



Foundations of Cybersecurity

C and C++ Secure Coding

Gianluca Dini
Dept. of Information Engineering
University of Pisa

Email: gianluca.dini@unipi.it

Version: 2021-03-03

1

Credits



- These slides come from a version originally produced by Dr. Pericle Perazzo

C and C++ Secure Coding

INTEGERS

3

Signed/Unsigned Integer Mismatch



```
int table[100];  
void func(int index) {  
    if (index >= 100) {  
        return;  
    }  
    table[index] = 25;  
}
```

4

This function writes a value 25 into the element of “table” of index “index”. However, it fails to check for negative values of the “index” parameter. An attacker who controls the “index” argument can cause the program to write a value in an arbitrary location in memory.

Signed/Unsigned Integer Mismatch

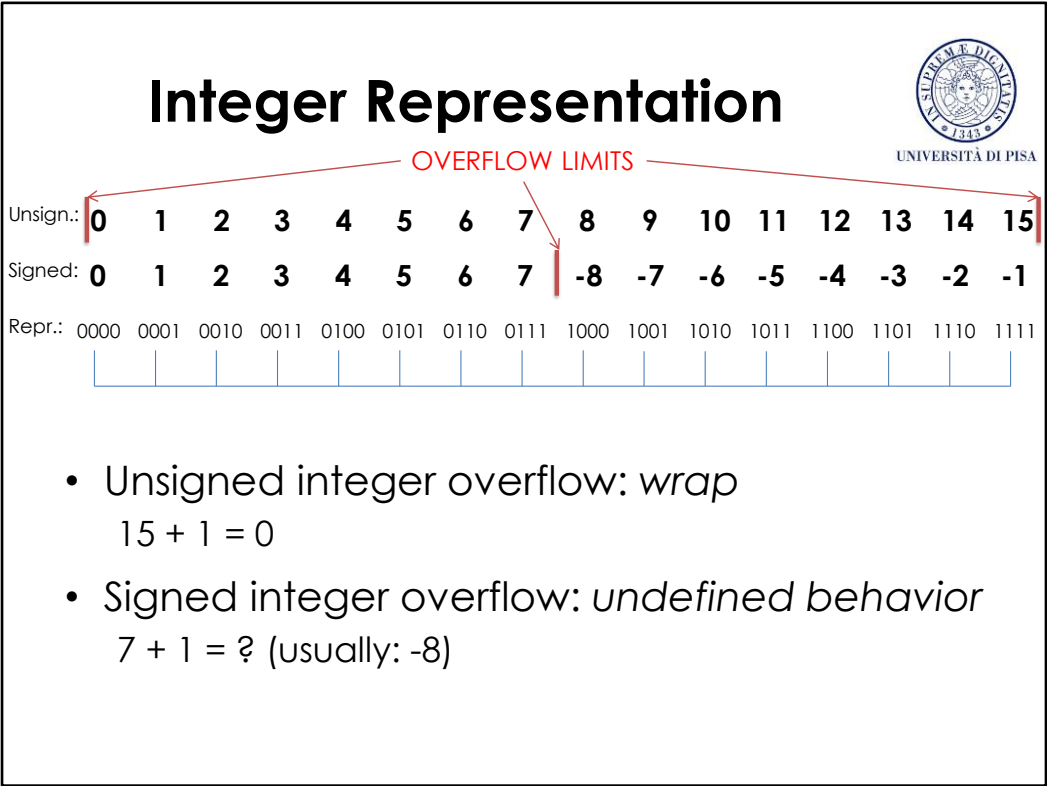


```
int table[100];  
void func(size_t index) {  
    if (index >= 100) {  
        return;  
    }  
    table[index] = 25;  
}
```

- Don't use *signed* ints when not needed

5

Using `size_t` (or any unsigned integer type) to size of index an array is always preferable, because less error-prone. It is very common in C/C++ to use signed integers everywhere, also in those cases (the majority) in which the signedness is useless, for example when sizing or indexing arrays. The default integer type (`int`) is signed only for historical reasons: plain-old C programs did not have exceptions, so they used signed integers to permit `int`-returning functions to return the special value `-1` in case of errors.



