Secure Coding in C++: Integers

Robert C. Seacord
SD Best Practices 2006

Software Engineering Institute

About this Presentation

Presentation assumes basic C++ programming skills but does not assume in-depth knowledge of software security

Ideas generalize but examples are specific to

- Microsoft Visual Studio
- Linux/GCC
- 32-bit Intel Architecture (IA-32)

Material in this presentation was borrowed from the Addison-Wesley book *Secure Coding in C and C++*

Integer Security

Integers represent a growing and underestimated source of vulnerabilities in C++ programs.

Integer range checking has not been systematically applied in the development of most C++ software.

- security flaws involving integers exist
- a portion of these are likely to be vulnerabilities

Unexpected Integer Values

An unexpected value is a value other than the one you would expect to get using a pencil and paper

Unexpected values are a common source of software vulnerabilities (even when this behavior is correct).

Integer Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

Summary

Integer Section Agenda

Representation

Types

Conversions

Error conditions

Operations

Two’s Complement

The two’s complement form of a negative integer is created by adding one to the one’s complement representation.

```
0 0 1 0  1 0 0 1          0 0 1 0  1 0 0 1
↓ ↓ ↓ ↓  ↓ ↓ ↓ ↓          ↓ ↓ ↓ ↓  ↓ ↓ ↓ ↓
1 1 0 1  0 1 1 0 + 1 =    1 1 0 1  0 1 1 1
```

Two’s complement representation has a single (positive) value for zero.

The sign is represented by the most significant bit.

The notation for positive integers is identical to their signed-magnitude representations.

Integer Section Agenda

Representation

Types

Conversions

Error conditions

Operations

Signed and Unsigned Types

Integers in C++ are either signed or unsigned.

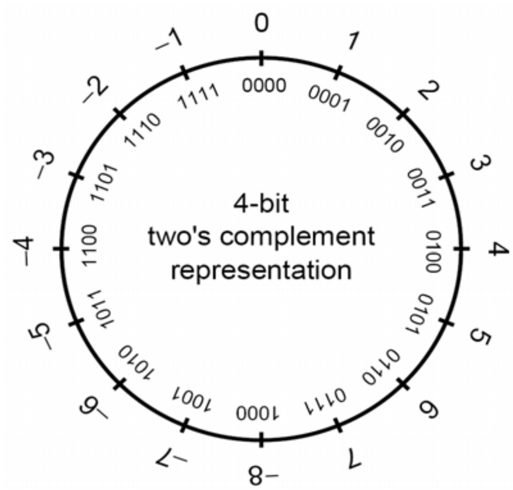
For each signed type there is an equivalent unsigned type.

Signed Integers

Signed integers are used to represent positive and negative values.

On a computer using two's complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1}-1$.

Signed Integer Representation



CERT

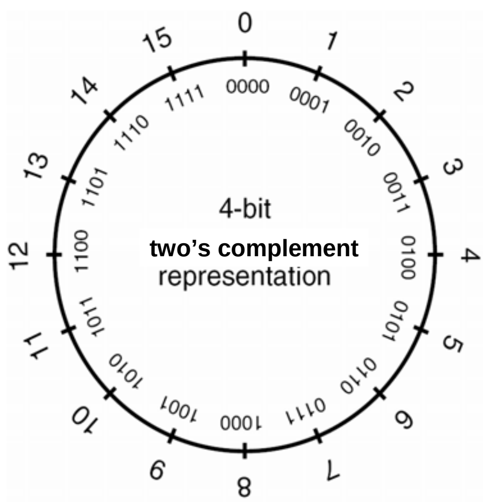
Unsigned Integers

Unsigned integer values range from zero to a maximum that depends on the size of the type

This maximum value can be calculated as 2^n-1, where n is the number of bits used to represent the unsigned type.

