C and C++ Secure Coding

Integers

Gianluca Dini
Dept. of Information Engineering
University of Pisa
Email: gianluca.dini@unipi.it
Version: 2022-05-03

1

C and C++ Secure Coding

INTEGERS

3

3

Introduction to Integer Security

- Constitute an underestimate source of vulnerabilities in C programs because boundary conditions have been intentionally ignored
- A software vulnerability may result when a program evaluates an integer to an unexpected value and then uses the value as an array index, size or loop counter
- Integer round checking has not been systematically applied so vulnerabilities do exist

4

Why are so many integer signed?

- C language lacks exception-handling mechanism
- Other mechanisms to return the status of a function
  - Call-by-reference parameter
    - Require to allocate a variable to test the status
  - Return value
    - If the function returns a value then you have to define a fictitious value (typically negative) to represent an error condition

5

Signed/Unsigned Integer Mismatch

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


6

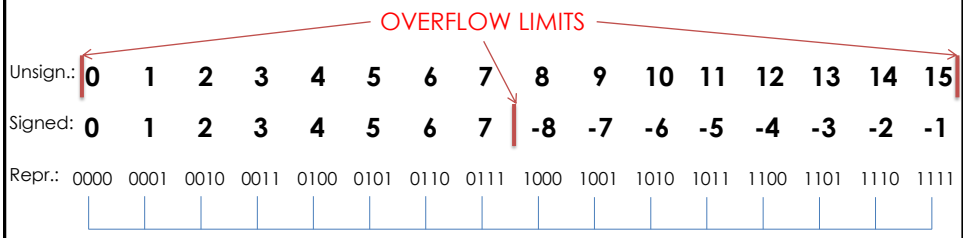
Signed/Unsigned Integer Mismatch

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

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

7

Integer Representation



- Unsigned integer wrap around: *wanted behaviour*
  15 + 1 = 0
- Signed integer overflow: *undefined behavior*
  7 + 1 = ? (usually: -8)

May-22 Secure Coding - Integers 8

8

WRAPAROUND, OVERFLOW AND SANITIZATION

9

9

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 */
}
```


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 */
}
```

only if I can restrict the domain of a and b

```
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 */
}
```

Integer Limits

- Unsigned: Range:
  - 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: Range:
  - 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]

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

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 */
}
```


General Sanitization Method

1. Write overflow condition «as is»
   a + b > UINT_MAX
   It may overflow, too!
2. Make it «safe» with an algebrical passage
   a > UINT_MAX - b
3. Avoid additional overflows
   (UINT_MAX - b cannot overflow)

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

General Sanitization Method

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

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


16

Unsigned General Sanitizations

- Sum:

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

* better version exists (see after)

- Subtraction:

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

- Increment:

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

* better version exists (see after)

- Decrement:

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

- Multiplication:

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

- Division:
  (no wrap, but check for division by zero)

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

- Modulo:
  (idem)

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


17

Overflow-Tolerant Sanitizations

- 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.), independed of limits.h → Less error-prone!
- Do not use with *signed* integers!

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)

Signed Integer Overflow

- Overflow beyond INT_MAX:
  1. Write overflow condition «as is»
     a + b > INT_MAX
  2. Make it «safe» with an algebrical passage
     a > INT_MAX - b -> it may overflow again if b < 0!
  3. Avoid additional overflows
     b >= 0 && a > INT_MAX - b

  (if b < 0, a + b can't overflow beyond INT_MAX -> ok)
- Overflow below INT_MIN (underflow):
  1. a + b < INT_MIN
  2. a < INT_MIN - b -> it may overflow again if b > 0!
  3. b <= 0 && a < INT_MIN - b

  (if b > 0, a + b can't overflow below INT_MIN -> ok)

Signed Integer Overflow

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

O.F. BEYOND INT_MAX

O.F. BELOW INT_MIN

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;
```


