Secure Coding in C and C++
Integral Security

Robert C. Seacord

FIRST Conference : June 26, 2006

Software Engineering Institute

About this Presentation

Derived from the Addison-Wesley book "Secure Coding in C and C++"

Presentation assumes basic C/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)

An Integer Story 1

GNU’s Bourne Again Shell (bash) is a drop-in replacement for the Bourne shell (/bin/sh).

- same syntax as the standard shell but provides additional functionality such as job control, command-line editing, and history.
- most prevalent use is on Linux.

A vulnerability exists in bash versions 1.14.6 and earlier where bash can be tricked into executing arbitrary commands.

An Integer Story 2

Bash contains an incorrectly declared variable in the **`yy_string_get()`** function responsible for parsing the user-provided command line into separate tokens.

The error involves the variable **`string`**, which has been declared to be of type **`char *`**.

The **`string`** variable is used to traverse the character string containing the command line to be parsed.

An Integer Story 3

As characters are retrieved from this pointer, they are stored in a variable of type `int`.

For compilers in which the `char` type defaults to `signed char`, this value is sign-extended when assigned to the `int` variable.

For character code 255 decimal (-1 in two’s complement form), this sign extension results in the value -1 being assigned to the integer.

-1 is used in other parts of the parser to indicate the end of a command.

An Integer Story 4

The character code 255 decimal (377 octal) serves as an unintended command separator for commands given to bash via the -c option.

Example:

- `bash -c 'ls\377who'`

(where `\377` represents the single character with value 255 decimal) executes two commands, ls and who.

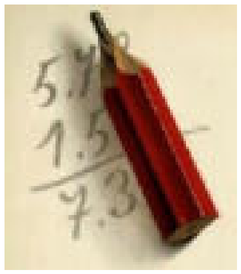
Integer Security

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

Integer range checking has not been systematically applied in the development of most C and 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 value 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

Integer Representation

Signed magnitude

One’s complement

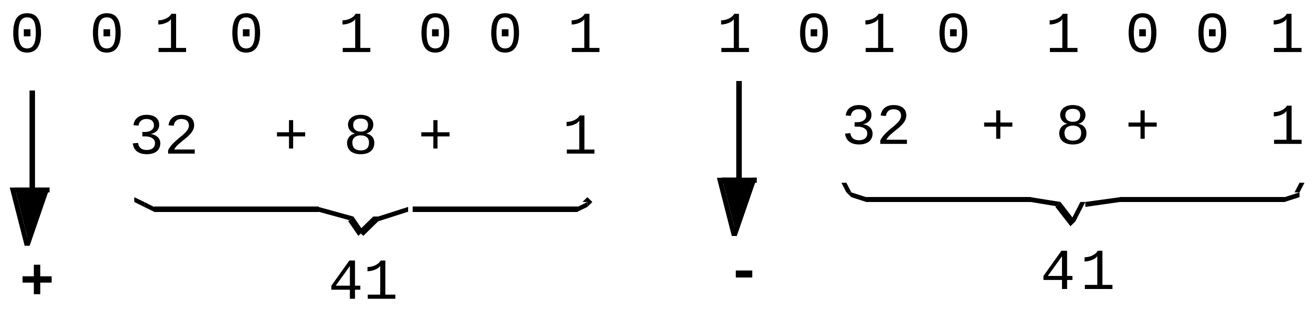
Two’s complement

These integer representations vary in how they represent negative numbers.

Signed-Magnitude Representation

Uses the high-order bit to indicate the sign

- 0 for positive
- 1 for negative
- remaining low-order bits indicate the magnitude of the value

