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

Integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data. This rule is particularly true for integer values that originate from untrusted sources and are used in any of the following ways:

- 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)

This rule also applies to arguments passed to the following library functions that are converted to unsigned char:

- memset()
- memset_s()
- fprintf() and related functions (For the length modifier c, if no 1 length modifier is present, the int argument is converted to an unsigned char, and the resulting character is written.)
- fputc()
- ungetc()
- memchr()

and to arguments to the following library functions that are converted to char:

- strchr()
- strrchr()
- All of the functions listed in <ctype.h>

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. The C Standard, subclause 6.3.1.3 [ISO/IEC 9899:2011], says

> 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.

Noncompliant Code Example (Unsigned to Signed)

Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of an unsigned integer type to a value of a signed integer type. This noncompliant code example results in a truncation error on most implementations:

```c
#include <limits.h>

void func(void) {
    unsigned long int u_a = ULONG_MAX;
    signed char sc;
    sc = (signed char)u_a; /* Cast eliminates warning */
    /* ... */
}
```

Compliant Solution (Unsigned to Signed)

Validate ranges when converting from an unsigned type to a signed type. This compliant solution can be used to convert a value of unsigned long int type to a value of signed char type:

```c
#include <limits.h>

void func(void) {
    unsigned long int u_a = ULONG_MAX;
    signed char sc;
    sc = static_cast<signed char>(u_a); /* Cast eliminates warning */
    /* ... */
}
```
```c
#include <limits.h>

void func(void) {
    unsigned long int u_a = ULONG_MAX;
    signed char sc;
    if (u_a <= SCHAR_MAX) {
        sc = (signed char)u_a; /* Cast eliminates warning */
    } else {
        /* Handle error */
    }
}
```

**Noncompliant Code Example (Signed to Unsigned)**

Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of a signed type to a value of an unsigned type. This noncompliant code example results in a negative number being misinterpreted as a large positive number.

```c
#include <limits.h>

void func(signed int si) {
    /* Cast eliminates warning */
    unsigned int ui = (unsigned int)si;
    /* ... */
}
/* ... */
func(INT_MIN);
```

**Compliant Solution (Signed to Unsigned)**

Validate ranges when converting from a signed type to an unsigned type. This compliant solution converts a value of a signed int type to a value of an unsigned int type:

```c
#include <limits.h>

void func(signed int si) {
    unsigned int ui;
    if (si < 0) {
        /* Handle error */
    } else {
        ui = (unsigned int)si; /* Cast eliminates warning */
    }
    /* ... */
}
/* ... */
func(INT_MIN + 1);
```

Subclause 6.2.5, paragraph 9, of the C Standard [ISO/IEC 9899:2011] provides the necessary guarantees to ensure this solution works on a conforming implementation:

> The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.

**Noncompliant Code Example (Signed, Loss of Precision)**

A loss of data (truncation) can occur when converting from a value of a signed integer type to a value of a signed type with less precision. This noncompliant code example results in a truncation error on most implementations:

```c
#include <limits.h>

void func(signed int si) {
    unsigned int ui;
    if (si < 0) {
        /* Handle error */
    } else {
        ui = (unsigned int)si; /* Cast eliminates warning */
    }
    /* ... */
}
/* ... */
func(INT_MIN);
```
#include <limits.h>

void func(void) {
    signed long int s_a = LONG_MAX;
    signed char sc = (signed char)s_a; /* Cast eliminates warning */
    /* ... */
}

Compliant Solution (Signed, Loss of Precision)

Validate ranges when converting from a signed type to a signed type with less precision. This compliant solution converts a value of a signed long int type to a value of a signed char type:

#include <limits.h>

void func(void) {
    signed long int s_a = LONG_MAX;
    signed char sc;
    if ((s_a < SCHAR_MIN) || (s_a > SCHAR_MAX)) {
        /* Handle error */
    } else {
        sc = (signed char)s_a; /* Use cast to eliminate warning */
    }
    /* ... */
}

Conversions from a value of a signed integer type to a value of a signed integer type with less precision requires that both the upper and lower bounds are checked.