CERT

Unsigned Integer Representation



Standard Integer Types

Standard integers include the following types, in non-decreasing length order:

- **`signed char`**
- **`short int`**
- **`int`**
- **`long int`**
- **`long long int`**

NOTE: The **`long long int`** type is not defined in ISO/IEC 14882:2003 but is defined in the 2006-04-21 working draft and many implementations

Other Integer Types

The following types are used for special purposes

- `ptrdiff_t` is the signed integer type of the result of subtracting two pointers
- `size_t` is the unsigned result of the **`sizeof`** operator
- `wchar_t` is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

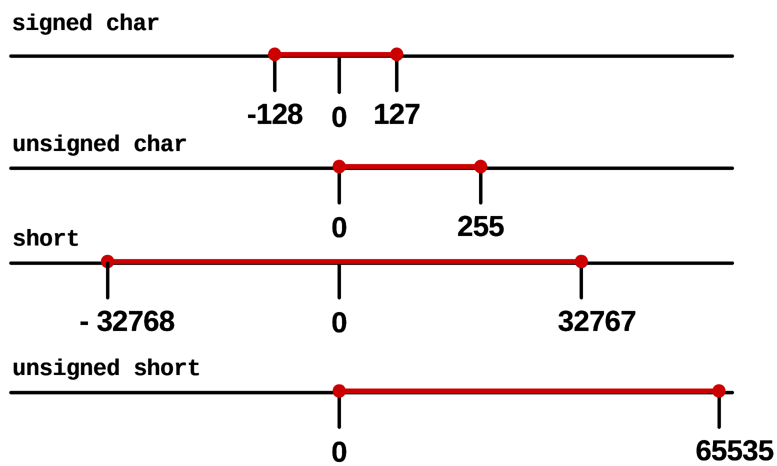
Integer Ranges

Minimum and maximum values for an integer type depend on

- the type’s representation
- signedness
- the number of allocated bits

The standard sets minimum requirements for these ranges.

Example Integer Ranges



Integer Section Agenda

Representation

Types

Conversions

Error conditions

Operations

Integer Conversions

Type conversions occur explicitly in C++ as the result of a cast or implicitly as required by an operation.

Conversions can lead to lost or misinterpreted data.

Implicit conversions are a consequence of the C++ ability to perform operations on mixed types.

The following rules influence how conversions are performed:

- integer promotions
- integer conversion rank
- usual arithmetic conversions

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 **`int`**
- otherwise, it is converted to **`unsigned int`**

Integer Promotion Purpose

Integer promotions require the promotion of each variable (**c1** and **c2**) to **int** size.

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

The two **int**s are added and the sum truncated to fit into the **char** type.

Integer promotions avoid arithmetic errors from the overflow of intermediate values.

Integer Promotion Example

```
1. char cresult, c1, c2, c3;
2. c1 = 100;
3. c2 = 90;
4. c3 = -120;
5. cresult = c1 + c2 + c3;
```

The sum of **c1** and **c2** exceeds the maximum size of **signed char**.

However, **c1**, **c2**, and **c3** are each converted to integers and the overall expression is successfully evaluated.

The sum is truncated and stored in **cresult** without a loss of data.

The value of **c1** is added to the value of **c2**.

Integer Promotions Consequences

Adding two small integer types always results in a value of type **`signed int`** or **`unsigned int`** and the actual operation takes place in this type

Applying the bitwise negation operator ~ to an unsigned char (on IA-32) results in a negative value of type **`signed int`** because the value is zero-extended to 32 bits.

Integer Conversion Rank

Every integer type has an integer conversion rank that determines how conversions are performed.

Usual Arithmetic Conversions

Set of rules that provides a mechanism to yield a common type when

- Both operands of a binary operator are balanced to a common type
- The second and third arguments of the conditional operator **(? :)** are balanced to a common type

Balancing conversions involve two operands of different types

One or both operands may be converted

Unsigned Integer Conversions 1

Conversions of smaller unsigned integer types to larger unsigned integer types is

- always safe
- typically accomplished by zero-extending the value

When a larger unsigned integer is converted to a smaller unsigned integer type, the

- larger value is truncated
- low-order bits are preserved

Unsigned Integer Conversions 2

When unsigned integer types are converted to the corresponding signed integer type

- the bit pattern is preserved so no data is lost
- the high-order bit becomes the sign bit

If the sign bit is set, both the sign and magnitude of the value change.

From unsigned	To	Method
char	char	Preserve bit pattern; high-order bit becomes sign bit
char	short	Zero-extend
char	long	Zero-extend
char	unsigned short	Zero-extend
char	unsigned long	Zero-extend
short	char	Preserve low-order byte
short	short	Preserve bit pattern; high-order bit becomes sign bit
short	long	Zero-extend
short	unsigned char	Preserve low-order byte
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	long	Preserve bit pattern; high-order bit becomes sign bit
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word

Key: Lost data | Misinterpreted data

Signed Integer Conversions 1

When a signed integer is converted to an unsigned integer of equal or greater size and the value of the signed integer is not negative

- the value is unchanged
- the signed integer is sign-extended

A signed integer is converted to a shorter signed integer by truncating the high-order bits.