The C standard imposes the unsigned integers to be represented with their binary representation, while allows the signed integers to be represented with three techniques: sign-and-magnitude, one's complement, two's complement. The most used representation in desktop systems is however two's complement. This figure shows the representation of a 4-bit unsigned integer, and of a 4-bit signed integer in two's complement.

If an unsigned integer has the max value (in this example, 15) and gets incremented, then it will assume the 0 value (integer wrap around). Similarly, if an unsigned integer has the min value (0) and gets decrements, then it will assume the 15 value (in this case). The C unsigned integers implement thus a modular arithmetic, and not the whole natural(+0) arithmetic.

Conversely, if a signed integer has the max value (7 in this case) and gets increments, then an undefined behavior happens (integer overflow). Similarly, if a signed integer has the min value (-8 in this case) and gets decrements, then an undefined behavior happens. The unsigned wrap around can be thus a wanted behavior, for example when the programmer want to perform modular operations, whereas the signed overflow is always a bad programming practice. Depending on the underlying implementation, an integer overflow could cause various behaviors. The majority of platforms use two's complement representation, and silently wrap in case of signed integer overflow. Many programmers rely on this usual behavior as if it were the standard one, but it is not. Notably, compilers leverage the assumption that no integer overflow happens in order to implement code optimization. So the actual behavior could vary on the compiler, the compiler version, and the single piece of code.

Despite the fact that the integers are used primarily for sizing and indexing arrays, for which the presence of the sign is meaningless, the signed integers are traditionally the most used integer types in C and in many modern programming languages. Think about, for example, the «classical» instruction: `for(int i=0; i<size; i++) { /* Some code */ }`. Historically, this is due because old standard library functions returned an integer to represent the number of bytes written or read or copied, and used negative values to represent errors.

Unsigned Wrap




```
void func(unsigned int a, unsigned int b) {  
    char* v = (char*)malloc(a + b);  
    if(!v) { /* Handle error */ }  
    /* Write on v */  
}
```




```
void func(unsigned int a, unsigned int b) {  
    if(a + b > 1024) { /* Handle error */ }  
    char* v = (char*)malloc(a + b);  
    if(!v) { /* Handle error */ }  
    /* Write on v */  
}
```

This function allocates an array of characters sized as the sum of two unsigned integers "a" and "b", and then writes some data in it.


However, the sum of two unsigned integers could wrap, resulting in a small integer. In this case, the malloc() function does not fail (thus the «if(!v)» check passes), but it allocates less memory than wanted. After that, the program believes to have a large allocated memory, whereas it has little. This situation easily leads to buffer overflows, as subsequent operations may write on unallocated memory locations. Note that checking if a + b goes beyond a given threshold (1024 in the slide) is not an acceptable sanitization, since such an operation may wrap again. In other words, we only moved the problem, not solved it.



Unsigned Wrap



```
void func(unsigned int a, unsigned int b) {  
    if(a>256 || b>768) { /* Handle error */ }  
    char* v = (char*)malloc(a + b);  
    if(!v) { /* Handle error */ }  
    /* Write on v */  
}
```



```
void func(unsigned int a, unsigned int b) {  
    if(a > UINT_MAX-b) { /* Handle error */ }  
    char* v = (char*)malloc(a + b);  
    if(!v) { /* Handle error */ }  
    /* Write on v */  
}
```

only if I can restrict the domain of a and b

When possible, checking both operands (a and b) to stay within some thresholds (respectively, 256 and 768 in the slide) is an acceptable sanitization. However, sometimes it is not possible to define a threshold for the single operands. A more general sanitization is based on the «UINT_MAX» limit as shown in the slide. «UINT_MAX» represents the maximum possible unsigned integer, and its value can change depending on the compiler and the target architecture (x86, x64). Note that «UINT_MAX-b» can never wrap.