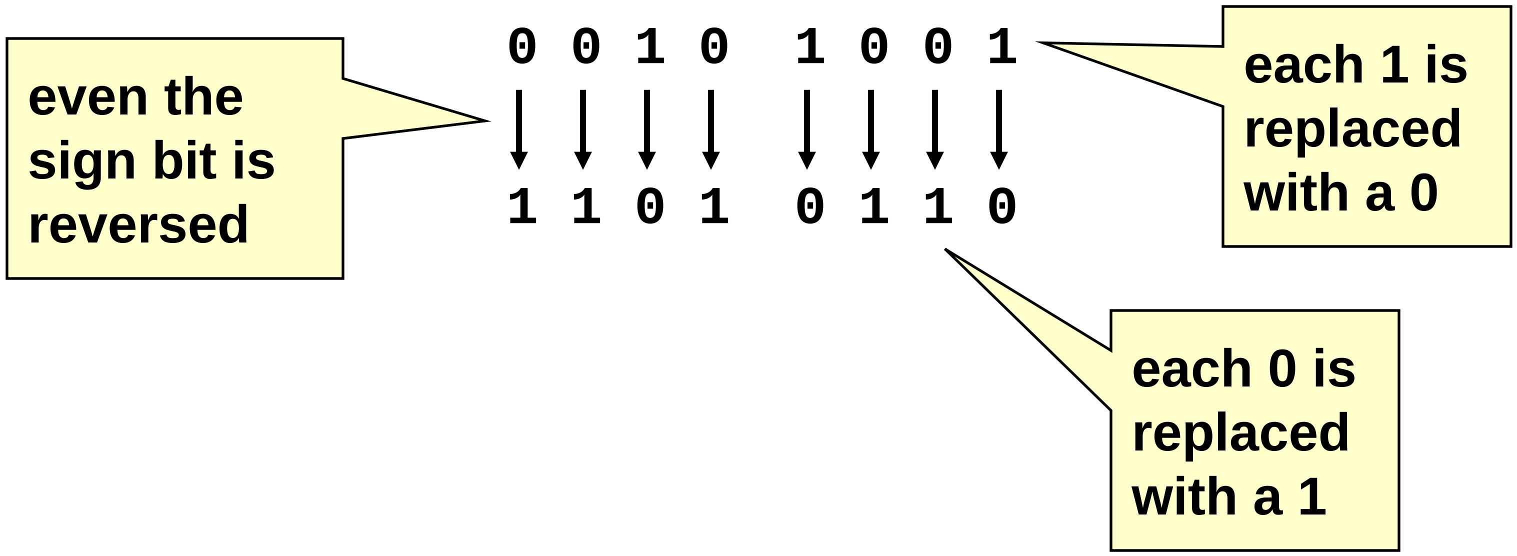


Signed-magnitude representation of +41 and -41

One’s Complement

One’s complement replaced signed magnitude because the circuitry was too complicated.

Negative numbers are represented in one’s complement form by complementing each bit


0 0 1 0 1 0 0 1
1 1 0 1 0 1 1 0
even the sign bit is reversed
each 1 is replaced with a 0
each 0 is replaced with a 1

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 and C++ are either signed or unsigned.

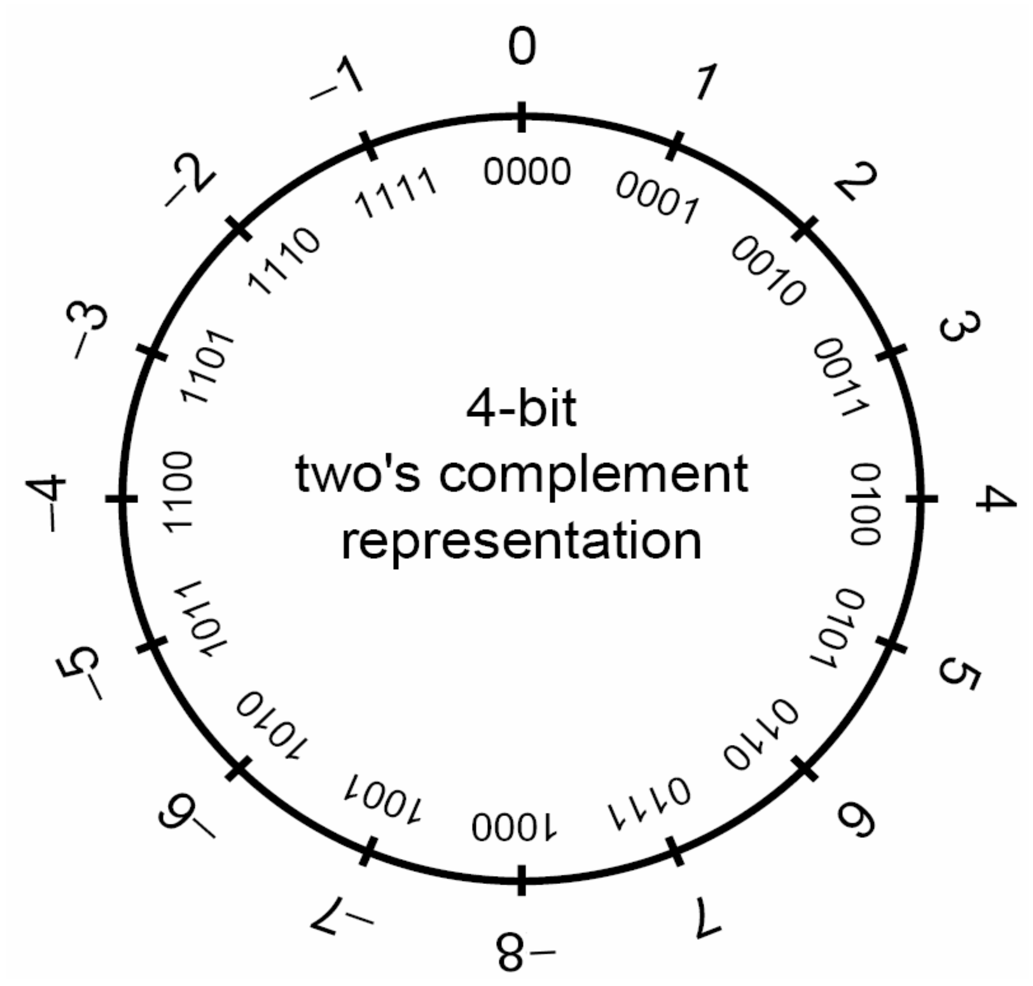
For each signed type there is an equivalent unsigned type.