Signed Integer Conversions 2

When signed integer types are converted to the corresponding unsigned integer type

- bit pattern is preserved—no lost data
- high-order bit loses its function as a sign bit

If the value of the signed integer is not negative, the value is unchanged.

If the value is negative, the resulting unsigned value is evaluated as a large, unsigned integer.

From	To	Method
char	short	Sign-extend
char	long	Sign-extend
char	unsigned char	Preserve pattern; high-order bit loses function as sign bit
char	unsigned short	Sign-extend to short; convert short to unsigned short
char	unsigned long	Sign-extend to long; convert long to unsigned long
short	char	Preserve low-order byte
short	long	Sign-extend
short	unsigned char	Preserve low-order byte
short	unsigned short	Preserve bit pattern; high-order bit loses function as sign bit
short	unsigned long	Sign-extend to long; convert long to unsigned long
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word
long	unsigned long	Preserve pattern; high-order bit loses function as sign bit

Key: Lost data | Misinterpreted data

CERT

Conversion Summary

Necessary to avoid conversions that result in

- Loss of value: conversion to a type where the magnitude of the value cannot be represented
- Loss of sign: conversion from a signed type to an unsigned type resulting in loss of sign

The only integer type conversion guaranteed safe for all data values and all conforming implementations is to a wider type of the same signedness

CERT

Integer Section Agenda

Representation

Types

Conversions

Error conditions

Operations

CERT

Integer Error Conditions

Integer operations can resolve to unexpected values as a result of an

- overflow
- sign error
- truncation

CERT

Overflow

An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.

Overflows can be signed or unsigned.

A signed overflow occurs when a value is carried over to the sign bit.

An unsigned overflow occurs when the underlying representation can no longer represent a value.

Overflow Examples 1

```
int i;
unsigned int j;

i = INT_MAX;  // 2,147,483,647
i++;
printf("i = %d\n", i);        i=-2,147,483,648

j = UINT_MAX; // 4,294,967,295;
j++;
printf("j = %u\n", j);        j = 0
```

Overflow Examples 2

```
 9. i = INT_MIN; // -2,147,483,648;
10. i--;
11. printf("i = %d\n", i);
```

i = 2,147,483,647

```
12. j = 0;
13. j--;
14. printf("j = %u\n", j);
```

j = 4,294,967,295

Truncation Errors

Truncation errors occur when

- an integer is converted to a smaller integer type and
- the value of the original integer is outside the range of the smaller type

Low-order bits of the original value are preserved and the high-order bits are lost.

Truncation Error Example

```
1. char cresult, c1, c2;
2. c1 = 100;
3. c2 = 90;
4. cresult = c1 + c2;
```

Adding **`c1`** and **`c2`** exceeds the max size of **`signed char (+127)`**

Truncation occurs when the value is assigned to a type that is too small to represent the resulting value

Integers smaller than **`int`** are promoted to **`int`** or **`unsigned int`** before being operated on

CERT

Sign Errors

Can occur when

- converting an unsigned integer to a signed integer
- converting a signed integer to an unsigned integer

CERT

Converting to Signed Integer

Converting an unsigned integer to a signed integer of

- equal size - preserve bit pattern; high-order bit becomes sign bit
- greater size - the value is zero-extended then converted
- lesser size - preserve low-order bits

If the high-order bit of the unsigned integer is

- not set - the value is unchanged
- set - results in a negative value

Converting to Unsigned Integer

Converting a signed integer to an unsigned integer of

- equal size - bit pattern of the original integer is preserved
- greater size - the value is sign-extended then converted
- lesser size - preserve low-order bits

If the value of the signed integer is

- not negative - the value is unchanged
- negative - a (typically large) positive value

Sign Error Example

```
1. int i = -3;

2. unsigned short u;

3. u = i;

4. printf("u = %hu\n", u);
```

Implicit conversion to smaller unsigned integer

There are sufficient bits to represent the value so no truncation occurs. The two's complement representation is interpreted as a large signed value, however, so **`u = 65533`**.

Integer Section Agenda

Representation

Types

Conversions

Error conditions

Operations

Integer Operations

Integer operations can result in errors and unexpected values.

Unexpected integer values can cause

- unexpected program behavior
- security vulnerabilities

Most integer operations can result in exceptional conditions.

Integer Addition

Addition can be used to add two arithmetic operands or a pointer and an integer.

