INT18-C. Evaluate integer expressions in a larger size before comparing or assigning to that size

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

Noncompliant Code Example

This code example is noncompliant 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 wrapping by comparing SIZE_MAX to length + BLOCK_HEADER_SIZE. Because length is declared as size_t, the addition is performed as a 32-bit operation and can result in wrapping. The comparison with SIZE_MAX will always test false. If any wrapping occurs, malloc() will allocate insufficient space for mBlock, which can lead to a subsequent buffer overflow.

```c
#include <stdlib.h>
#include <stdint.h> /* For SIZE_MAX */
enum { BLOCK_HEADER_SIZE = 16 };
void *AllocateBlock(size_t length) {
    struct memBlock *mBlock;
    if (length + BLOCK_HEADER_SIZE > (unsigned long long)SIZE_MAX)
        return NULL;
    mBlock = (struct memBlock *)malloc(
        length + BLOCK_HEADER_SIZE
    );
    if (!mBlock) { return NULL; }
    /* Fill in block header and return data portion */
    return mBlock;
}
```

Some compilers will diagnose this condition.

Compliant Solution (Upcast)

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

```c
#include <stdlib.h>
#include <stdint.h>
enum { BLOCK_HEADER_SIZE = 16 };
void *AllocateBlock(size_t length) {
    struct memBlock *mBlock;
    if ((unsigned long long)length + BLOCK_HEADER_SIZE > SIZE_MAX) {
        return NULL;
    } 
    mBlock = (struct memBlock *)malloc(
        length + BLOCK_HEADER_SIZE
    );
    if (!mBlock) { return NULL; }
    /* Fill in block header and return data portion */
    return mBlock;
}
```

This test for wrapping is effective only when the sizeof(unsigned long long) > sizeof(size_t). If both size_t and unsigned long long types are represented as 64-bit unsigned values, the result of the addition operation may not be representable as an unsigned long long value.

Compliant Solution (Rearrange Expression)
In this compliant solution, length is subtracted from SIZE_MAX, ensuring that wrapping cannot occur. See INT30-C. Ensure that unsigned integer operations do not wrap.

```c
#include <stdlib.h>
#include <stdint.h>

enum { BLOCK_HEADER_SIZE = 16 };

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

    if (SIZE_MAX - length < BLOCK_HEADER_SIZE) return NULL;
    mBlock = (struct memBlock *)malloc(length + BLOCK_HEADER_SIZE);
    if (!mBlock) { return NULL; }
    /* Fill in block header and return data portion */
    return mBlock;
}
```

Noncompliant Code Example

In this noncompliant code example, the programmer attempts to prevent wrapping by allocating an `unsigned long long` integer called `alloc` and assigning it the result from `cBlocks * 16`:

```c
#include <stdlib.h>
#include <limits.h>

void *AllocBlocks(size_t cBlocks) {
    if (cBlocks == 0) { return NULL; }
    unsigned long long alloc = cBlocks * 16;
    return (alloc < UINT_MAX) ? malloc(cBlocks * 16) : NULL;
}
```

Two problems occur in this noncompliant code example. The first problem is that this code assumes an implementation where `unsigned long long` has at least 4 more bits than `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 to be compliant with C, multiplying two 32-bit numbers in this context must yield a 32-bit result. Any wrapping 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:

```c
#include <stdlib.h>
#include <assert.h>
#include <limits.h>

static_assert(Char_BIT * sizeof(unsigned long long) >=
    Char_BIT * sizeof(size_t) + 4,
    "Unable to detect wrapping after multiplication");

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 does not prevent wrapping unless the `unsigned long long` type is at least 4 bits larger than `size_t`.

Noncompliant Code Example (`size_t`)
The mbstowcs() function converts a multibyte string to a wide character string, returning the number of characters converted. If an invalid multibyte 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.

```c
#include <stdlib.h>

void func(wchar_t *pwcs, const char *restrict s, size_t n) {
    size_t count_modified = mbstowcs(pwcs, s, n);
    if (count_modified == (size_t)-1) {
        /* Handle error */
    }
}
```

Compliant Solution (size_t)

To ensure the comparison is properly performed, the return value of mbstowcs() should be compared against -1 cast to type size_t:

```c
#include <stdlib.h>

void func(wchar_t *pwcs, const char *restrict s, size_t n) {
    size_t count_modified = mbstowcs(pwcs, s, n);
    if (count_modified == (size_t)-1) {
        /* Handle error */
    }
}
```

Risk Assessment

Failure to cast integers before comparing or assigning them 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.

<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>INT18-C</td>
<td>high</td>
<td>likely</td>
<td>medium</td>
<td>P18</td>
<td>L1</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>19.04</td>
<td></td>
<td>Supported, but no explicit checker</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>LANG.TYPE.AWID LANG.TYPE.OWID</td>
<td>Expression value widened by assignment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LANG.CAST.VALUE LANG.CAST.COERCCE</td>
<td>Expression value widened by other operand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.JOFLOW</td>
<td>Cast alters value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW ALLOC.SIZE.TRUNC MISC.MEM.SIZE. ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW MISC.MEM.SIZE.TRUNC</td>
<td>Coercion alters value</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td>Can detect violations of this rule. It should look for patterns of (a op1 b) op2 c where</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• c has a bigger type than a or b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Neither a nor b is typecast to c's type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• op2 is assignment or comparison</td>
</tr>
<tr>
<td>Tool Suite</td>
<td>Version</td>
<td>Rule(s)</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Coverity</td>
<td>6.5</td>
<td>OVERFLOW_BEFORE_WIDEN</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>452 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-INT18-a, CERT_C-INT18-b, CERT_C-INT18-c</td>
<td>Avoid possible integer overflow in expressions in which the result is assigned to a variable of a wider integer type. Avoid possible integer overflow in expressions in which the result is compared to an expression of a wider integer type. Avoid possible integer overflow in expressions in which the result is cast to a wider integer type.</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT_C: Rec. INT18-C</td>
<td>Checks for integer overflow and unsigned integer overflow (rec. fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>1890, 1891, 1892, 1893, 1894, 1895, 4490, 4491, 4492</td>
<td>Partially implemented</td>
</tr>
</tbody>
</table>

**Related Vulnerabilities**

Search for vulnerabilities resulting from the violation of this rule on the [CERT website](https://www.cert.org).

**Related Guidelines**

<table>
<thead>
<tr>
<th>Standard</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEI CERT C++ Coding Standard</td>
<td>INT35-CPP</td>
<td>Evaluate integer expressions in a larger size before comparing or assigning to that size</td>
</tr>
<tr>
<td>ISO/IEC TR 24772:2013</td>
<td>Numeric Conversion Errors [FLC]</td>
<td></td>
</tr>
<tr>
<td>MITRE CWE</td>
<td>CWE-681, CWE-190</td>
<td>Incorrect conversion between numeric types (CWE-681), Integer overflow (CWE-190)</td>
</tr>
</tbody>
</table>

**Bibliography**

- [Dowd 2006](#): Chapter 6, "C Language Issues"
- [Seacord 2013](#): Chapter 5, "Integer Security"