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

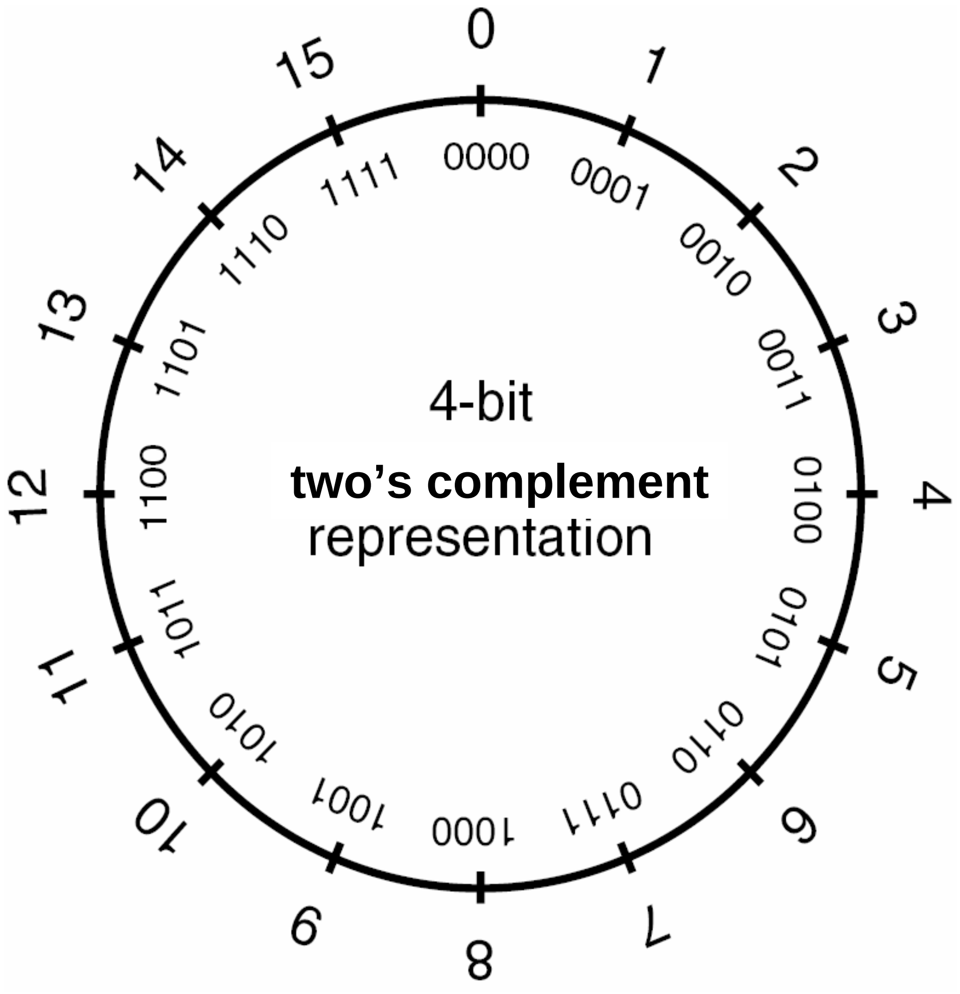


Unsigned Integers

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.

Unsigned Integer Representation



CERT

Standard Integer Types

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

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

Other C99 Integer Types

The following types are used for special purposes

- `ptrdiff_t` is the signed integer type of the result of subtracting two pointer
- `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.

Platform-Specific Integer Types

Vendors often define platform-specific integer types.

The Microsoft Windows API defines a large number of integer types:

- **`__int8, __int16, __int32, __int64`**
- **`ATOM`**
- **`BOOLEAN, BOOL`**
- **`BYTE`**
- **`CHAR`**
- **`DWORD, DWORDLONG, DWORD32, DWORD64`**
- **`WORD`**
- **`INT, INT32, INT64`**
- **`LONG, LONGLONG, LONG32, LONG64`**
- **`Etc.`**

