INT30-C. Ensure that unsigned integer operations do not wrap

The C Standard, 6.2.5, paragraph 9 [ISO/IEC 9899:2011], states

> *A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.*

This behavior is more informally called unsigned integer wrapping. Unsigned integer operations can wrap if the resulting value cannot be represented by the underlying representation of the integer. The following table indicates which operators can result in wrapping:

Operator	Wrap	Operator	Wrap	Operator	Wrap	Operator	Wrap
`+`	Yes	`-=`	Yes	`<<`	Yes	`<`	No
`-`	Yes	`*=`	Yes	`>>`	No	`>`	No
`*`	Yes	`/=`	No	`&`	No	`>=`	No
`/`	No	`%=`	No	`|`	No	`<=`	No
`%`	No	`<<=`	Yes	`^`	No	`==`	No
`++`	Yes	`>>=`	No	`~`	No	`!=`	No
`--`	Yes	`&=`	No	`!`	No	`&&`	No
`=`	No	`|=`	No	`un +`	No	`||`	No
`+=`	Yes	`^=`	No	`un -`	Yes	`?:`	No

The following sections examine specific operations that are susceptible to unsigned integer wrap. 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.)

Integer values must not be allowed to wrap, especially if they 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)
- In security-critical code

The C Standard defines arithmetic on atomic integer types as read-modify-write operations with the same representation as regular integer types. As a result, wrapping of atomic unsigned integers is identical to regular unsigned integers and 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 an unsigned integer wrap during the addition of the unsigned operands `ui_a` and `ui_b`. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable vulnerability.

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  /* ... */
}
```

Compliant Solution (Precondition Test)

This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap:

```
#include <limits.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum;
  if (UINT_MAX - ui_a < ui_b) {
    /* Handle error */
  } else {
    usum = ui_a + ui_b;
  }
  /* ... */
}
```

Compliant Solution (Postcondition Test)

This compliant solution performs a postcondition test to ensure that the result of the unsigned addition operation `usum` is not less than the first operand:

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  if (usum < ui_a) {
    /* Handle error */
  }
  /* ... */
}
```

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 an unsigned integer wrap during the subtraction of the unsigned operands `ui_a` and `ui_b`. If this behavior is unanticipated, it may lead to an exploitable vulnerability.

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff = ui_a - ui_b;
  /* ... */
}
```

Compliant Solution (Precondition Test)

This compliant solution performs a precondition test of the unsigned operands of the subtraction operation to guarantee there is no possibility of unsigned wrap:

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff;
  if (ui_a < ui_b){
    /* Handle error */
  } else {
    udiff = ui_a - ui_b;
  }
  /* ... */
}
```

Compliant Solution (Postcondition Test)

This compliant solution performs a postcondition test that the result of the unsigned subtraction operation `udiff` is not greater than the minuend:

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff = ui_a - ui_b;
  if (udiff > ui_a) {
    /* Handle error */
  }
  /* ... */
}
```

Multiplication

Multiplication is between two operands of arithmetic type.

Noncompliant Code Example

The Mozilla Foundation Security Advisory 2007-01 describes a heap buffer overflow vulnerability in the Mozilla Scalable Vector Graphics (SVG) viewer resulting from an unsigned integer wrap during the multiplication of the `signed int` value `pen->num_vertices` and the `size_t` value `sizeof(cairo_pen_vertex_t)` [VU#551436]. The `signed int` operand is converted to `size_t` prior to the multiplication operation so that the multiplication takes place between two `size_t` integers, which are unsigned. (See INT02-C. Understand integer conversion rules.)

```
pen->num_vertices = _cairo_pen_vertices_needed(
  gstate->tolerance, radius, &gstate->ctm
);
pen->vertices = malloc(
  pen->num_vertices * sizeof(cairo_pen_vertex_t)
);
```

The unsigned integer wrap can result in allocating memory of insufficient size.

Compliant Solution

This compliant solution tests the operands of the multiplication to guarantee that there is no unsigned integer wrap:

```
pen->num_vertices = _cairo_pen_vertices_needed(
  gstate->tolerance, radius, &gstate->ctm
);

