# INT32-C. Ensure that operations on signed integers do not result in overflow

Signed integer overflow is undefined behavior 36. Consequently, implementations have considerable latitude in how they deal with signed integer overflow. (See MSC15-C. Do not depend on undefined behavior.) An implementation that defines signed integer types as being modulo, for example, need not detect integer overflow. Implementations may also trap on signed arithmetic overflows, or simply assume that overflows will never happen and generate object code accordingly. It is also possible for the same conforming implementation to emit code that exhibits different behavior in different contexts. For example, an implementation may determine that a signed integer loop control variable declared in a local scope cannot overflow and may emit efficient code on the basis of that determination, while the same implementation may determine that a global variable used in a similar context will wrap.

For these reasons, it is important to ensure that operations on signed integers do not result in overflow. Of particular importance are operations on signed integer values that originate from a tainted source and are used as

- Integer operands of any pointer arithmetic, including array indexing
- The assignment expression for the declaration of a variable length array
- The postfix expression preceding square brackets `[]` or the expression in square brackets `[]` of a subscripted designation of an element of an array object
- Function arguments of type `size_t` or `rsize_t` (for example, an argument to a memory allocation function)

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

| Operator | Overflow | Operator | Overflow | Operator | Overflow | Operator | Overflow |
|---|---|---|---|---|---|---|---|
| `+` | Yes | `-=` | Yes | `<<` | Yes | `<` | No |
| `-` | Yes | `*=` | Yes | `>>` | No | `>` | No |
| `*` | Yes | `/=` | Yes | `&` | No | `>=` | No |
| `/` | Yes | `%=` | Yes | `\|` | No | `<=` | No |
| `%` | Yes | `<<=` | Yes | `^` | No | `==` | No |
| `++` | Yes | `>>=` | No | `~` | No | `!=` | No |
| `--` | Yes | `&=` | No | `!` | No | `&&` | No |
| `=` | No | `\|=` | No | `unary +` | No | `\|\|` | No |
| `+=` | Yes | `^=` | No | `unary -` | Yes | `?:` | No |

The following sections examine specific operations that are susceptible to integer overflow. When operating on integer types with less precision than `int`, integer promotions are applied. The usual arithmetic conversions may also be applied to (implicitly) convert operands to equivalent types before arithmetic operations are performed. Programmers should understand integer conversion rules before trying to implement secure arithmetic operations. (See INT02-C. Understand integer conversion rules.)

## Implementation Details

GNU GCC invoked with the `-fwrapv` command-line option defines the same modulo arithmetic for both unsigned and signed integers.

GNU GCC invoked with the `-ftrapv` command-line option causes a trap to be generated when a signed integer overflows, which will most likely abnormally exit. On a UNIX system, the result of such an event may be a signal sent to the process.

GNU GCC invoked without either the `-fwrapv` or the `-ftrapv` option may simply assume that signed integers never overflow and may generate object code accordingly.

## Atomic Integers

The C Standard defines the behavior of arithmetic on atomic signed integer types to use two's complement representation with silent wraparound on overflow; there are no undefined results. Although defined, these results may be unexpected and therefore carry similar risks to unsigned integer wrapping. (See INT30-C. Ensure that unsigned integer operations do not wrap.) Consequently, signed integer overflow of atomic integer types should also be prevented or detected.

## Addition

Addition is between two operands of arithmetic type or between a pointer to an object type and an integer type. This rule applies only to addition between two operands of arithmetic type. (See ARR37-C. Do not add or subtract an integer to a pointer to a non-array object and ARR30-C. Do not form or use out-of-bounds pointers or array subscripts.)

Incrementing is equivalent to adding 1.

### Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the addition of the signed operands `si_a` and `si_b`:

```
void func(signed int si_a, signed int si_b) {
  signed int sum = si_a + si_b;
  /* ... */
}
```

### Compliant Solution

This compliant solution ensures that the addition operation cannot overflow, regardless of representation:

```
#include <limits.h>

void f(signed int si_a, signed int si_b) {
  signed int sum;
  if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
      ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
    /* Handle error */
  } else {
    sum = si_a + si_b;
  }
  /* ... */
}
```

## Subtraction

Subtraction is between two operands of arithmetic type, two pointers to qualified or unqualified versions of compatible object types, or a pointer to an object type and an integer type. This rule applies only to subtraction between two operands of arithmetic type. (See ARR36-C. Do not subtract or compare two pointers that do not refer to the same array, ARR37-C. Do not add or subtract an integer to a pointer to a non-array object, and ARR30-C. Do not form or use out-of-bounds pointers or array subscripts for information about pointer subtraction.)

Decrementing is equivalent to subtracting 1.

### Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the subtraction of the signed operands `si_a` and `si_b`:

```
void func(signed int si_a, signed int si_b) {
  signed int diff = si_a - si_b;
  /* ... */
}
```

### Compliant Solution

This compliant solution tests the operands of the subtraction to guarantee there is no possibility of signed overflow, regardless of representation:

```
#include <limits.h>

void func(signed int si_a, signed int si_b) {
  signed int diff;
  if ((si_b > 0 && si_a < INT_MIN + si_b) ||
      (si_b < 0 && si_a > INT_MAX + si_b)) {
    /* Handle error */
  } else {
    diff = si_a - si_b;
  }

  /* ... */
}
```

## Multiplication

Multiplication is between two operands of arithmetic type.

### Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the multiplication of the signed operands `si_a` and `si_b`:

```
void func(signed int si_a, signed int si_b) {
  signed int result = si_a * si_b;
  /* ... */
}
```

### Compliant Solution

The product of two operands can always be represented using twice the number of bits than exist in the precision of the larger of the two operands. This compliant solution eliminates signed overflow on systems where `long long` is at least twice the precision of `int`:

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

extern size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)

void func(signed int si_a, signed int si_b) {
  signed int result;
  signed long long tmp;
  assert(PRECISION(ULLONG_MAX) >= 2 * PRECISION(UINT_MAX));
  tmp = (signed long long)si_a * (signed long long)si_b;

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

The assertion fails if `long long` has less than twice the precision of `int`. The `PRECISION()` macro and `popcount()` function provide the correct precision for any integer type. (See INT35-C. Use correct integer precisions.)

### Compliant Solution

The following portable compliant solution can be used with any conforming implementation, including those that do not have an integer type that is at least twice the precision of `int`:

```
#include <limits.h>

void func(signed int si_a, signed int si_b) {
  signed int result;
  if (si_a > 0) {  /* si_a is positive */
    if (si_b > 0) {  /* si_a and si_b are positive */
      if (si_a > (INT_MAX / si_b)) {
        /* Handle error */
      }
    } else { /* si_a positive, si_b nonpositive */
      if (si_b < (INT_MIN / si_a)) {
        /* Handle error */
      }
    } /* si_a positive, si_b nonpositive */
  } else { /* si_a is nonpositive */
    if (si_b > 0) { /* si_a is nonpositive, si_b is positive */
      if (si_a < (INT_MIN / si_b)) {
        /* Handle error */
      }
    } else { /* si_a and si_b are nonpositive */
      if ( (si_a != 0) && (si_b < (INT_MAX / si_a))) {
        /* Handle error */
      }
    } /* End if si_a and si_b are nonpositive */
  } /* End if si_a is nonpositive */

  result = si_a * si_b;
}
```

## Division

Division is between two operands of arithmetic type. Overflow can occur during two's 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`. Division operations are also susceptible to divide-by-zero errors. (See INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors.)

### Noncompliant Code Example

This noncompliant code example prevents divide-by-zero errors in compliance with INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors but does not prevent a signed integer overflow error in two's-complement.

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

### Implementation Details

On the x86-32 architecture, overflow results in a fault, which can be exploited as a denial-of-service attack.