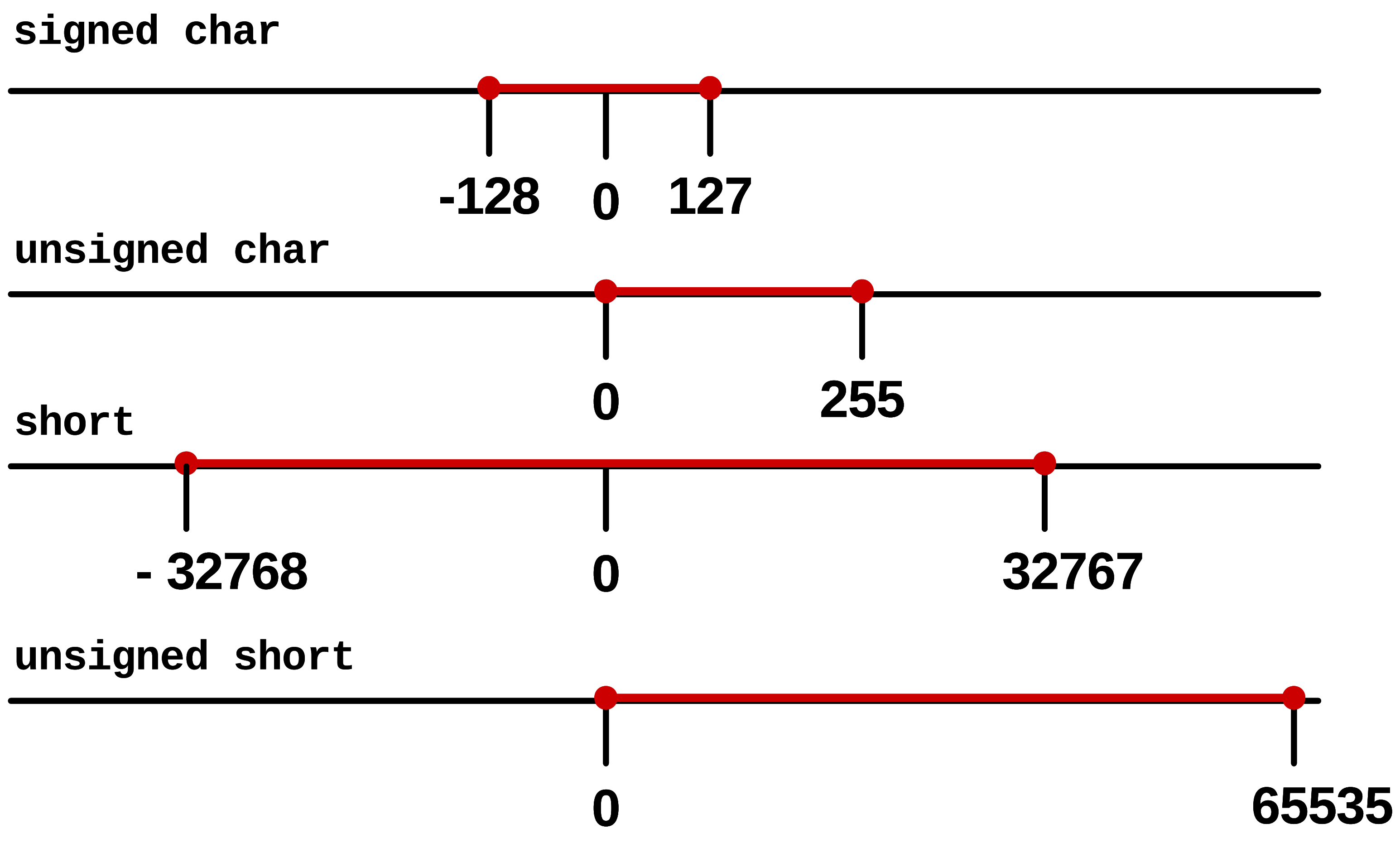
Integer Ranges

Minimum and maximum values for an integer type depend on

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

The C99 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 and 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 language ability to perform operations on mixed types.

C99 rules define how C compilers handle conversions:

- 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 promotions are applied as part of the usual arithmetic conversions.

Integer Promotion Example

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.

Implicit Conversions

```
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 Conversion Rank

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

Integer Conversion Rank Rules

No two signed integer types have the same rank, even if they have the same representation.

The rank of a signed integer type is > the rank of any signed integer type with less precision.

The rank of **`long long int`** is > the rank of **`long int`**, which is > the rank of **`int`**, which is > the rank of **`short int`**, which is > the rank of **`signed char`**.

The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type.

Usual Arithmetic Conversions

If both operands have the same type, no conversion is needed.

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.

If the operand that has unsigned integer type has rank >= 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.

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.

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

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

Signed Integer Conversion Example

```
unsigned int l = ULONG_MAX;
char c = -1;
if (c == l) {
  printf("-1 = 4,294,967,295?\n");
}
```

The value of **`c`** is compared to the value of **`l`**.

Because of integer promotions, **`c`** is converted to an unsigned integer with a value of **`0xFFFFFFFF`** or 4,294,967,295.

Signed/Unsigned Characters

The type **`char`** can be signed or unsigned.

When a **`signed char`** with its high bit set is saved in an integer, the result is a negative number.

Use **`unsigned char`** for buffers, pointers, and casts when dealing with character data that may have values greater than 127 (**`0x7f`**).

Integer Section Agenda

Representation

Types

Conversions

Error conditions

Operations

Integer Error Conditions

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

- overflow
- sign error
- truncation

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