22

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

23

Signed Multiplication

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

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;
  /* ... */
}
```

Integers

CONVERSIONS

26

26

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)~~

?

May-22 Secure Coding - Integers 27

27

Conversions

- Conversion is a change in the underlying type used to represent a value
- A conversion occurs upon casting, assignment, expressions
- A conversion from a type to a *wider* type generally preserves the mathematical value
- A conversion in the opposite direction may cause a loss of high order bits

28

Conversions

- Rules used by compiler to manage conversions
  - Integer conversion rank
  - Integer promotions
  - Usual arithmetic conversions

29

Integer conversion rank [→]

- Provides a standard rank ordering of integer types that is used to determine a common type for computations
- Rank increases with precision
  - long long int unsigned long log
  - long int unsigned long
  - int unsigned int
  - short unsigned shor
  - signed char unsigned char char

32

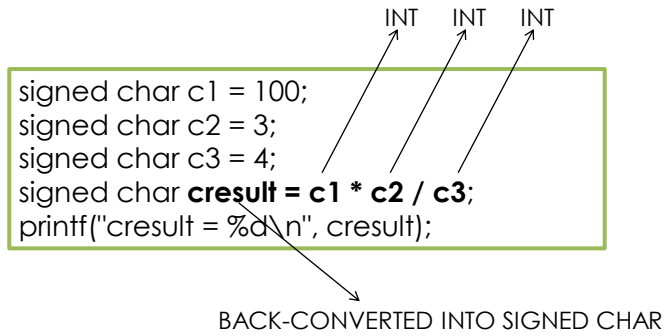
Integer conversion rank [→]

- Rank is transitive
- Any aspects not covered above are implementation defined
- Disclaimer: extended types (e.g., size_t) and _Bool are not addressed here

33

Integer Promotions

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



34

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 ?

35

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

36

Usual Arithmetic Conversions (→)

- After integer promotions are performed on both operands, the following rules are applied
  1. If both operands have the same type, no conversion is needed
  2. Otherwise if both operands have signed integer types or both have unsigned integer types, the the type at lower conversion rank is converted to the type at higher rank
     - signed int + signed long → signed long

37

Usual Arithmetic Conversions (→)

3. Otherwise, if one operand has an unsigned integer type and the other operand has a signed integer type, the following rules apply:
   A. If the rank for the unsigned type is greater than, or equal to, the rank for the signed integer type, the signer integer type is converted to the unsigned integer type
      – signed int and unsigned int → unsigned int
   B. Otherwise, if the signed type can represent all the values of the unsigned integer type, the unsigned integer type is converted to signed type
      – 64-bit signed long + 32-bit unsigned int → signed long

38

Usual Arithmetic Conversions (→)

   C. Otherwise, both types are converted to the unsigned integer type corresponding to the signed integer type
      – 64-bit unsigned long + 64-bit long long → unsigned long long

- Rules 3.A and 3.C (rare) are dangerous because they are conducive to information loss

39

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");
}
```

41

Usual Arithmetic Conversions

```
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 */
  }
  /* … */
}
```

42

Conversions from unsigned types (→)

- unsigned → unsigned
  - smaller type → larger type is always safe
    - Zero-extension
  - larger type → smaller
    - For unsigned integer types only, C specifies reduction modulo $2^{width(type)}$
    - Well-defined behaviour
    - Example
      - unsigned int ui = 300;
      - unsigned char = ui; /* 44 */

Conversions from unsigned types [→]

- unsigned → signed
  - Lost or misrepresented data when a value cannot be represented in the signed type
- large unsigned → signed of the same width
  - If the unsigned value is not representable on the signed type
    - The result is implementation defined
    - An implementation-defined signal is raised
  - Implementation behaviour causes porting issues, if unanticipated a likely source of errors

Conversions from unsigned types (→)

- A common implementation is to not raise a signal but preserve the bit pattern, so no data is lost
  - unsigned int ui = UINT_MAX – 1
  - int i = ui; /* -2 */

- large unsigned → smaller signed
  - Truncation and the MSB becomes the sign bit
  - Data will be lost or misinterpreted if the value cannot be represented in the signed type
  - If the programmer does not anticipate, the programming errors or vulnerabilities