### Compliant Solution

This compliant solution eliminates the 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. Because many platforms implement remainder and division in the same instruction, the remainder operator is also susceptible to arithmetic overflow and division by zero. (See INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors.)

### Noncompliant Code Example

Many hardware architectures implement remainder as part of the division operator, which can overflow. Overflow can occur during a remainder operation when the dividend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to 1. It occurs even though the result of such a remainder operation is mathematically 0. This noncompliant code example prevents divide-by-zero errors in compliance with INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors but does not prevent integer overflow:

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

### Implementation Details

On x86-32 platforms, the remainder operator for signed integers is implemented by the `idiv` instruction code, along with the divide operator. Because `LONG_MIN / 1` overflows, it results in a software exception with `LONG_MIN % 1` as well.

### Compliant Solution

This compliant solution also tests the remainder operands to guarantee there is no possibility of an 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;
  }
  /* ... */
}
```

## Left-Shift Operator

The left-shift operator takes two integer operands. The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros.

The C Standard, 6.5.7, paragraph 4 [ISO/IEC 9899:2011], states

> *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.*

In almost every case, an attempt to shift by a negative number of bits or by more bits than exist in the operand indicates a logic error. These issues are covered by INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.

### Noncompliant Code Example

This noncompliant code example performs a left shift, after verifying that the number being shifted is not negative, and the number of bits to shift is valid. The `PRECISION()` macro and `popcount()` function provide the correct precision for any integer type. (See INT35-C. Use correct integer precisions.) However, because this code does no overflow check, it can result in an unrepresentable value.

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

extern size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)

void func(signed long si_a, signed long si_b) {
  signed long result;
  if ((si_a < 0) || (si_b < 0) ||
      (si_b >= PRECISION(ULONG_MAX)) {
    /* Handle error */
  } else {
    result = si_a << si_b;
  }
  /* ... */
}
```

### Compliant Solution

This compliant solution eliminates the possibility of overflow resulting from a left-shift operation:

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

extern size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)

void func(signed long si_a, signed long si_b) {
  signed long result;
  if ((si_a < 0) || (si_b < 0) ||
      (si_b >= PRECISION(ULONG_MAX)) ||
      (si_a > (LONG_MAX >> si_b))) {
    /* Handle error */
  } else {
    result = si_a << si_b;
  }
  /* ... */
}
```

## Unary Negation

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

### Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the unary negation of the signed operand s_a:

```
void func(signed long s_a) {
  signed long result = -s_a;
  /* ... */
}
```

### Compliant Solution

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

```
#include <limits.h>

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

## 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 | High | Likely | High | P9 | L2 |

### Automated Detection

| Tool | Version | Checker | Description |
|---|---|---|---|
| Astrée | 18.10 | **integer-overflow** | Fully checked |
| CodeSonar | 5.0p0 | **ALLOC.SIZE.ADDOFLOW**<br>**ALLOC.SIZE.IOFLOW**<br>**ALLOC.SIZE.MULOFLOW**<br>**ALLOC.SIZE.SUBUFLOW**<br>**MISC.MEM.SIZE.ADDOFLOW**<br>**MISC.MEM.SIZE.BAD**<br>**MISC.MEM.SIZE.MULOFLOW**<br>**MISC.MEM.SIZE.SUBUFLOW** | Addition overflow of allocation size<br>Integer overflow of allocation size<br>Multiplication overflow of allocation size<br>Subtraction underflow of allocation size<br>Addition overflow of size<br>Unreasonable size argument<br>Multiplication overflow of size<br>Subtraction underflow of size |
| Coverity | 2017.07 | **TAINTED_SCALAR**<br><br>**BAD_SHIFT** | Implemented |
| LDRA tool suite | 9.7.1 | **493 S, 494 S** | Partially implemented |
| Parasoft C/C++test | 10.4 | **CERT_C-INT32-a**<br>**CERT_C-INT32-b**<br>**CERT_C-INT32-c** | Avoid integer overflows<br>Integer overflow or underflow in constant expression in '+', '-', '*' operator<br>Integer overflow or underflow in constant expression in '<<' operator |
| Parasoft Insure++ | | | Runtime analysis |