If both operands are of arithmetic type, the usual arithmetic conversions are performed on them.

Integer addition can result in an overflow if the sum cannot be represented in the allocated bits.

Integer Multiplication

Multiplication is prone to overflow errors because relatively small operands can overflow.

One solution is to allocate storage for the product that is twice the size of the larger of the two operands.

The max product for an unsigned integer is 2^n-1

- $2^n-1 \times 2^n-1 = 2^{2n} - 2^{n+1} + 1 < 2^{2n}$

The minimum product for a signed integer is -2^{n-1}

- $-2^{n-1} \times -2^{n-1} = 2^{2n-2} < 2^{2n}$

Multiplication Instructions

The IA-32 instruction set includes a

- `mul` (unsigned multiply) instruction
- `imul` (signed multiply) instruction

Unsigned Multiplication

```
if (OperandSize == 8) {
   AX = AL * SRC;
else {
   if (OperandSize == 16) {
      DX:AX = AX * SRC;
   }
   else {  // OperandSize == 32
      EDX:EAX = EAX * SRC;
   }
}
```

Product of 8-bit operands is stored in 16-bit destination registers

Product of 16-bit operands is stored in 32-bit destination registers

Product of 32-bit operands is stored in 64-bit destination registers

Upcasting

Cast both operands to an integer with at least 2x bits and then multiply.

For unsigned integers

- Check high-order bits in the next larger integer.
- If any are set, throw an error.

For signed integers, all zeros or all ones in the high-order bits and the sign bit in the low-order bit indicate no overflow.

Upcast Example

```
void* AllocBlocks(size_t cBlocks) {
  // allocating no blocks is an error
  if (cBlocks == 0) return NULL;
  // Allocate enough memory
  // Upcast the result to a 64-bit integer
  // and check against 32-bit UINT_MAX
  // to make sure there's no overflow
  unsigned long long alloc = cBlocks * 16;
  return (alloc < UINT_MAX)
    ? malloc(cBlocks * 16)
    : NULL;
}
```

Multiplication results in a 32-bit value. The result is assigned to an **unsigned long long** but the calculation may have already overflowed.

Standard Compliance

To be compliant with the standard, multiplying two 32-bit numbers in this context must yield a 32-bit result.

The language was not modified because the result would be burdensome on architectures that do not have widening multiply instructions.

The correct result could be achieved by casting one of the operands.

Corrected Upcast Example

```
void* AllocBlocks(size_t cBlocks) {
  // allocating no blocks is an error
  if (cBlocks == 0) return NULL;
  // Allocate enough memory
  // Upcast the result to a 64-bit integer
  // and check against 32-bit UINT_MAX
  // to make sure there's no overflow
  unsigned long long alloc =
           (unsigned long long)cBlocks*16;
  return (alloc < UINT_MAX)
    ? malloc(cBlocks * 16)
    : NULL;
}
```


Integer Division

An integer overflow condition occurs when the minimum integer value for 32-bit or 64-bit integers is divided by -1.

- In the 32-bit case, –2,147,483,648/-1 should be equal to 2,147,483,648.

- 2,147,483,648 /-1 = - 2,147,483,648

- Because 2,147,483,648 cannot be represented as a signed 32-bit integer, the resulting value is incorrect.

Error Detection

The Intel division instructions do not set the overflow flag.

A division error is generated if

- the source operand (divisor) is zero
- the quotient is too large for the designated register

A divide error results in a fault on interrupt vector 0.

When a fault is reported, the processor restores the machine state to the state before the beginning of execution of the faulting instruction.

Microsoft Visual Studio

C++ exception handling does not allow recovery from

- a hardware exception
- a fault such as
  - an access violation
  - divide by zero

Visual Studio provides structured exception handling (SEH) facility for dealing with hardware and other exceptions

Structured exception handling is an operating system facility that is distinct from C++ exception handling.

C++ Exception Handling

```
Sint operator /(unsigned int divisor) {
  try {
    return ui / divisor;
  }
  catch (...) {
    throw SintException(
         ARITHMETIC_OVERFLOW
     );
  }
}
```

C++ exceptions in Visual C++ are implemented using structured exceptions, making it possible to use C++ exception handling on this platform.

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Summary

Vulnerabilities

A vulnerability is a set of conditions that allows violation of an explicit or implicit security policy.

Security flaws can result from hardware-level integer error conditions or from faulty logic involving integers.