Integer Limits



- **Unsigned:**

– unsigned char	[0, UCHAR_MAX]
– unsigned short	[0, USHRT_MAX]
– unsigned int	[0, UINT_MAX]
– unsigned long	[0, ULONG_MAX]
– unsigned long long	[0, ULLONG_MAX]
– size_t	[0, SIZE_MAX]
- **Signed:**

– signed char	[SCHAR_MIN, SCHAR_MAX]
– short	[SHRT_MIN, SHRT_MAX]
– int	[INT_MIN, INT_MAX]
– long	[LONG_MIN, LONG_MAX]
– long long	[LLONG_MIN, LLONG_MAX]

Standard libraries (typically `<limits.h>`) define macros for the maximum and the minimum values of all the common integer types. The most important ones are: `UINT_MAX` (the max value of an unsigned int), `INT_MAX`, `INT_MIN` (respectively the max and the min value of a signed int), and `SIZE_MAX` (the max value of a `size_t`). These values depend on the platform, because the C standard poses no constraint on how many bits an "int" or an "unsigned int" are represented. They should always be used to sanitize tainted integers in order to avoid unwanted overflows.



Unsigned Wrap



```
void func(unsigned int a) {  
    int* v = (int*)malloc(a*sizeof(int));  
    if(!v) { /* Handle error */ }  
    /* Write on v */  
}
```



```
void func(unsigned int a) {  
    if (a > 1024) { /* Handle error */ }  
    int* v = (int*)malloc(a*sizeof(int));  
    if(!v) { /* Handle error */ }  
    /* Write on v */  
}
```

only if I can
restrict the
domain of a

This function allocates an array of integers sized as an unsigned integers "a", and then writes some data in it.

However, the product between «a» and sizeof(int) could wrap, leading to an insufficient memory allocation. Subsequent operations may then write on unallocated memory locations. When possible, the simplest way to solve the problem is to check "a" not to exceed a given threshold.

Unsigned Wrap



```
void func(unsigned int a) {  
    if (a > SIZE_MAX/sizeof(int)) {  
        /* Handle error */  
    }  
    int* v = (int*)malloc(a*sizeof(int));  
    if(!v) { /* Handle error */ }  
    /* Write on v */  
}
```

A more general sanitization is based on the «SIZE_MAX» limit as shown in the slide. «SIZE_MAX» represents the maximum possible «size_t», which is the unsigned integer type returned by the sizeof() operator.

General Sanitization Method



1. Write overflow condition «as is»

$a + b > \text{UINT_MAX}$

It may overflow, too!

2. Make it «safe» with an algebrical passage

$a > \text{UINT_MAX} - b$

3. Avoid additional overflows

($\text{UINT_MAX} - b$ cannot overflow)

RESULT: `if (a > UINT_MAX - b){ /* Handle error */ }`

A general method to sanitize integers before potentially overflowing operations is the following:

- 1) Identify how an arithmetic operation could overflow, and write the error condition «as is», i.e., as if no overflow would happen within the error condition itself. In the sum example: « $a + b > \text{UINT_MAX}$ ».
- 2) Algebrically change the condition to avoid overflow inside the error condition itself. In the sum example: « $a > \text{UINT_MAX} - b$ ».
- 3) Possibly avoid the additional overflows in the error condition just obtained. In the sum example, this step leaves the error condition unchanged since $\text{UINT_MAX} - b$ cannot overflow.

The final sanitization condition will be: «`if(a>UINT_MAX - b){ /* Handle error */ }`».

General Sanitization Method




1. Write overflow condition «as is»
 $a * b > \text{UINT_MAX}$
2. Make it «safe» with an algebrical passage
 $b \neq 0 \ \&\& \ a > \text{UINT_MAX}/b$
(ignore $b == 0$ because $a * b$ cannot overflow)
3. Avoid additional overflows
($\text{UINT_MAX}/b$ cannot overflow)

RESULT: `if (b != 0 && a > UINT_MAX/b){ /* Handle error */ }`

It is possible to apply the method for the case of unsigned integer multiplication. This time, in step 2 the possible division by zero must be addressed. So we must add a precondition « $b \neq 0$ ». We can ignore the opposite case $b == 0$, since in this case $a * b$ cannot overflow. The final sanitization condition will be: «if($b \neq 0 \ \&\& \ a > \text{UINT_MAX}/b$){ /* Handle error */ }».