```
1. int i;
2. unsigned int j;

3. i = INT_MAX;  // 2,147,483,647
4. i++;
5. printf("i = %d\n", i);

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

```
i=-2,147,483,648
```

```
j = 0
```

Overflow Examples 2

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

```
i = 2,147,483,647
```

```
j = 0;
j--;
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, c3;
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

Sign Errors

Can occur when

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

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.

Add Instruction

IA-32 instruction set includes an **`add`** instruction that takes the form

```
add destination, source
```

Adds the 1st (destination) op to the 2nd (source) op

- Stores the result in the destination operand
- Destination operand can be a register or memory location
- Source operand can be an immediate, register, or memory location

Signed and unsigned overflow conditions are detected and reported.

Add Instruction Example

The instruction

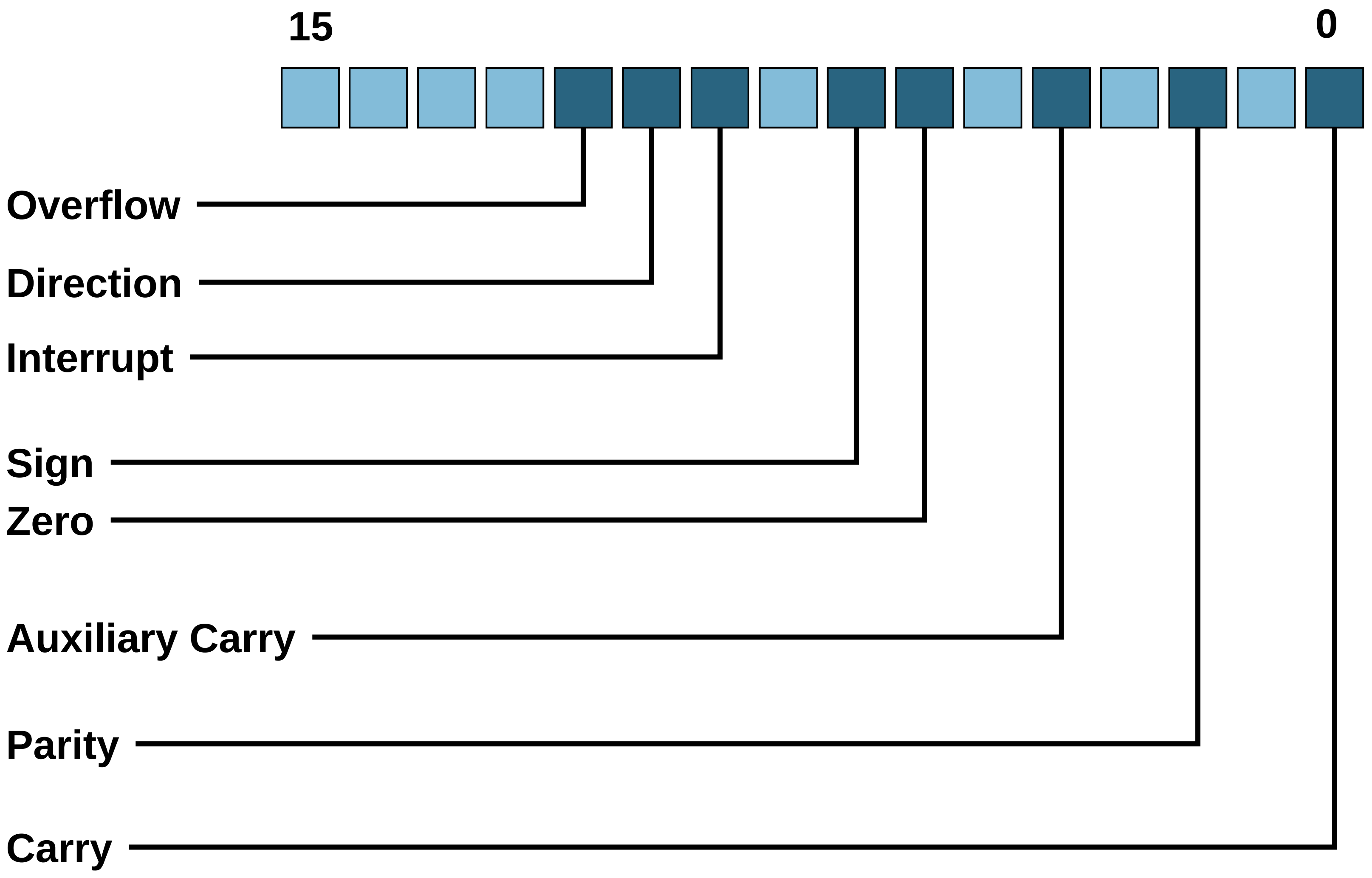
```
add  eax,  ebx
```

- adds the 32-bit **ebx** register to the 32-bit **eax** register
- leaves the sum in the **eax** register

The **add** instruction sets flags in the flags register

- overflow flag indicates signed arithmetic overflow
- carry flag indicates unsigned arithmetic overflow

Layout of the Flags Register



Interpreting Flags

There are no distinctions between the addition of signed and unsigned integers at the machine level.

Overflow and carry flags must be interpreted in context.

Adding signed and unsigned int

Both **`signed int`** and **`unsigned int`** values are added as follows:

```
si1 + si2

7. mov         eax, dword ptr [ui1]

8. add         eax, dword ptr [ui2]
```

Adding signed long long int

The **add** instruction adds the low-order 32 bits

```
sll1 + sll2
 9. mov         eax, dword ptr [sll1]
10. add         eax, dword ptr [sll2]
11. mov         ecx, dword ptr [ebp-98h]
12. adc         ecx, dword ptr [ebp-0A8h]
```

The **adc** instruction adds the high-order 32 bits and the value of the carry bit

Unsigned Overflow Detection

The carry flag denotes an unsigned arithmetic overflow.

Unsigned overflows can be detected using the

- `jc` instruction (jump if carry)
- `jnc` instruction (jump if not carry)

Conditional jump instructions are placed after the

- **add** instruction in the 32-bit case
- **adc** instruction in the 64-bit case

Signed Overflow Detection

The overflow flag denotes a signed arithmetic overflow.

Signed overflows can be detected using the

- `jo` instruction (jump if overflow)
- `jno` instruction (jump if not overflow)

Conditional jump instructions are placed after the

- **add** instruction in the 32-bit case
- **adc** instruction in the 64-bit case

Integer Subtraction

The IA-32 instruction set includes

- **sub** (subtract)
- **sbb** (subtract with borrow)

The **sub** and **sbb** instructions set the overflow and carry flags to indicate an overflow in the signed or unsigned result.

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.

Signed/Unsigned Examples

The max value 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 value 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

The `mul` instruction

- performs an unsigned multiplication of the 1st (destination) operand and the 2nd (source) operand
- stores the result in the destination operand

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

Signed/Unsigned `int` Multiplication

```
si_product = si1 * si2;

ui_product = ui1 * ui2;

 9. mov  eax, dword ptr [ui1]

10. imul eax, dword ptr [ui2]

11. mov  dword ptr [ui_product], eax
```

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 C99, 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 IA-32 instruction set includes the **`div`** and **`idiv`** instructions.

The **`div`** instruction

- divides the (unsigned) integer value in the **`ax`**, **`dx:ax`**, or **`edx:eax`** registers (dividend) by the source operand (divisor)
- stores the result in the **`ax`** (**`ah:al`**), **`dx:ax`**, or **`edx:eax`** registers

The **`idiv`** instruction performs the same operations on (signed) values.

Signed Integer Division

```
si_quotient = si_dividend / si_divisor;
```

```
1. mov  eax, dword ptr [si_dividend]
2. cdq
3. idiv eax, dword ptr [si_divisor]
4. mov  dword ptr [si_quotient], eax
```

The **cdq** instruction copies the sign (bit 31) of the value in the **eax** register into every bit position in the **edx** register.

NOTE: Assembly code generated by Visual C++

Unsigned Integer Division

```
ui_quotient = ui1_dividend / ui_divisor;
```

```
5. mov eax, dword ptr [ui_dividend]
6. xor edx, edx
7. div eax, dword ptr [ui_divisor]
8. mov dword ptr [ui_quotient], eax
```

NOTE: Assembly code generated by Visual C++

Error Detection

The Intel division instructions `div` and `idiv` 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
- extensions to the C language that enable C programs to handle Win32 structured exceptions

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

Structured Exception Handling in C

```
int x, y;

__try {
  x = INT_MIN;
  y = -1;
  x = x / y;
}
__except (GetExceptionCode() ==
           EXCEPTION_INT_OVERFLOW ?
           EXCEPTION_EXECUTE_HANDLER :
           EXCEPTION_CONTINUE_SEARCH) {
  printf("Integer overflow during division.\n");
}
```

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.

Linux Error Handling 1

In the Linux environment, hardware exceptions such as division errors are managed using signals.

If the source operand (divisor) is zero or if the quotient is too large for the designated register, a **`SIGFPE`** (floating point exception) is generated.

To prevent abnormal termination of the program, a signal handler can be installed.