These security flaws can, when combined with other conditions, contribute to a vulnerability.

Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional

JPEG Example

Based on a real-world vulnerability in the handling of the comment field in JPEG files.

Comment field includes a two-byte length field indicating the length of the comment, including the two-byte length field.

To determine the length of the comment string (for memory allocation), the function reads the value in the length field and subtracts two.

The function then allocates the length of the comment plus one byte for the terminating null byte.

Integer Overflow Example

```
void getComment(unsigned int len, char *src) {
  unsigned int size;
  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}
int main(int argc, char *argv[]) {
  getComment(1, "Comment ");
  return 0;
}
```

0 byte **malloc()** succeeds

Size is interpreted as a large positive value of **0xffffffff**

Possible to cause an overflow by creating an image with a comment length field of 1

Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional

2006 Carnegie Mellon University 63

CERT

Sign Error Example 1

```
1. #define BUFF_SIZE 10
2. int main(int argc, char* argv[]){
3.   int len;
4.   char buf[BUFF_SIZE];
5.   len = atoi(argv[1]);
6.   if (len < BUFF_SIZE){
7.     memcpy(buf, argv[2], len);
8.   }
9. }
```

Program accepts two arguments (the length of data to copy and the actual data)

len declared as a signed integer

argv[1] can be a negative value

A negative value bypasses the check

Value is interpreted as an unsigned value of type size_t

2006 Carnegie Mellon University 64

CERT

Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional

Vulnerable Implementation

```
bool func(char *name, long cbBuf) {
  unsigned short bufSize = cbBuf;
  char *buf = (char *)malloc(bufSize);
  if (buf) {
    memcpy(buf, name, cbBuf);
    if (buf) free(buf);
    return true;
  }
  return false;
}
```

cbBuf is used to initialize **bufSize**, which is used to allocate memory for **buf**

cbBuf is declared as a **long** and used as the size in the **memcpy()** operation

Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional

Non-Exceptional Integer Errors

Integer-related errors can occur without an exceptional condition (such as an overflow) occurring.

Negative Indices

```
int *table = NULL;
int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *)malloc(sizeof(int) * 100);
  }
  if (pos > 99) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

Storage for the array is allocated on the heap

pos is not > 99

value is inserted into the array at the specified position

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

Summary

Mitigation Section Agenda

Type range checking

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Type Range Checking

Type range checking can eliminate integer vulnerabilities.

Languages such as Pascal and Ada allow range restrictions to be applied to any scalar type to form subtypes.

Ada allows range restrictions to be declared on derived types using the range keyword:

```
type day is new INTEGER range 1..31;
```

Range restrictions are enforced by the language runtime.

C++ lacks an equivalent mechanism

Type Range Checking Example

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
  unsigned int len;
  char buf[BUFF_SIZE];
  len = atoi(argv[1]);
  if ((0<len) && (len<BUFF_SIZE) ){
    memcpy(buf, argv[2], len);
  }
  else
    printf("Too much data\n");
}
```

Implicit type check from the declaration as an unsigned integer

Explicit check for both upper and lower bounds

Range Checking

External inputs should be evaluated to determine whether there are identifiable upper and lower bounds.

- These limits should be enforced by the interface.
- It's easier to find and correct input problems than it is to trace internal errors back to faulty inputs.

Limit input of excessively large or small integers.

Typographic conventions can be used in code to

- distinguish constants from variables
- distinguish externally influenced variables from locally used variables with well-defined ranges

Mitigation Section Agenda

Type range checking

Types

Compiler checks

Safe integer operations

Testing and reviews

CERT

Types

One way to provide better type checking is to provide better types.

Using an unsigned type can guarantee that a variable does not contain a negative value.

This solution does not prevent overflow.

Strong typing should be used so that the compiler can be more effective in identifying range problems.

CERT

Problem: Representing Object Size

Really bad:

```
short total = strlen(argv[1])+ 1;
```

Better:

```
size_t total = strlen(argv[1])+ 1;
```

Better still:

```
rsize_t total = strlen(argv[1])+ 1;
```


CERT

Problem with `size_t`

Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly.

As we have seen, negative numbers appear as very large positive numbers when converted to an unsigned type like **`size_t`**.

CERT

rsize_t

rsize_t cannot be greater than **RSIZE_MAX**.