| | | | |
|---|---|---|---|
| Polyspace Bug Finder | R2018a | Integer overflow<br><br>Tainted division operand<br><br>Tainted modulo operand | Overflow from operation between integers<br><br>Division (`/`) operands from an unsecure source<br><br>Remainder (`%`) operands are from an unsecure source |
| PRQA QA-C | 9.3 | **2800, 2801, 2802, 2803,2860,2861,2862,2863** | Fully implemented |
| PRQA QA-C++ | 4.1 | **2791, 2792, 2793, 2800, 2801, 2802, 2803** | |

## 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 | INT02-C. Understand integer conversion rules | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT C | INT35-C. Use correct integer precisions | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT C | INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT C | INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT C | ARR30-C. Do not form or use out-of-bounds pointers or array subscripts | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT C | ARR36-C. Do not subtract or compare two pointers that do not refer to the same array | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT C | ARR37-C. Do not add or subtract an integer to a pointer to a non-array object | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT C | MSC15-C. Do not depend on undefined behavior | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT C | CON08-C. Do not assume that a group of calls to independently atomic methods is atomic | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CERT Oracle Secure Coding Standard for Java | INT00-J. Perform explicit range checking to avoid integer overflow | Prior to 2018-01-12: CERT: Unspecified Relationship |
| ISO/IEC TR 24772:2013 | Arithmetic Wrap-Around Error [FIF] | Prior to 2018-01-12: CERT: Unspecified Relationship |
| ISO/IEC TS 17961 | Overflowing signed integers [intoflow] | Prior to 2018-01-12: CERT: Unspecified Relationship |
| CWE 2.11 | CWE-190, Integer Overflow or Wraparound | 2017-05-18: CERT: Partial overlap |
| CWE 2.11 | CWE-191 | 2017-05-18: CERT: Partial overlap |
| CWE 2.11 | CWE-680 | 2017-05-18: CERT: Partial overlap |

# CERT-CWE Mapping Notes

Key here for mapping notes

## CWE-20 and INT32-C

See CWE-20 and ERR34-C

## CWE-680 and INT32-C

Intersection( INT32-C, MEM35-C) = Ø

Intersection( CWE-680, INT32-C) =

- Signed integer overflows that lead to buffer overflows

CWE-680 - INT32-C =

- Unsigned integer overflows that lead to buffer overflows

INT32-C – CWE-680 =

- Signed integer overflows that do not lead to buffer overflows

## CWE-191 and INT32-C

Union( CWE-190, CWE-191) = Union( INT30-C, INT32-C)

Intersection( INT30-C, INT32-C) == Ø

Intersection(CWE-191, INT32-C) =

- Underflow of signed integer operation

CWE-191 – INT32-C =

- Underflow of unsigned integer operation

INT32-C – CWE-191 =

- Overflow of signed integer operation

## CWE-190 and INT32-C

Union( CWE-190, CWE-191) = Union( INT30-C, INT32-C)

Intersection( INT30-C, INT32-C) == Ø

Intersection(CWE-190, INT32-C) =

- Overflow (wraparound) of signed integer operation

CWE-190 – INT32-C =

- Overflow of unsigned integer operation

INT32-C – CWE-190 =

- Underflow of signed integer operation

# Bibliography


| | |
|---|---|
| [Dowd 2006] | Chapter 6, "C Language Issues" ("Arithmetic Boundary Conditions," pp. 211–223) |
| [ISO/IEC 9899:2011] | Subclause 6.5.5, "Multiplicative Operators" |

| | |
|---|---|
| [Seacord 2013b] | Chapter 5, "Integer Security" |
| [Viega 2005] | Section 5.2.7, "Integer Overflow" |
| [Warren 2002] | Chapter 2, "Basics" |