INT02-C. Understand integer conversion rules

Conversions can occur explicitly as the result of a cast or implicitly as required by an operation. Although conversions are generally required for the correct execution of a program, they can also lead to lost or misinterpreted data. Conversion of an operand value to a compatible type causes no change to the value or the representation.

The C integer conversion rules define how C compilers handle conversions. These rules include *integer promotions*, *integer conversion rank*, and the *usual arithmetic conversions*. The intent of the rules is to ensure that the conversions result in the same numerical values and that these values minimize surprises in the rest of the computation. Prestandard C usually preferred to preserve signedness of the type.

Integer Promotions

Integer types smaller than `int` are promoted when an operation is performed on them. If all values of the original type can be represented as an `int`, the value of the smaller type is converted to an `int`; otherwise, it is converted to an `unsigned int`. Integer promotions are applied as part of the usual arithmetic conversions to certain argument expressions; operands of the unary `+`, `-`, and `~` operators; and operands of the shift operators. The following code fragment shows the application of integer promotions:

```
char c1, c2;
c1 = c1 + c2;
```

Integer promotions require the promotion of each variable (`c1` and `c2`) to `int` size. The two `int` values are added, and the sum is truncated to fit into the `char` type. Integer promotions are performed to avoid arithmetic errors resulting from the overflow of intermediate values:

```
signed char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3;
```

In this example, the value of `c1` is multiplied by `c2`. The product of these values is then divided by the value of `c3` (according to operator precedence rules). Assuming that `signed char` is represented as an 8-bit value, the product of `c1` and `c2` (300) cannot be represented. Because of integer promotions, however, `c1`, `c2`, and `c3` are each converted to `int`, and the overall expression is successfully evaluated. The resulting value is truncated and stored in `cresult`. Because the final result (75) is in the range of the `signed char` type, the conversion from `int` back to `signed char` does not result in lost data.

Integer Conversion Rank

Every integer type has an integer conversion rank that determines how conversions are performed. The ranking is based on the concept that each integer type contains at least as many bits as the types ranked below it. The following rules for determining integer conversion rank are defined in the C Standard, subclause 6.3.1.1 [ISO/IEC 9899:2011]:

- No two signed integer types shall have the same rank, even if they have the same representation.
- The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.
- The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.
- The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.
- The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.
- The rank of `char` shall equal the rank of `signed char` and `unsigned char`.
- The rank of `_Bool` shall be less than the rank of all other standard integer types.
- The rank of any enumerated type shall equal the rank of the compatible integer type.
- The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined but still subject to the other rules for determining the integer conversion rank.
- For all integer types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

The integer conversion rank is used in the usual arithmetic conversions to determine what conversions need to take place to support an operation on mixed integer types.

Usual Arithmetic Conversions

The usual arithmetic conversions are rules that provide a mechanism to yield a common type when both operands of a binary operator are balanced to a common type or the second and third operands of the conditional operator (`? :`) are balanced to a common type. Conversions involve two operands of different types, and one or both operands may be converted. Many operators that accept arithmetic operands perform conversions using the usual arithmetic conversions. After integer promotions are performed on both operands, the following rules are applied to the promoted operands:

1. If both operands have the same type, no further conversion is needed.
2. If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
3. If the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.
4. If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.
5. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

Example

In the following example, assume the code is compiled using an implementation with 8-bit `char`, 32-bit `int`, and 64-bit `long long`:

```
signed char sc = SCHAR_MAX;
unsigned char uc = UCHAR_MAX;
signed long long sll = sc + uc;
```

Both the `signed char sc` and the `unsigned char uc` are subject to integer promotions in this example. Because all values of the original types can be represented as `int`, both values are automatically converted to `int` as part of the integer promotions. Further conversions are possible if the types of these variables are not equivalent as a result of the usual arithmetic conversions. The actual addition operation, in this case, takes place between the two 32-bit `int` values. This operation is not influenced by the resulting value being stored in a `signed long long` integer. The 32-bit value resulting from the addition is simply sign-extended to 64 bits after the addition operation has concluded.

Assuming that the precision of `signed char` is 7 bits, and the precision of `unsigned char` is 8 bits, this operation is perfectly safe. However, if the compiler represents the `signed char` and `unsigned char` types using 31- and 32-bit precision (respectively), the variable `uc` would need to be converted to `unsigned int` instead of `signed int`. As a result of the usual arithmetic conversions, the `signed int` is converted to unsigned, and the addition takes place between the two `unsigned int` values. Also, because `uc` is equal to `UCHAR_MAX`, which is equal to `UINT_MAX`, the addition results in an overflow in this example. The resulting value is then zero-extended to fit into the 64-bit storage allocated by `sll`.

Noncompliant Code Example (Comparison)

The programmer must be careful when performing operations on mixed types. This noncompliant code example shows an idiosyncrasy of integer promotions:

```
int si = -1;
unsigned int ui = 1;
printf("%d\n", si < ui);
```

In this example, the comparison operator operates on a `signed int` and an `unsigned int`. By the conversion rules, `si` is converted to an `unsigned int`. Because 1 cannot be represented as an `unsigned int` value, the 1 is converted to `UINT_MAX` in accordance with the C Standard, subclause 6.3.1.3, paragraph 2 [ISO/IEC 9899:2011]:

> *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.*

Consequently, the program prints 0 because `UINT_MAX` is not less than 1.