Conversions from unsigned types [→]

```
unsigned long int ul = ULONG_MAX;
signed char sc;
sc = (signed char)ul;
```

```
unsigned long int ul = ULONG_MAX;
signed char sc;
if (ul <= SCHAR_MAX){
        sc = (signed char)ul;
}
else { /* handle error */ }
```

Ranges should be validated when converting from unsigned to signed

Conversions from signed types (→)

- Smaller signed type → larger signed type is always safe
  - Sign extension
- Signed type → smaller signed type width
  - Implementation defined or
  - Raise an implementation-defined signal
  - Typical implementation is truncation
  - If not anticipated, programming error or vulnerability may occur

47

Conversions from signed types (→)

```
signed long int sl = LONG_MAX;
signed char sc; sc = (signed char)ul; /* -1 */
```

```
signed long int sl = LONG_MAX;
signed char sc;
if ( (sl < SCHAR_MIN)  | |  (sl > SCHAR_MAX)) {
        /* handle error condition */
}
else {
          sc = (signed char)sl;
```

Conversions from signed types with greater precision to signed types with lesser precision require both the upper and lower bounds to be checked

48

Conversions from signed types (→)

- (signed → unsigned) && (w(signed) > w(unsigned))
  - Preserve low order bits
- (signed → unsigned) && (w(signed) < w(unsigned))
  - Sign-extend to the signed type corresponding to unsigned and convert
    - char → unsigned long: sign estend to long and convert to unsigned long
- (signed → unsigned) && (w(signed) = w(unsigned))
  - Bit pattern is preserved; msb loses its function of sign bit

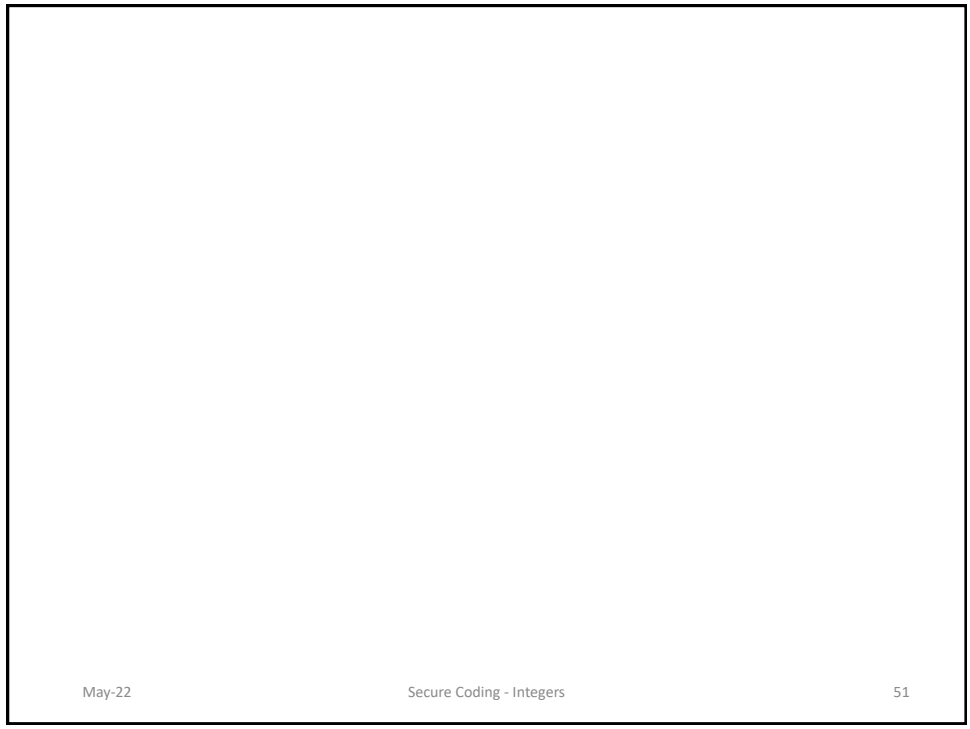
49

Conversions from signed types

```
unsigned int ui = UINT_MAX;
signed char sc = -1;
if (sc == ui){
        puts ("Why is -1 = 4,294,967,295?");
 }
```

```
signed int si = INT_MIN;
unsigned int ui = (unsigned int)si;
```


50



51