Noncompliant Code Example (Unsigned, Loss of Precision)

A loss of data (truncation) can occur when converting from a value of an unsigned integer type to a value of an unsigned type with less precision. This noncompliant code example results in a truncation error on most implementations:

#include <limits.h>

void func(void) {
    unsigned long int u_a = ULONG_MAX;
    unsigned char uc = (unsigned char)u_a; /* Cast eliminates warning */
    /* ... */
}

Compliant Solution (Unsigned, Loss of Precision)

Validate ranges when converting a value of an unsigned integer type to a value of an unsigned integer type with less precision. This compliant solution converts a value of an unsigned long int type to a value of an unsigned char type:

#include <limits.h>

void func(void) {
    unsigned long int u_a = ULONG_MAX;
    unsigned char uc;
    if (u_a > UCHAR_MAX) {
        /* Handle error */
    } else {
        uc = (unsigned char)u_a; /* Cast eliminates warning */
    }
    /* ... */
}

Conversions from unsigned types with greater precision to unsigned types with less precision require only the upper bounds to be checked.
Noncompliant Code Example \((\text{time\_t Return Value})\)

The \texttt{time()} function returns the value \((-1)\) to indicate that the calendar time is not available. The C Standard requires that the \texttt{time\_t} type is only a real type capable of representing time. (The integer and real floating types are collectively called real types.) It is left to the implementor to decide the best real type to use to represent time. If \texttt{time\_t} is implemented as an unsigned integer type with less precision than a signed \texttt{int}, the return value of \texttt{time()} will never compare equal to the integer literal \(-1\).

```c
#include <time.h>

void func(void) {
    time_t now = time(NULL);
    if (now != -1) {
        /* Continue processing */
    }
}
```

Compliant Solution \((\text{time\_t Return Value})\)

To ensure the comparison is properly performed, the return value of \texttt{time()} should be compared against \(-1\) cast to type \texttt{time\_t}:

```c
#include <time.h>

void func(void) {
    time_t now = time(NULL);
    if (now != (time_t)-1) {
        /* Continue processing */
    }
}
```

This solution is in accordance with INT18-C. Evaluate integer expressions in a larger size before comparing or assigning to that size. Note that \((\text{time\_t})\) \(-1\) also complies with INT31-C-EX3.

Noncompliant Code Example \((\text{memset()})\)

For historical reasons, certain C Standard functions accept an argument of type \texttt{int} and convert it to either \texttt{unsigned char} or plain \texttt{char}. This conversion can result in unexpected behavior if the value cannot be represented in the smaller type. The second argument to \texttt{memset()} is an example; it indicates what byte to store in the range of memory indicated by the first and third arguments. If the second argument is outside the range of a signed \texttt{char} or plain \texttt{char}, then its higher order bits will typically be truncated. Consequently, this noncompliant solution unexpectedly sets all elements in the array to 0, rather than 4096:

```c
#include <string.h>
#include <stddef.h>

int *init_memory(int *array, size_t n) {
    return memset(array, 4096, n);
}
```

Compliant Solution \((\text{memset()})\)

In general, the \texttt{memset()} function should not be used to initialize an integer array unless it is to set or clear all the bits, as in this compliant solution:

```c
#include <string.h>
#include <stddef.h>

int *init_memory(int *array, size_t n) {
    return memset(array, 0, n);
}
```

Exceptions
The C Standard defines minimum ranges for standard integer types. For example, the minimum range for an object of type `unsigned short int` is 0 to 65,535, whereas the minimum range for `int` is 32,767 to +32,767. Consequently, 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 to +2,147,483,647, meaning that it is quite possible to represent all the values of an `unsigned short int` as an `int` for this architecture. 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. A good way to document these assumptions is to use static assertions. (See DCL03-C: Use a static assertion to test the value of a constant expression.)

Conversion from any integer type with a value between `SCHAR_MIN` and `UCHAR_MAX` to a character type is permitted provided the value represents a character and not an integer.

Conversions to unsigned character types are well defined by C to have modular behavior. A character's value is not misinterpreted by the loss of sign or conversion to a negative number. For example, the Euro symbol € is sometimes represented by bit pattern `0x80` which can have the numerical value 128 or 127 depending on the signedness of the type.

Conversions to signed character types are more problematic. The C Standard, subclause 6.3.1.3, paragraph 3 [ISO/IEC 9899:2011], says, regarding conversions

> 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.

Furthermore, subclause 6.2.6.2, paragraph 2, says, regarding integer modifications

> If the sign bit is one, the value shall be modified in one of the following ways:
> — the corresponding value with sign bit 0 is negated (sign and magnitude)
> — the sign bit has the value (2M ) (two's complement);
> — the sign bit has the value (2M  1) (ones' complement).