Compliant Solution

The noncompliant code example can be modified to produce the intuitive result by forcing the comparison to be performed using `signed int` values:

```
int si = -1;
unsigned ui = 1;
printf("%d\n", si < (int)ui);
```

This program prints 1 as expected. Note that `(int)ui` is correct in this case only because the value of `ui` is known to be representable as an `int`. If it were not known, the compliant solution would need to be written as

```
int si = /* Some signed value */;
unsigned ui = /* Some unsigned value */;
printf("%d\n", (si < 0 || (unsigned)si < ui));
```

Noncompliant Code Example

This noncompliant code example demonstrates how performing bitwise operations on integer types smaller than `int` may have unexpected results:

```
uint8_t port = 0x5a;
uint8_t result_8 = ( ~port ) >> 4;
```

In this example, a bitwise complement of `port` is first computed and then shifted 4 bits to the right. If both of these operations are performed on an 8-bit unsigned integer, then `result_8` will have the value `0x0a`. However, `port` is first promoted to a `signed int`, with the following results (on a typical architecture where type `int` is 32 bits wide):

Expression	Type	Value	Notes
`port`	`uint8_t`	`0x5a`	
`~port`	`int`	`0xffffffa5`	
`~port >> 4`	`int`	`0x0ffffffa`	Whether or not value is negative is implementation-defined
`result_8`	`uint8_t`	`0xfa`	

Compliant Solution

In this compliant solution, the bitwise complement of `port` is converted back to 8 bits. Consequently, `result_8` is assigned the expected value of `0x0aU`.

```
uint8_t port = 0x5a;
uint8_t result_8 = (uint8_t) (~port) >> 4;
```

Risk Assessment

Misunderstanding integer conversion rules can lead to errors, which in turn can lead to exploitable vulnerabilities. The major risks occur when narrowing the type (which requires a specific cast or assignment), converting from unsigned to signed, or converting from negative to unsigned.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT02-C	Medium	Probable	Medium	**P8**	**L2**

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.0p0	**ALLOC.SIZE.TRUNC** **LANG.CAST.COERCE** **LANG.CAST.VALUE** **MISC.MEM.SIZE.TRUNC**	Truncation of Allocation Size Coercion Alters Value Cast Alters Value Truncation of Size
ECLAIR	1.2	**CC2.INT02**	Fully implemented
Klocwork	2018	**MISRA.CAST.INT** **MISRA.CAST.UNSIGNED_BITS** **MISRA.CONV.INT.SIGN** **MISRA.CVALUE.IMPL.CAST** **MISRA.UMINUS.UNSIGNED** **PRECISION.LOSS**	
LDRA tool suite	9.7.1	**52 S, 93 S, 96 S, 101 S, 107 S, 332 S, 334 S, 433 S, 434 S, 446 S, 452 S, 457 S, 458 S**	Fully implemented
Parasoft C/C++test	10.4	**CERT_C-INT02-a** **CERT_C-INT02-b**	Implicit conversions from wider to narrower integral type which may result in a loss of information shall not be used Avoid mixing arithmetic of different precisions in the same expression
Polyspace Bug Finder	R2018a	Integer conversion overflow Integer overflow Tainted sign change conversion Unsigned integer conversion overflow MISRA C:2012 Rule 10.1 MISRA C:2012 Rule 10.3 MISRA C:2012 Rule 10.4 MISRA C:2012 Rule 10.6 MISRA C:2012 Rule 10.7 MISRA C:2012 Rule 10.8	Overflow when converting between integer types Overflow from operation between integers Value from an unsecure source changes sign Overflow when converting between unsigned integer types Operands shall not be of an inappropriate essential type An expression value shall not be assigned to an object with a narrower essential type or of a different category Both operands of an operator in which usual arithmetic conversions are performed shall have the same 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 The value of a composite expression shall not be cast to a different essential type category or a wider essential type

PRQA QA-C	9.3	**1256, 1257, 1266, 1290, 1291, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299, 4401, 4402, 4403, 4404, 4405, 4410, 4412, 4413, 4414, 4415, 4420, 4421, 4422, 4423, 4424, 4425, 4430, 4431, 4432, 4434, 4435, 4436, 4437, 4440, 4441, 4442, 4443, 4445, 4446, 4447, 4460, 4461, 4463, 4464, 4470, 4471, 4480, 4481, 1250, 1251, 1252, 1253, 1260, 1263, 1274, 1800, 1802, 1803, 1804, 1810, 1811, 1812, 1813, 1820, 1821, 1822, 1823, 1824, 1830, 1831, 1832, 1833, 1834, 1840, 1841, 1842, 1843, 1844, 1850, 1851, 1852, 1853, 1854, 1860, 1861, 1862, 1863, 1864, 1880, 1881, 1882, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2122, 2124, 2130, 2132, 2134**	Fully implemented
PVS-Studio	6.23	**V555**, **V605**, **V673**	

Related Vulnerabilities

This vulnerability in Adobe Flash arises because Flash passes a signed integer to `calloc()`. An attacker has control over this integer and can send negative numbers. Because `calloc()` takes `size_t`, which is unsigned, the negative number is converted to a very large number, which is generally too big to allocate, and as a result, `calloc()` returns `NULL`, causing the vulnerability to exist.

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

Related Guidelines

SEI CERT C++ Coding Standard	VOID INT02-CPP. Understand integer conversion rules
ISO/IEC TR 24772:2013	Numeric Conversion Errors [FLC]
MISRA C:2012	Rule 10.1 (required) Rule 10.3 (required) Rule 10.4 (required) Rule 10.6 (required) Rule 10.7 (required) Rule 10.8 (required)
MITRE CWE	CWE-192, Integer coercion error CWE-197, Numeric truncation error

Bibliography

[Dowd 2006]	Chapter 6, "C Language Issues" ("Type Conversions," pp. 223–270)
[Seacord 2013]	Chapter 5, "Integer Security"

← ↑ →