```
signal(SIGFPE, Sint::divide_error);
```

Linux Error Handling 2

The **`signal()`** call accepts two parameters:

- signal number
- address of signal handler

Because the return address points to the faulting instruction, if the signal handler simply returns, the instruction and the signal handler will be alternately called in an infinite loop.

To solve this problem, the signal handler throws a C++ exception that can then be caught by the calling function.

Signal Handler

```
static void divide_error(int val) {
  throw

     SintException(ARITHMETIC_OVERFLOW);
}
```

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

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 _tmain(int argc, _TCHAR* 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

Memory Allocation Example

Integer overflow can occur in **`calloc()`** and other memory allocation functions when computing the size of a memory region.

A buffer smaller than the requested size is returned, possibly resulting in a subsequent buffer overflow.

The following code fragments may lead to vulnerabilities:

- C: **`p = calloc(sizeof(element_t), count);`**
- C++: **`p = new ElementType[count];`**

Memory Allocation

The **`calloc()`** library call accepts two arguments:

- the storage size of the element type
- the number of elements

The element type size is not specified explicitly in the case of the **new** operator in C++.

To compute the size of the memory required, the storage size is multiplied by the number of elements.

Overflow Condition

If the result cannot be represented in a signed integer, the allocation routine can appear to succeed but allocate an area that is too small.

The application can write beyond the end of the allocated buffer, resulting in a heap-based buffer overflow.

Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional

Sign Error Example 1

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
  int len;
  char buf[BUFF_SIZE];
  len = atoi(argv[1]);
  if (len < BUFF_SIZE){
    memcpy(buf, argv[2], len);
  }
}
```

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`

Sign Errors Example 2

The negative length is interpreted as a large, positive integer with the resulting buffer overflow.

This vulnerability can be prevented by restricting the integer `len` to a valid value.

- more effective range check that guarantees `len` is greater than 0 but less than `BUFF_SIZE`
- declare as an unsigned integer
  - eliminates the conversion from a signed to unsigned type in the call to `memcpy()`
  - prevents the sign error from occurring

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);
    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

Vulnerability 1

`cbBuf` is temporarily stored in the unsigned short `bufSize`.

The maximum size of an **`unsigned short`** for both GCC and the Visual C++ compiler on IA-32 is 65,535.

The maximum value for a **`signed long`** on the same platform is 2,147,483,647.

A truncation error will occur on line 2 for any values of `cbBuf` between 65,535 and 2,147,483,647.

Vulnerability 2

This would only be an error and not a vulnerability if **`bufSize`** were used for both the calls to **`malloc()`** and **`memcpy()`**.

Because **`bufSize`** is used to allocate the size of the buffer and **`cbBuf`** is used as the size on the call to **`memcpy()`**, it is possible to overflow **`buf`** by anywhere from 1 to 2,147,418,112 (2,147,483,647 - 65,535) bytes.

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

Vulnerability

There is a vulnerability resulting from incorrect range checking of **pos**.

- Because **pos** is declared as a signed integer, both positive and negative values can be passed to the function.
- An out-of-range positive value would be caught but a negative value would not.

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 and C++ are not nearly as good at enforcing type safety.

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 Explained

Declaring `len` to be an unsigned integer is insufficient for range restriction because it only restricts the range from `0..MAX_INT`.

Checking upper and lower bounds ensures no out-of-range values are passed to **`memcpy()`**.

Using both the implicit and explicit checks may be redundant but is recommended as “healthy paranoia.”

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

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Strong Typing

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.

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;
```

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`**.

`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

Strong Typing Example

Declare an integer to store the temperature of water using the Fahrenheit scale:

```
unsigned char waterTemperature;
```

`waterTemperature` is an unsigned 8-bit value in the range 1-255.

`unsigned char`

- sufficient to represent liquid water temperatures, which range from 32 degrees Fahrenheit (freezing) to 212 degrees Fahrenheit (the boiling point)
- does not prevent overflow
- allows invalid values (e.g., 1-31 and 213-255)

Abstract Data Type

One solution is to create an abstract data type in which `waterTemperature` is private and cannot be directly accessed by the user.

A user of this data abstraction can only access, update, or operate on this value through public method calls.

These methods must provide type safety by ensuring that the value of `waterTemperature` does not leave the valid range.

If implemented properly, there is no possibility of an integer type range error occurring.

Mitigation Section Agenda

Type range checking

Strong typing

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

Postcondition

For unsigned integers, if the sum is smaller than either operand, an overflow has occurred.

For signed integers, let `sum = lhs + rhs`.

- If `lhs` is non-negative and `sum < rhs`, an overflow has occurred.
- If `lhs` is negative and `sum > rhs`, an overflow has occurred.
- In all other cases, the addition operation succeeds.

Adding Signed Integers

Function from the **gcc** runtime system used to detect errors resulting from the addition of signed 16-bit integers