Unsigned General Sanitizations



- Sum:

```
if (a > UINT_MAX - b){ ... }
result = a + b;
```

* better version exists (see after)
- Multiplication:

```
if (b != 0 && a > UINT_MAX/b){ ... }
result = a*b;
```
- Subtraction:

```
if (a < b){ ... }
result = a - b;
```
- Division:
(no wrap, but check for division by zero)

```
if (b == 0){ ... }
result = a/b;
```
- Increment:

```
if (a == UINT_MAX){ ... }
a++;
```


* better version exists (see after)
- Modulo:
(idem)

```
if (b == 0){ ... }
result = a%b;
```
- Decrement:

```
if (a == 0){ ... }
a--;
```

This slide shows the sanitization condition for all the unsigned integer operations. Note that the unsigned division (/) and remainder (%) can never wrap. Still, the case in which the divisor is 0 must be checked in both operations. Indeed, the division by zero produces an undefined behavior, and some implementations could not raise an exception but silently produce an unexpected result. All the above sanitization tests are *precondition tests*, because they test the overflow condition *before* performing the actual operation.

Overflow-Tolerant Sanitizations



UNIVERSITÀ DI PISA

- Sum:

POSTCONDITION:

```
result = a + b;  
if (result < a){ ... }
```

PRECONDITION:

```
if (a + b < a){ ... }  
result = a + b;
```
- Increment:

POSTCONDITION:

```
a++;  
if (a == 0){ ... }
```

PRECONDITION:

```
if (a + 1 == 0){ ... }  
a++;
```
- Independent from the bitsize (unsigned int, unsigned short, etc.) → Less error-prone!
- Do not use with *signed* integers!

15

In case of sum or increment of unsigned integers, it is possible to apply *postcondition tests*, which test the overflow condition *after* having performed the operation. Postcondition tests allow the overflow to happen, and they check it afterwards, leveraging the fact that unsigned overflow is a well-defined behavior. An equivalent precondition test is also possible (see the example snippets on the right of the slide), by letting the overflow happen inside the check itself. The advantage of the tests shown in this slide is that they are independent from the bitsize of the integers (unsigned int, unsigned short, etc.) and they do not use the `<limits.h>` macros (`UINT_MAX`, `USHRT_MAX`, etc.), so they are less prone to errors. Of course, postcondition sanitizations cannot be used for signed integers, because signed overflow leads to undefined behavior.

Signed Integer Overflow



```
void func(int a, int b) {  
    int result = a + b;  
    /* ... */  
}
```

- Two overflow conditions!
 - Beyond INT_MAX
 - Below INT_MIN
- Countermeasures:
 - Sanitize a and b to be ≥ 0
 - Tell compiler to raise exception on overflow (`-ftrapv` in gcc, `/RTCc` in Visual Studio)
 - Apply general sanitization method *twice* (overflow and underflow)

Sanitizing operations with signed integers is in general more difficult than unsigned integers. This is because signed operations typically can overflow in both directions: beyond INT_MAX and below INT_MIN. For example, a signed sum can overflow beyond INT_MAX if the operands are both positive, or below INT_MIN if they are both negative.

The simplest countermeasure is to sanitize the signed integers to be non-negative, thus making them unsigned. Then, apply the sanitization rules for the unsigned ints seen before. However, sanitizing away the negative values is not always possible, as they may be valid values of the program. Another solution is to tell the compiler to raise an exception on integer overflows, so that the program crashes instead of going in undefined behavior. GNU GCC compiler allows to do that with the `-ftrapv` flag. The `-ftrapv` flag forces the integer overflow to raise an exception, which by default results in a program termination. This makes a program more secure, but also less efficient because it causes implicit function calls in every signed operation and prevents some hardware optimizations. Moreover, it does not cover all the signed operations, so it is a partial solution. Another solution is to apply the general sanitization method twice, once for each overflow direction.


Signed Integer Overflow




- Overflow beyond INT_MAX:
 1. Write overflow condition «as is»
 $a + b > \text{INT_MAX}$
 2. Make it «safe» with an algebrical passage
 $a > \text{INT_MAX} - b \rightarrow$ it may overflow again if $b < 0$!
 3. Avoid additional overflows
 $b \geq 0 \ \&\& \ a > \text{INT_MAX} - b$
(if $b < 0$, $a + b$ can't overflow beyond INT_MAX \rightarrow ok)
- Overflow below INT_MIN (underflow):
 1. $a + b < \text{INT_MIN}$
 2. $a < \text{INT_MIN} - b \rightarrow$ it may overflow again if $b > 0$!
 3. $b \leq 0 \ \&\& \ a < \text{INT_MIN} - b$
(if $b > 0$, $a + b$ can't overflow below INT_MIN \rightarrow ok)

