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

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 standard integer promotions are first performed on the operands, each of which has an 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. (See undefined behavior S1.)

Do not shift an expression by a negative number of bits or by a number greater than or equal to the precision of the promoted left operand. The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. For unsigned integer types, the width and the precision are the same; whereas for signed integer types, the width is one greater than the precision. This rule uses precision instead of width because, in almost every case, an attempt to shift by a number of bits greater than or equal to the precision of the operand indicates a bug (logic error). A logic error is different from overflow, in which there is simply a representational deficiency. In general, shifts should be performed only on unsigned operands. (See INT1 3-C. Use bitwise operators only on unsigned operands.)

Noncompliant Code Example (Left Shift, Unsigned Type)

The result of \( E_1 \ll E_2 \) is \( E_1 \) left-shifted \( E_2 \) bit positions; vacated bits are filled with zeros. The following diagram illustrates the left-shift operation.

```
 1 0 0 1 0 1 1 1

 0 0 1 0 1 1 1 0
```

According to the C Standard, if \( E_1 \) has an unsigned type, the value of the result is \( E_1 \times 2^{E_2} \), reduced modulo 1 more than the maximum value representable in the result type.

This noncompliant code example fails to ensure that the right operand is less than the precision of the promoted left operand:

```c
void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int uresult = ui_a << ui_b;
    /* ... */
}
```

Compliant Solution (Left Shift, Unsigned Type)

This compliant solution eliminates the possibility of shifting by greater than or equal to the number of bits that exist in the precision of the left operand:

```c
#include <limits.h>
#include <stdbool.h>
#include <inttypes.h>

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

void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int uresult = 0;
    if (ui_b >= PRECISION(UINT_MAX)) {
        /* Handle error */
    } else {
        uresult = ui_a << ui_b;
    }
    /* ... */
}
```
The `PRECISION()` macro and `popcount()` function provide the correct precision for any integer type. (See INT35-C. Use correct integer precisions.)

Modulo behavior resulting from left-shifting an unsigned integer type is permitted by exception INT30-EX3 to INT30-C. Ensure that unsigned integer operations do not wrap.

Noncompliant Code Example (Left Shift, Signed Type)

The result of \( E_1 \ll E_2 \) is \( E_1 \) left-shifted \( E_2 \) bit positions; vacated bits are filled with zeros. If \( E_1 \) has a signed type and nonnegative value, and \( E_1 \cdot 2^{E_2} \) is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

This noncompliant code example fails to ensure that left and right operands have nonnegative values and that the right operand is less than the precision of the promoted left operand. This example does check for signed integer overflow in compliance with INT32-C. Ensure that operations on signed integers do not result in overflow.

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

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

Shift operators and other bitwise operators should be used only with unsigned integer operands in accordance with INT13-C. Use bitwise operators only on unsigned operands.

Compliant Solution (Left Shift, Signed Type)

In addition to the check for overflow, this compliant solution ensures that both the left and right operands have nonnegative values and that the right operand is less than the precision of the promoted left operand:

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

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

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;
    }
    /* ... */
}
```

Noncompliant Code Example (Right Shift)

The result of \( E_1 \gg E_2 \) is \( E_1 \) right-shifted \( E_2 \) bit positions. If \( E_1 \) has an unsigned type or if \( E_1 \) has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of \( E_1 / 2^{E_2} \). If \( E_1 \) has a signed type and a negative value, the resulting value is implementation-defined and can be either an arithmetic (signed) shift.
or a logical (unsigned) shift

This noncompliant code example fails to test whether the right operand is greater than or equal to the precision of the promoted left operand, allowing undefined behavior:

```c
void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int uresult = ui_a >> ui_b;
    /* ... */
}
```

When working with signed operands, 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-C. Use bitwise operators only on unsigned operands.)

**Compliant Solution (Right Shift)**

This compliant solution eliminates the possibility of shifting by greater than or equal to the number of bits that exist in the precision of the left operand:

```c
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>
extern size_t popcount(uintmax_t);
#define PRECISION(x) popcount(x)

void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int uresult = 0;
    if (ui_b >= PRECISION(UINT_MAX)) {
        /* Handle error */
    } else {
        uresult = ui_a >> ui_b;
    }
    /* ... */
}
```

**Implementation Details**
GCC has no options to handle shifts by negative amounts or by amounts outside the width of the type predictably or to trap on them; they are always treated as undefined. Processors may reduce the shift amount modulo the width of the type. For example, 32-bit right shifts are implemented using the following instruction on x86-32:

```
sarl %cl, %eax
```

The `sarl` instruction takes a bit mask of the least significant 5 bits from `%cl` to produce a value in the range \([0, 31]\) and then shift `%eax` that many bits:

```
// 64-bit right shifts on IA-32 platforms become
shrdl %edx, %eax
sarl %cl, %edx
```

where `%eax` stores the least significant bits in the doubleword to be shifted, and `%edx` stores the most significant bits.

**Risk Assessment**

Although shifting a negative number of bits or shifting a number of bits greater than or equal to the width of the promoted left operand is undefined behavior in C, the risk is generally low because processors frequently reduce the shift amount modulo the width of the type.

<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>INT34-C</td>
<td>Low</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P2</td>
<td>L3</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>precision-shift-width</td>
<td>Fully checked</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precision-shift-width-constant</td>
<td></td>
</tr>
<tr>
<td>Axivion</td>
<td>7.2.0</td>
<td>CertC-INT34</td>
<td>Can detect shifts by a negative or an excessive number of bits and right shifts on negative values.</td>
</tr>
<tr>
<td>Bauhaus Suite</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeSonar</td>
<td>7.1p0</td>
<td>LANG.ARITH.BIGSHIFT</td>
<td>Shift amount exceeds bit width</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LANG.ARITH.NEGSHIFT</td>
<td>Negative shift amount</td>
</tr>
<tr>
<td>Compass</td>
<td>2017.07</td>
<td>BAD_SHIFT</td>
<td>Can detect violations of this rule. Unsigned operands are detected when checking for INT13-C. Use bitwise operators only on unsigned operands</td>
</tr>
<tr>
<td>/ROSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>BAD_SHIFT</td>
<td>Implemented</td>
</tr>
<tr>
<td>Cppcheck</td>
<td>1.66</td>
<td>shiftNegative, shiftTooManyBits</td>
<td>Context sensitive analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Warns whenever Cppcheck sees a negative shift for a POD expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(The warning for shifting too many bits is written only if Cppcheck has sufficient type information and you use --platform to specify the sizes of the standard types.)</td>
</tr>
<tr>
<td>ECLAIR</td>
<td>1.2</td>
<td>CC2.INT34</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2022.2</td>
<td>C0499, C2790, C2791, C2792, C2793</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C++2791, C++2792, C++2793, C++3003, C++3321, C++3322</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2022.2</td>
<td>MISRA.SHIFT.RANGE.2012</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>51 S, 403 S, 479 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C</td>
<td>2022.1</td>
<td>CERT_C-INT34-a</td>
<td>Avoid incorrect shift operations</td>
</tr>
<tr>
<td>/C++test</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2022b</td>
<td>CERT C: Rule INT34-C</td>
<td>Checks for:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Shift of a negative value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Shift operation overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rule partially covered.</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>0499, 2790 [C], 2791 [D], 2792 [A], 2793 [S]</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>2791, 2792, 2793, 3003, 3321, 3322</td>
<td></td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C</td>
<td>INT13-C. Use bitwise operators only on unsigned operands</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C</td>
<td>INT35-C. Use correct integer precisions</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C</td>
<td>INT32-C. Ensure that operations on signed integers do not result in overflow</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-682</td>
<td>2017-07-07: CERT: Rule subset of CWE</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-758</td>
<td>2017-07-07: CERT: Rule subset of CWE</td>
</tr>
</tbody>
</table>

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-758 and INT34-C


CWE-758 = Union( INT34-C, list) where list =

- Undefined behavior that results from anything other than incorrect bit shifting

CWE-682 and INT34-C

Independent( INT34-C, FLP32-C, INT33-C) CWE-682 = Union( INT34-C, list) where list =

- Incorrect calculations that do not involve out-of-range bit shifts

Bibliography

[C99 Rationale 2003] 6.5.7, "Bitwise Shift Operators"
[Dowd 2006] Chapter 6, "C Language Issues"
[Seacord 2013b] Chapter 5, "Integer Security"
[Viega 2005] Section 5.2.7, "Integer Overflow"