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:

```c
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:

```c
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 any of the 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:

```c
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:

```c
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:

```c
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

```c
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:
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):

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>port</code></td>
<td><code>uint8_t</code></td>
<td><code>0x5a</code></td>
<td></td>
</tr>
<tr>
<td><code>-port</code></td>
<td><code>int</code></td>
<td><code>0xfffffa5</code></td>
<td></td>
</tr>
<tr>
<td><code>-port &gt;&gt; 4</code></td>
<td><code>int</code></td>
<td><code>0x0fffffff</code></td>
<td>Whether or not value is negative is implementation-defined</td>
</tr>
<tr>
<td><code>result_8</code></td>
<td><code>uint8_t</code></td>
<td><code>0xfa</code></td>
<td></td>
</tr>
</tbody>
</table>

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`.

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

Noncompliant Code Example

This noncompliant code example, adapted from the Cryptography Services blog, demonstrates how signed overflow can occur even when it seems that only unsigned types are in use:

```c
unsigned short x = 45000, y = 50000;
unsigned int z = x * y;
```

On implementations where `short` is 16 bits wide and `int` is 32 bits wide, the program results in undefined behavior due to signed overflow. This is because the `unsigned short`s become signed when they are automatically promoted to integer, and their mathematical product (2250000000) is greater than the largest signed 32-bit integer (2^31 - 1, which is 2147483647).

Compliant Solution

In this compliant solution, by manually casting one of the operands to `unsigned int`, the multiplication will be unsigned and so will not result in undefined behavior:

```c
unsigned short x = 45000, y = 50000;
unsigned int z = x * (unsigned int)y;
```

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.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT02-C</td>
<td>Medium</td>
<td>Probable</td>
<td>Medium</td>
<td>P8</td>
<td>L2</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>Astére</td>
<td>20.10</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Tool</td>
<td>Version</td>
<td>Rules</td>
<td>Implementation</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------</td>
<td>-------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>6.2p0</td>
<td>ALLOC.SIZE.TRUNC, LANG.CAST.COERC, LANG.CAST.VALUE, MISC.MEM.SIZE.TRUNC</td>
<td>Truncation of Allocation Size, Coercion Alters Value, Cast Alters Value, Truncation of Size</td>
</tr>
<tr>
<td>ECLAIR</td>
<td>1.2</td>
<td>CC2.INT02</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2021.3</td>
<td>C1250, C1251, C1252, C1253, C1256, C1257, C1260, C1263, C1266, C1274, C1290, C1291, C1292, C1293, C1294, C1295, C1296, C1297, C1298, C1299, C1800, C1802, C1803, C1804, C1810, C1811, C1812, C1813, C1821, C1822, C1823, C1824, C1830, C1831, C1832, C1833, C1834, C1840, C1841, C1842, C1843, C1844, C1845, C1850, C1851, C1852, C1853, C1854, C1860, C1861, C1862, C1863, C1864, C1880, C1881, C1882, C2100, C2101, C2102, C2103, C2104, C2105, C2106, C2107, C2109, C2110, C2111, C2112, C2113, C2114, C2115, C2116, C2117, C2118, C2119, C2120, C2122, C2124, C2130, C2132, C2134, C4401, C4402, C4403, C4404, C4405, C4410, C4412, C4413, C4414, C4415, C4420, C4421, C4422, C4423, C4424, C4425, C4430, C4431, C4432, C4433, C4434, C4435, C4436, C4437, C4440, C4441, C4442, C4443, C4445, C4446, C4447, C4460, C4461, C4463, C4464, C4470, C4471, C4480, C4481</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2021.4</td>
<td>MISRA.CAST.INT, MISRA.CAST.UNSIGNED_BITS, MISRA.CONV.INT.SIGN, MISRA.CVALUE.IMPL.CAST, MISRA.UMINUS.UNSIGNED, PRECISION LOSS</td>
<td></td>
</tr>
<tr>
<td>Parasoft C /C++ test</td>
<td>2021.2</td>
<td>CERT_C-INT02-a, CERT_C-INT02-b</td>
<td>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</td>
</tr>
<tr>
<td>PC-int Plus</td>
<td>1.4</td>
<td>501, 502, 569, 570, 573, 574, 701, 702, 732, 734, 737</td>
<td>Partially supported</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2021a</td>
<td>CERT C: Rec. INT02-C</td>
<td>Checks for sign change integer conversion overflow (rec. fully supported)</td>
</tr>
<tr>
<td>PROQA QA-C</td>
<td>9.7</td>
<td>1250, 1251, 1252, 1253, 1256, 1257, 1260, 1263, 1266, 1274, 1290, 1291, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299, 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, 1868, 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, 4401, 4402, 4403, 4404, 4405, 4410, 4412, 4413, 4414, 4415, 4420, 4421, 4422, 4423, 4424, 4425, 4430, 4431, 4432, 4433, 4434, 4435, 4436, 4440, 4441, 4442, 4443, 4445, 4446, 4447, 4448, 4450, 4451, 4452, 4453, 4454, 4455, 4456, 4460, 4461, 4463, 4464, 4470, 4471, 4480, 4481</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.16</td>
<td>V555, V605, V673, V5006</td>
<td></td>
</tr>
</tbody>
</table>

**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](https://www.cert.org).
### Bibliography

<table>
<thead>
<tr>
<th>Reference</th>
<th>Citation</th>
</tr>
</thead>
</table>