For applications targeting machines with large address spaces, **RSIZE_MAX** should be defined as the smaller of

- the size of the largest object supported
- **(SIZE_MAX >> 1) (**even if this limit is smaller than the size of some legitimate, but very large, objects)

rsize_t is the same type as **size_t** so they are binary compatible

Mitigation Section Agenda

Type range checking

Types

Compiler checks

Safe integer operations

Testing and reviews

Visual C++ Compiler Checks

Visual C++ .NET 2003 generates a warning (C4244) when an integer value is assigned to a smaller integer type.

- At level 1 a warning is issued if `__int64` is assigned to `unsigned int`.
- At level 3 and 4, a "possible loss of data" warning is issued if an integer is converted to a smaller type.

For example, the following assignment is flagged at warning level 4:

```
int main() {
   int b = 0, c = 0;

   short a = b + c;    // C4244
}
```


Visual C++ Runtime Checks

Visual C++ .NET 2003 includes runtime checks that catch truncation errors as integers are assigned to shorter variables that result in lost data.

The `/RTCc` compiler flag catches those errors and creates a report.

Visual C++ includes a `runtime_checks` pragma that disables or restores the `/RTC` settings but does not include flags for catching other runtime errors such as overflows.

Runtime error checks are not valid in a release (optimized) build for performance reasons.

GCC Runtime Checks

GCC compilers provide an `-ftrapv` option

- provides limited support for detecting integer exceptions at runtime
- generates traps for signed overflow for addition, subtraction, and multiplication
- generates calls to existing library functions

GCC runtime checks are based on post-conditions—the operation is performed and the results are checked for validity

Mitigation Section Agenda

Type range checking

Types

Compiler checks

Safe integer operations

Testing and reviews

SafeInt Class

SafeInt is a C++ template class written by David LeBlanc.

Implements a precondition approach that tests the values of operands before performing an operation to determine if an error will occur.

The class is declared as a template, so it can be used with any integer type.

Every operator has been overridden except for the subscript **`operator[]`**.

Precondition Example

Overflow occurs when **`lhs`** and **`rhs`** are **`unsigned int`** and

```
lhs + rhs > UINT_MAX
```

To prevent the addition from overflowing the **`operator+ can`** test that

```
lhs > UINT_MAX – rhs
```

Or alternatively :

```
~lhs < rhs
```

SafeInt Example

The variables **s1** and **s2** are declared as SafeInt types

```
int main(int argc, char *const *argv) {
  try{
    SafeInt<unsigned long> s1(strlen(argv[1]));
    SafeInt<unsigned long> s2(strlen(argv[2]));
    char *buff = (char *) malloc(s1 + s2 + 1);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
  }
  catch(SafeIntException err) {
    abort();
  }
}
```

When the **+** operator is invoked it uses the safe version of the operator implemented as part of the SafeInt class.

When to Use Safe Integers

Use safe integers when integer values can be manipulated by untrusted sources such as

- the size of a structure
- the number of structures to allocate

```
void* CreateStructs(int StructSize, int HowMany) {
   SafeInt<unsigned long> s(StructSize);

   s *= HowMany;
   return malloc(s.Value());
}
```

Structure size multiplied by # required to determine size of memory to allocate

The multiplication can overflow the integer and create a buffer overflow vulnerability

When Not to Use Safe Integers

Don’t use safe integers when no overflow is possible.

- tight loop
- variables are not externally influenced

```
…
char a[INT_MAX];
for (size_t i = 0; i < INT_MAX; i++)
  a[i] = '\0';
…
```


SafeInt Summary

SafeInt advantages:

- Portability - does not depend on assembly language instructions
- Usability
  - operators can be used in inline expressions
  - uses C++ exception handling

SafeInt issues:

- Incorrect behavior - fails to provide correct integer promotion behavior.
- Performance

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Summary

Summary

The key to preventing integer vulnerabilities is to understand integer behavior in digital systems.

Concentrate on integers used as indices (or other pointer arithmetic), lengths, sizes, and loop counters

- Use safe integer operations to eliminate exception conditions
- Range check all integer values used as indices.
- Use **`size_t`** or **`rsize_t`** for all sizes and lengths (including temporary variables)

For More Information

Visit the CERT® web site

http://www.cert.org/secure-coding/

Contact Presenter

Robert C. Seacord rcs@cert.org

Contact CERT Coordination Center

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh PA 15213-3890

Hotline**: 412-268-7090**
CERT/CC personnel answer 8:00 a.m.–5:00 p.m. and are on call for emergencies during other hours.

Fax: **412-268-6989**

E-mail: **cert@cert.org**