if (pen->num_vertices > SIZE_MAX / sizeof(cairo_pen_vertex_t)) {
  /* Handle error */
}
pen->vertices = malloc(
  pen->num_vertices * sizeof(cairo_pen_vertex_t)
);
```

Exceptions

INT30-C-EX1: Unsigned integers can exhibit modulo behavior (wrapping) when necessary for the proper execution of the program. It is recommended that the variable declaration be clearly commented as supporting modulo behavior and that each operation on that integer also be clearly commented as supporting modulo behavior.

INT30-C-EX2: Checks for wraparound can be omitted when it can be determined at compile time that wraparound will not occur. As such, the following operations on unsigned integers require no validation:

- Operations on two compile-time constants
- Operations on a variable and 0 (except division or remainder by 0)
- Subtracting any variable from its type's maximum; for example, any `unsigned int` may safely be subtracted from `UINT_MAX`
- Multiplying any variable by 1
- Division or remainder, as long as the divisor is nonzero
- Right-shifting any type maximum by any number no larger than the type precision; for example, `UINT_MAX >> x` is valid as long as `0 <= x < 32` (assuming that the precision of `unsigned int` is 32 bits)

INT30-C-EX3. The left-shift operator takes two operands of integer type. Unsigned left shift << can exhibit modulo behavior (wrapping). This exception is provided because of common usage, because this behavior is usually expected by the programmer, and because the behavior is well defined. For examples of usage of the left-shift operator, see 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.

Risk Assessment

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT30-C	High	Likely	High	**P9**	**L2**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**integer-overflow**	Fully checked
CodeSonar	5.1p0	**ALLOC.SIZE.ADDOFLOW** **ALLOC.SIZE.IOFLOW** **ALLOC.SIZE.MULOFLOW** **ALLOC.SIZE.SUBUFLOW** **MISC.MEM.SIZE.ADDOFLOW** **MISC.MEM.SIZE.BAD** **MISC.MEM.SIZE.MULOFLOW** **MISC.MEM.SIZE.SUBUFLOW**	Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size
Compass/ROSE			Can detect violations of this rule by ensuring that operations are checked for overflow before being performed (Be mindful of exception INT30-EX2 because it excuses many operations from requiring validation, including all the operations that would validate a potentially dangerous operation. For instance, adding two `unsigned int`s together requires validation involving subtracting one of the numbers from `UINT_MAX`, which itself requires no validation because it cannot wrap.)
Coverity	2017.07	**INTEGER_OVERFLOW**	Implemented
Klocwork	2018	**NUM.OVERFLOW** **CWARN.NOEFFECT.OUTOFRANGE**	
LDRA tool suite	9.7.1	**493 S, 494 S**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_C-INT30-a** **CERT_C-INT30-b** **CERT_C-INT30-c**	Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '*' operator Integer overflow or underflow in constant expression in '<<' operator

Polyspace Bug Finder	R2019b	CERT C: Rule INT30-C	Checks for: • Unsigned integer overflow • Unsigned integer constant overflow Rule fully covered.
PRQA QA-C	9.7	**2910 [C], 2911 [D], 2912 [A], 2913 [S], 3383, 3384, 3385, 3386**	Partially implemented
PRQA QA-C++	4.4	**2910, 2911, 2912, 2913**	
PVS-Studio	6.23	**V658**	
TrustInSoft Analyzer	1.38	**unsigned overflow**	Exhaustively verified.

Related Vulnerabilities

CVE-2009-1385 results from a violation of this rule. The value performs an unchecked subtraction on the `length` of a buffer and then adds those many bytes of data to another buffer [xorl 2009]. This can cause a buffer overflow, which allows an attacker to execute arbitrary code.

A Linux Kernel vmsplice exploit, described by Rafal Wojtczuk [Wojtczuk 2008], documents a vulnerability and exploit arising from a buffer overflow (caused by unsigned integer wrapping).

Don Bailey [Bailey 2014] describes an unsigned integer wrap vulnerability in the LZO compression algorithm, which can be exploited in some implementations.

CVE-2014-4377 describes a vulnerability in iOS 7.1 resulting from a multiplication operation that wraps, producing an insufficiently small value to pass to a memory allocation routine, which is subsequently overflowed.

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	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	CON08-C. Do not assume that a group of calls to independently atomic methods is atomic	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
CWE 2.11	CWE-190, Integer Overflow or Wraparound	2016-12-02: CERT: Rule subset of CWE
CWE 2.11	CWE-131	2017-05-16: 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-131 and INT30-C

- Intersection(INT30-C, MEM35-C) = Ø

- Intersection(CWE-131, INT30-C) =

- Calculating a buffer size such that the calculation wraps. This can happen, for example, when using malloc() or operator new[] to allocate an array, multiplying the array item size with the array dimension. An untrusted dimension could cause wrapping, resulting in a too-small buffer being allocated, and subsequently overflowed when the array is initialized.

- CWE-131 – INT30-C =

- Incorrect calculation of a buffer size that does not involve wrapping. This includes off-by-one errors, for example.

INT30-C – CWE-131 =

- Integer wrapping where the result is not used to allocate memory.

CWE-680 and INT30-C

Intersection(CWE-680, INT30-C) =

- Unsigned integer overflows that lead to buffer overflows

CWE-680 - INT30-C =

- Signed integer overflows that lead to buffer overflows

INT30-C – CWE-680 =

- Unsigned integer overflows that do not lead to buffer overflows

CWE-191 and INT30-C

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

Intersection(CWE-191, INT30-C) =

- Underflow of unsigned integer operation

CWE-191 – INT30-C =

- Underflow of signed integer operation

INT30-C – CWE-191 =

- Overflow of unsigned integer operation

Bibliography

[Bailey 2014]	Raising Lazarus - The 20 Year Old Bug that Went to Mars
[Dowd 2006]	Chapter 6, "C Language Issues" ("Arithmetic Boundary Conditions," pp. 211–223)
[ISO/IEC 9899:2011]	Subclause 6.2.5, "Types"
[Seacord 2013b]	Chapter 5, "Integer Security"
[Viega 2005]	Section 5.2.7, "Integer Overflow"
[VU#551436]	
[Warren 2002]	Chapter 2, "Basics"
[Wojtczuk 2008]	
[xorl 2009]	"CVE-2009-1385: Linux Kernel E1000 Integer Underflow"