This time, the resulting sanitization check after step 2 (« $a > \text{INT_MAX} - b$ ») can overflow if $b < 0$, so we have to (manually) avoid this additional overflow in step 3. This is usually done by adding a precondition to the sanitization check to avoid overflows in it. In this case, we can add the precondition « $b \geq 0$ », thus obtaining the sanitization check « $b \geq 0 \ \&\& \ a > \text{INT_MAX} - b$ », which cannot overflow. Note that adding the « $b \geq 0$ » precondition causes the sanitization check to neglect all the « $b < 0$ » cases. However, these cases are can be neglected, since if $b < 0$ then $a+b$ cannot overflow beyond INT_MAX.

Signed Integer Overflow



```
void func(int a, int b) {  
    if (b > 0 && a > INT_MAX - b) { O.F. BEYOND INT_MAX  
        /* Handle error */  
    }  
    if (b < 0 && a < INT_MIN - b) { O.F. BELOW INT_MIN  
        /* Handle error */  
    }  
    int result = a + b;  
    /* ... */  
}
```



This slide shows the complete sanitization check. Note that the sanitization check for the signed sum is much more complex than the corresponding one for the unsigned sum. This is because there are two possible overflows (beyond INT_MAX and below INT_MIN) and we have to add preconditions (si_b>0, si_b<0) to avoid overflows in the step 2 of the method.

Signed General Sanitizations



- Sum:

```
if (b >= 0 && a > INT_MAX - b){ ... }
if (b <= 0 && a < INT_MIN - b){ ... }
result = a + b;
```

- Subtraction:

```
if (b <= 0 && a > INT_MAX + b){ ... }
if (b >= 0 && a < INT_MIN + b){ ... }
result = a - b;
```

- Increment:

```
if (a == INT_MAX){ ... }
a++;
```

- Decrement:

```
if (a == INT_MIN){ ... }
a--;
```

- Multiplication:

(SEE AFTER)

- Division:

(case INT_MIN/-1 and division by zero)

```
if (b == 0){ ... }
if (b == -1 && a == INT_MIN){ ... }
result = a/b;
```

- Modulo:

```
if (b == 0){ ... }
if (b == -1 && a == INT_MIN){ ... }
result = a%b;
```

This slide shows the sanitization condition for all the signed integer operations (except multiplication). Note that the signed division (/) and remainder (%) can overflow in a single case, i.e. when $a == \text{INT_MIN}$ and $b = -1$. This is because the result would be $-\text{INT_MIN}$, but this number cannot be represented in 2's complement. Moreover, the case in which the divisor is 0 must be checked in both operations.

Signed Multiplication



```
void func(int a, int b) {  
    int result = a * b;  
    /* ... */  
}
```

- Hard to sanitize!
 - Two overflow conditions
 - General method has tricky points
- Recommended: sanitize a and b to be ≥ 0

The signed multiplication is hard to sanitize in the general case. This is because it has two overflow conditions and many tricky points to consider during the application of the general sanitization method. It is thus recommended to sanitize them to be non-negative, thus making them unsigned. Then, apply the sanitization rules for the unsigned ints seen before.

Signed Multiplication



- Overflow beyond INT_MAX:
 1. $a * b > \text{INT_MAX}$
 2. $(b > 0 \ \&\& \ a > \text{INT_MAX}/b) \ || \ (b < 0 \ \&\& \ a < \text{INT_MAX}/b)$
- Overflow below INT_MIN:
 1. $a * b < \text{INT_MIN}$
 2. $(b > 0 \ \&\& \ a < \text{INT_MIN}/b) \ || \ (b < 0 \ \&\& \ a > \text{INT_MIN}/b)$ -> the last condition may overflow if $b == -1$
 3. $(b > 0 \ \&\& \ a < \text{INT_MIN}/b) \ || \ (b < -1 \ \&\& \ a > \text{INT_MIN}/b)$

