | |
|---------------------------------|-------------------------------|
| [Seacord 2013b] | Chapter 5, "Integer Security" |
| [Warren 2002] | Chapter 2, "Basics" |