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value. [See note.]

NOTE: Two's complement is shorthand for "radix complement in radix 2." Ones' complement is shorthand for "diminished radix complement in radix 2."

Consequently, the standard allows for this code to trap:

```c
int i = 128; /* 1000 0000 in binary */
assert(SCHAR_MAX == 127);
signed char c = i; /* can trap */
```

However, platforms where this code traps or produces an unexpected value are rare. According to The New C Standard: An Economic and Cultural Commentary by Derek Jones [Jones 2008],

> Implementations with such trap representations are thought to have existed in the past. Your author was unable to locate any documents describing such processors.

ISO C, section 7.27.2.4, paragraph 3 says:

> The time function returns the implementation's best approximation to the current calendar time.

> The value (time_t) (1) is returned if the calendar time is not available.

If `time_t` is an unsigned type, then the expression `((time_t) (-1))` is guaranteed to yield a large positive value.

Therefore, conversion of a negative compile-time constant to an unsigned value with the same or larger width is permitted by this rule. This exception does not apply to conversion of unsigned to signed values, nor does it apply if the resulting value would undergo truncation.

Risk Assessment

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT31-C</td>
<td>High</td>
<td>Probable</td>
<td>High</td>
<td>P6</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection
<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>22.04</td>
<td></td>
<td>Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>7.1p0</td>
<td>LANG.CAST.PC.AV, LANG.CAST.PC.CONST2PTR, LANG.CAST.PC.INT, LANG.CAST.COERCCE, LANG.CAST.VALUE, ALLOC.SIZE.TRUNC, MISC.MEM.SIZE.TRUNC, LANG.MEM.TBA</td>
<td>Cast: arithmetic type/void pointer Conversion: integer constant to pointer Conversion: pointer/integer Coercion alters value Cast alters value Truncation of allocation size Truncation of size Tainted buffer access</td>
</tr>
<tr>
<td>Compas/Rose</td>
<td></td>
<td></td>
<td>Can detect violations of this rule. However, false warnings may be raised if limits.h is included</td>
</tr>
<tr>
<td>Coverity*</td>
<td>2017.07</td>
<td>NEGATIVE_RETURNS, REVERSE_NEGATIVE, MISRA.Cast</td>
<td>Can find array accesses, loop bounds, and other expressions that may contain dangerous implied integer conversions that would result in unexpected behavior Can find instances where a negativity check occurs after the negative value has been used for something else Can find instances where an integer expression is implicitly converted to a narrower integer type, where the signedness of an integer value is implicitly converted, or where the type of a complex expression is implicitly converted</td>
</tr>
<tr>
<td>Cppcheck</td>
<td>1.66</td>
<td>memsetValueOutOfRange</td>
<td>The second argument to memset() cannot be represented as unsigned char</td>
</tr>
</tbody>
</table>
| Helix QAC               | 2022.2  | C2850, C2851, C2852, C2853, C2855, C2856, C2857, C2858, C2890, C2891, C2892, C2893, C2895, C2896, C2897, C2898, C2900, C2901, C2902, C2903, C2905, C2906, C2907, C2908, C++2850, C++2851, C++2852, C++2853, C++2855, C++2856, C++2857, C++2858, C++2890, C++2891, C++2892, C++2893, C++2895, C++2896, C++2897, C++2898, C++2900, C++2901, C++2902, C++2903, C++2905, C++2906, C++2908, C++2909, C++2910 | An expression of essentially Boolean type should always be used where an operand is interpreted as a Boolean value An operand of essentially Boolean type should not be used where an operand is interpreted as a numeric value An operand of essentially character type should not be used where an operand is interpreted as a numeric value An operand of essentially enum type should not be used in an arithmetic operation Shift and bitwise operations should not be performed on operands of essentially signed or enum type An operand of essentially signed or enum type should not be used as the right hand operand to the bitwise shifting operator An operand of essentially unsigned type should not be used as the operand to the unary minus operator The value of an expression shall not be assigned to an object with a narrower essential type The value of an expression shall not be assigned to an object of a different essential type category Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category The second and third operands of the ternary operator shall have the same essential type category The value of a composite expression shall not be assigned to an object with wider essential type If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type If a composite expression is used as one (second or third) operand of a conditional operator then the other operand shall not have wider essential type Avoid integer overflows |}
| Klocwork                | 2022.2  | PORTING.CAST SIZE                           | Partially implemented                                                                                                                                                                                                                                                                                                                                                                                                                    |
| LDRA tool suite         | 9.7.1   | 93 S, 433 S, 434 S                          | Partially implemented                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Parasoft C/C++test      | 2022.1  | CERT_C-INT31-a, CERT_C-INT31-b, CERT_C-INT31-c, CERT_C-INT31-d, CERT_C-INT31-e, CERT_C-INT31-f, CERT_C-INT31-g, CERT_C-INT31-h, CERT_C-INT31-i, CERT_C-INT31-j, CERT_C-INT31-k, CERT_C-INT31-l, CERT_C-INT31-m, CERT_C-INT31-n, CERT_C-INT31-o | An expression of essentially Boolean type should always be used where an operand is interpreted as a Boolean value An operand of essentially Boolean type should not be used where an operand is interpreted as a numeric value An operand of essentially character type should not be used where an operand is interpreted as a numeric value An operand of essentially enum type should not be used in an arithmetic operation Shift and bitwise operations should not be performed on operands of essentially signed or enum type An operand of essentially signed or enum type should not be used as the right hand operand to the bitwise shifting operator An operand of essentially unsigned type should not be used as the operand to the unary minus operator The value of an expression shall not be assigned to an object with a narrower essential type The value of an expression shall not be assigned to an object of a different essential type category Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category The second and third operands of the ternary operator shall have the same essential type category The value of a composite expression shall not be assigned to an object with wider essential type If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type If a composite expression is used as one (second or third) operand of a conditional operator then the other operand shall not have wider essential type Avoid integer overflows |}
<p>| Polyspace Bug Finder    | R2022b  | CERT C: Rule INT31-C                        | Checks for: Integer conversion overflow Call to memset with unintended value Sign change integer conversion overflow Tainted sign change conversion Unsigned integer conversion overflow Rule partially covered. |</p>
<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C</td>
<td>DCL03-C. Use a static assertion to test the value of a constant expression</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C</td>
<td>INT18-C. Evaluate integer expressions in a larger size before comparing or assigning to that size</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C</td>
<td>FIO34-C. Distinguish between characters read from a file and EOF or WEOF</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>NUM12-J. Ensure conversions of numeric types to narrower types do not result in lost or misinterpreted data</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>MISRA C:2012</td>
<td>Rule 10.3 (required)</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>MISRA C:2012</td>
<td>Rule 10.6 (required)</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>MISRA C:2012</td>
<td>Rule 10.7 (required)</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-192, Integer Coercion Error</td>
<td>2017-07-17: CERT: Exact</td>
</tr>
</tbody>
</table>