The general method can still be applied, but attention must be paid on step 2 and step 3. In step 2, when the operand b is algebraically moved to the second term, the case $b < 0$ must be addressed because it changes a «>» comparison to a «<» comparison and vice versa. Thus two preconditions « $b > 0$ » and « $b < 0$ » must be added to treat separately the two cases. The sanitization check for overflow beyond INT_MAX becomes « $(b > 0 \ \&\& \ a > \text{INT_MAX}/b) \ || \ (b < 0 \ \&\& \ a < \text{INT_MAX}/b)$ ». Note that the case $b == 0$ is left out safely, since in this case $a*b$ cannot overflow. The similar sanitization check for overflow below INT_MIN is « $(b > 0 \ \&\& \ a > \text{INT_MAX}/b) \ || \ (b < 0 \ \&\& \ a < \text{INT_MIN}/b)$ ». Note that this check causes an additional overflow in the operation « $\text{INT_MIN}/b$ » when $b == -1$. To address this, in the step 3 we modify the sanitization check in « $(b > 0 \ \&\& \ a > \text{INT_MAX}/b) \ || \ (b < -1 \ \&\& \ a < \text{INT_MIN}/b)$ ». Note that the case $b == -1$ is left out safely, since in this case $a*b$ cannot overflow beyond INT_MIN.

Signed Multiplication



```
void func(int a, int b) {
  if((b > 0 && a > INT_MAX/b)
    || (b < 0 && a < INT_MAX/b)) {
    /* Handle error */
  }
  if((b > 0 && a < INT_MIN/b)
    || (b < -1 && a > INT_MIN/b)) {
    /* Handle error */
  }
  int result = a * b;
  /* ... */
}
```




This slide shows the final sanitization check.

Quiz



```
signed char c1 = 100;
signed char c2 = 3;
signed char c3 = 4;
signed char cresult = c1 * c2 / c3;
printf("cresult = %d\n", cresult);
```

 **cresult = 75**


OR:

~~**cresult = 11 (overflow)**~~
?



In this code snippet, even if the partial result 100*3 is not representable with a signed char, «cresult» does not overflow like expected and takes the correct value 75. This is due to the *integer promotion* rule of C, which will be explained in a moment.

Implicit Integer Conversions



UNIVERSITÀ DI PISA

- Rules
 - Integer conversion rank
 - Integer promotions
 - Usual arithmetic conversions
- Integer conversion rank
 - Greater rank means *more or same* bits

signed char	<=	short	<=	int	<=	long	<=	long long
unsigned char	<=	unsigned short	<=	unsigned int	<=	unsigned long	<=	unsigned long long

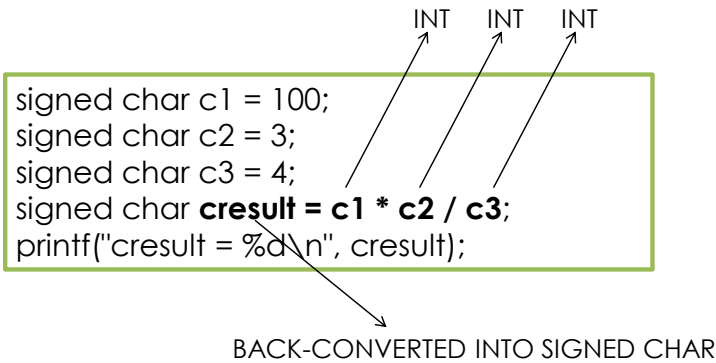
The C standard states three rules for implicit integer conversions: *integer conversion rank*, *integer promotion* and *usual arithmetic conversion*. All these rules are designed to minimize the possibility of «surprises» while operating with integers, without compromising too much the efficiency.

The integer conversion rank rule states that the five default integer types have a *rank* with the following increasing order: signed/unsigned char, signed/unsigned short, signed/unsigned int, signed/unsigned long, signed/unsigned long long. A higher-rank type cannot be represented on less bits than a lower-rank type, but it could be represented with the same number of bits.



Integer Promotions

- (Un)signed integers with rank < «int»: promoted to «int» for operations



25

Integer promotion rule says that every (signed or unsigned) integer with *rank* less than «int» is promoted to «int» (or «unsigned int» if «int» would not accomodate all the possible values) to perform operations. Finally, the result is back-converted to the original type.

Quiz



```
int si = -1;
unsigned int ui = 1;
if(si < ui) printf("-1 less than +1\n");
else printf("-1 NOT less than +1\n");
```

~~-1 less than +1~~

OR:

 -1 NOT less than +1

?



