INT10-C. Do not assume a positive remainder when using the % operator

In C89 (and historical K&R implementations), the meaning of the remainder operator for negative operands was implementation-defined. This behavior was changed in C99, and the change remains in C11.

Because not all C compilers are strictly C-conforming, programmers cannot rely on the behavior of the % operator if they need to run on a wide range of platforms with many different compilers.

The C Standard, subclause 6.5.5 [ISO/IEC 9899:2011], states:

> *The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.*

and

> *When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded. If the quotient a/b is representable, the expression (a/b)*b + a%b shall equal a.*

Discarding the fractional part of the remainder is often called *truncation toward zero*.

The C definition of the % operator implies the following behavior:

```
 17 %  3  ->  2
 17 % -3  ->  2
-17 %  3  -> -2
-17 % -3  -> -2
```

The result has the same sign as the dividend (the first operand in the expression).

Noncompliant Code Example

In this noncompliant code example, the `insert()` function adds values to a buffer in a modulo fashion, that is, by inserting values at the beginning of the buffer once the end is reached. However, both `size` and `index` are declared as `int` and consequently are not guaranteed to be positive. Depending on the implementation and on the sign of `size` and `index`, the result of `(index + 1) % size` may be negative, resulting in a write outside the bounds of the `list` array.

```
int insert(int index, int *list, int size, int value) {
  if (size != 0) {
    index = (index + 1) % size;
    list[index] = value;
    return index;
  }
  else {
    return -1;
  }
}
```

This code also violates ERR02-C. Avoid in-band error indicators.

Noncompliant Code Example

Taking the absolute value of the modulo operation returns a positive value:

```
int insert(int index, int *list, int size, int value) {
  if (size != 0) {
    index = abs((index + 1) % size);
    list[index] = value;
    return index;
  }
  else {
    return -1;
  }
}
```

However, this noncompliant code example violates INT01-C. Use rsize_t or size_t for all integer values representing the size of an object. There is also a possibility that `(index + 1)` could result in a signed integer overflow in violation of INT32-C. Ensure that operations on signed integers do not result in overflow.

Compliant Solution (Unsigned Types)

The most appropriate solution in this case is to use unsigned types to eliminate any possible implementation-defined behavior, as in this compliant solution. For compliance with ERR02-C. Avoid in-band error indicators, this solution fills a result argument with the mathematical result and returns nonzero only if the operation succeeds.

```
int insert(size_t* result, size_t index, int *list, size_t size, int value) {
  if (size != 0 && size != SIZE_MAX) {
    index = (index + 1) % size;
    list[index] = value;
    *result = index;
    return 1;
  }
  else {
    return 0;
  }
}
```

Risk Assessment

Incorrectly assuming that the result of the remainder operator for signed operands will always be positive can lead to an out-of-bounds memory accessor other flawed logic.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT10-C	High	Unlikely	High	P3	L3

Automated Detection

Tool	Version	Checker	Description
Compass/ROSE			Could detect the specific noncompliant code example. It could identify when the result of a % operation might be negative and flag usage of that result in an array index. It could conceivably flag usage of any such result without first checking that the result is positive, but it would likely introduce many false positives
LDRA tool suite	9.7.1	**584 S**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_C-INT10-a**	Avoid accessing arrays out of bounds
Polyspace Bug Finder	R2019b	CERT C: Rec. INT10-C	Checks for tainted modulo operand (rec. fully covered)
PRQA QA-C	9.7	**3103**	Fully implemented

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	VOID INT10-CPP. Do not assume a positive remainder when using the % operator
CERT Oracle Secure Coding Standard for Java	NUM02-J. Ensure that division and remainder operations do not result in divide-by-zero errors
MITRE CWE	CWE-682, Incorrect calculation CWE-129, Unchecked array indexing

Bibliography

[Beebe 2005]	
[ISO/IEC 9899:2011]	Subclause 6.5.5, "Multiplicative Operators"
[Microsoft 2007]	C Multiplicative Operators
[Sun 2005]	Appendix E, "Implementation-Defined ISO/IEC C90 Behavior"