* Coverity Prevent cannot discover all violations of this rule, so further verification is necessary.

**Related Vulnerabilities**

CVE-2009-1376 results from a violation of this rule. In version 2.5.5 of Pidgin, a `size_t` offset is set to the value of a 64-bit unsigned integer, which can lead to truncation [xort 2009] on platforms where a `size_t` is implemented as a 32-bit unsigned integer. An attacker can execute arbitrary code by carefully choosing this value and causing a buffer overflow.

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

**Related Guidelines**

**Key here** (explains table format and definitions)
CERT-CWE Mapping Notes

Key here for mapping notes

CWE-195 and INT31-C

CWE-195 = Subset( CWE-192)
INT31-C = Union( CWE-195, list) where list =
  • Unsigned-to-signed conversion error
  • Truncation that does not change sign

CWE-197 and INT31-C

See CWE-197 and FLP34-C

CWE-194 and INT31-C

CWE-194 = Subset( CWE-192)
INT31-C = Union( CWE-194, list) where list =
  • Integer conversion that truncates significant data, but without loss of sign

CWE-20 and INT31-C

See CWE-20 and ERR34-C

CWE-704 and INT31-C

CWE-704 = Union( INT31-C, list) where list =
  • Improper type casts where either the source or target type is not an integral type

CWE-681 and INT31-C

CWE-681 = Union( INT31-C, FLP34-C)
Intersection( INT31-C, FLP34-C) = Ø

Bibliography

[Jones 2008] Section 6.2.6.2, "Integer Types"
[Seacord 2013b] Chapter 5, "Integer Security"
[Viega 2005] Section 5.2.9, "Truncation Error"
Section 5.2.10, "Sign Extension Error"
Section 5.2.11, "Signed to Unsigned Conversion Error"
Section 5.2.12, "Unsigned to Signed Conversion Error"