Intuitively, the program should print «-1 less than +1». However, due to the usual arithmetic conversion rules, «si» is converted to unsigned int before comparing it with «ui», and this results in an overflow. In most platforms, «si» will silently wrap to UINT_MAX, so the program will print «-1 NOT less than +1».

Usual Arithmetic Conversions




- Operations between mixed-type integers
 - E.g., unsigned long + int
 - Critical case:
 - operation mixes signed and unsigned operands, and
 - the signed one is negative
- The signed can be converted to the unsigned:
OVERFLOW

27



The usual arithmetic conversion rules apply in case of an operation with mixed-type integer operands, for example a sum between an unsigned long and an int. As a general rule, we must pay attention in those cases in which an operation mixes signed and unsigned operands, and the signed one may be negative. In these cases, usual arithmetic conversion rules may cause the signed operand to be converted into the unsigned one, thus resulting in an overflow.

Usual Arithmetic Conversions



UNIVERSITÀ DI PISA

1. *Same-signedness rule*: If both operands have same signedness, the smaller rank is converted into the higher
– E.g., unsigned long + unsigned int = unsigned long
2. *Mixed-signedness rule I*: Else, if the unsigned has rank \geq the signed, the signed is converted into the unsigned
– E.g., unsigned int + int = unsigned int (!!!)
3. *Mixed-signedness rule II*: Else, if the signed can represent all the values of the unsigned, the unsigned is converted into the signed
– E.g., unsigned int (on 32 bits) + long long (on 64 bits) = long long
4. *Mixed signedness rule III*: Else, both operands are converted to the unsigned type with the same rank of the signed operand
– E.g., unsigned long (on 64 bits) + long long (on 64 bits) = unsigned long long (!!!)




28

The usual arithmetic conversion rules are the following four:


- 1) Same-signedness rule: If both operands have the same signedness, then the one with the smaller rank will be converted to the type of the higher-rank one. For example, unsigned long + unsigned int = unsigned long. This rule is safe from overflows.
- 2) Mixed-signedness rule I: Else, if the unsigned operand has rank greater than or equal to the signed one, then the signed operand will be converted to the type of the unsigned one. For example, unsigned int + int = unsigned int. This rule produces an overflow if the signed operand is negative, like the one we saw in the previous slides.
- 3) Mixed-signedness rule II: Else, if the type of the signed operand is able to represent all the values of the type of the unsigned one, then the unsigned one is converted to the type of the signed one. For example, unsigned int (on 32 bits) + long long (on 64 bits) = long long. This rule is safe from overflows.
- 4) Mixed-signedness rule III: Else, both operands are converted to the unsigned type with the same rank of the signed operand. For example, unsigned long (on 64 bits) + long long (on 64 bits) = unsigned long long. This rule may produce an overflow if the signed operand is negative.

The last two rules are quite rare to meet in practice. The dangerous rules are those that can lead to information loss, which are the mixed-signedness rule I, and the (rare) mixed-signedness rule III. For example, in the quiz of the slide before, the unexpected behavior is caused by the mixed-signedness rule I.


Usual Arithmetic Conversions



```
void func(int si, unsigned int ui){  
  if(si < ui) printf("si less than ui\n");  
  else printf("si NOT less than ui\n");  
}
```



```
void func(int si, unsigned int ui){  
  if(si < 0 || si < ui) printf("si less than ui\n");  
  else printf("si NOT less than ui\n");  
}
```



29

The solution to the vulnerabilities due to the mixed-signedness rules is to exclude or treat separately the value ranges that would be loss, that are always the negative ones.

Usual Arithmetic Conversions



UNIVERSITÀ DI PISA



```
void func(int si, unsigned int ui){
  int res = si + ui;
  if(res < ui) { /* Handle error */ }
  /* ... */
}
```



```
void func(int si, unsigned int ui){
  int res;
  if(si < 0){
    if(ui > INT_MAX) { /* Handle error */ }
    res = si + (int)ui; /* int + int */
  }
  else{
    if(si > UINT_MAX - ui) { /* Handle error */ }
    res = si + ui; /* uint + uint */
  }
  /* ... */
}
```