```
Wtype __addvsi3 (Wtype a, Wtype b) {
  const Wtype w = a + b;
  if (b >= 0 ? w < a : w > a)
    abort ();
  return w;
}
```

The addition is performed and the sum is compared to the operands to determine if an error occurred

abort() is called if
- **b** is non-negative and **w < a**
- **b** is negative and **w > a**

Mitigation Section Agenda

Type range checking

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Safe Integer Operations 1

Integer operations can result in error conditions and possible lost data.

The first line of defense against integer vulnerabilities should be range checking.

- explicitly
- implicitly - through strong typing

It is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program.

Safe Integer Operations 2

An alternative or ancillary approach is to protect each operation.

This approach can be labor intensive and expensive to perform.

Use a safe integer library for all operations on integers where one or more of the inputs could be influenced by an untrusted source.

Safe Integer Solutions

C language compatible library

- written by Michael Howard at Microsoft
- detects integer overflow conditions using IA-32 specific mechanisms

Unsigned Add Function

```
int bool UAdd(size_t a, size_t b, size_t *r) {
  __asm {
    mov eax, dword ptr [a]
    add eax, dword ptr [b]
    mov ecx, dword ptr [r]
    mov dword ptr [ecx], eax
    jc  short j1
    mov al, 1 // 1 is success
    jmp short j2
j1:
    xor al, al // 0 is failure
j2:
  };
}
```

Unsigned Add Function Example

```
int main(int argc, char *const *argv) {
  unsigned int total;
  if (UAdd(strlen(argv[1]), 1, &total) &&
     UAdd(total, strlen(argv[2]), &total)) {
    char *buff = (char *)malloc(total);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
  else {
    abort();
  }
}
```

The length of the combined strings is calculated using **`UAdd()`** with appropriate checks for error conditions.

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[]`**.

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.

Addition

Addition of unsigned integers can result in an integer overflow if the sum of the left-hand side (LHS) and right-hand side (RHS) of an addition operation is greater than

- **`UINT_MAX`** for addition of **`unsigned int`** type
- **`ULLONG_MAX`** for addition of **`unsigned long long`** type

Precondition Example

Overflow occurs when **`A`** and **`B`** are **`unsigned int`** and

```
    A + B > UINT_MAX
```

To prevent the addition from overflowing the **`operator+`** tests that

```
    A > UINT_MAX – B
```

Safe Integer Solutions Compared

SafeInt library has several advantages:

- more portable than safe arithmetic operations that depend on assembly language instructions
- more usable
  - operators can be used inline in expressions
  - SafeInt uses C++ exception handling
- better performance (with optimized code)

However, SafeInt fails to provide correct integer promotion behavior.

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

Mitigation Section Agenda

Type range checking

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Testing 1

Input validation does not guarantee that subsequent operations on integers will not result in an overflow or other error condition.

Testing does not provide any guarantees either.

- It is impossible to cover all ranges of possible inputs on anything but the most trivial programs.
- If applied correctly, testing can increase confidence that the code is secure.

Testing 2

Integer vulnerability tests should include boundary conditions for all integer variables.

- If type range checks are inserted in the code, test that they function correctly for upper and lower bounds.
- If boundary tests have not been included, test for minimum and maximum integer values for the various integer sizes used.

Use white box testing to determine the types of integer variables.

If source code is not available, run tests with the various maximum and minimum values for each type.

Source Code Audit

Source code should be audited or inspected for possible integer range errors.

When auditing check that

- integer type ranges are properly checked
- input values are restricted to a valid range based on their intended use

Integers that do not require negative values are

- declared as unsigned
- properly range-checked for upper and lower bounds

Operations on integers originating from untrusted sources are performed using a safe integer library.

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

Summary

Notable Vulnerabilities

Integer Overflow In XDR Library

- SunRPC xdr_array buffer overflow
- http://xforce.iss.net/xforce/xfdb/9170

Windows DirectX MIDI Library

- eEye Digital Security advisory AD20030723
- http://www.eeye.com/html/Research/Advisories/AD20030723.html

Bash

- CERT Advisory CA-1996-22
- http://www.cert.org/advisories/CA-1996-22.html

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

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)

Questions about Integers

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**

CERT

Backup Slides

sub Instruction

Subtracts the 2nd (source) operand from the 1st (destination) operand.

Stores the result in the destination operand.

The destination operand can be a

- register
- memory location

The source operand can be a(n)

- immediate
- register
- memory location

sbb **Instruction**

The **sbb** instruction is executed as part of a multibyte or multiword subtraction.

The **sbb** instruction adds the 2nd (source) operand and the carry flag and subtracts the result from the 1st (destination) operand.

The result of the subtraction is stored in the destination operand.

The carry flag represents a borrow from a previous subtraction.

signed long long int Sub

`sll1 - sll2`

The **sub** instruction subtracts the low-order 32 bits

```
1. mov eax, dword ptr [sll1]
2. sub eax, dword ptr [sll2]
3. mov ecx, dword ptr [ebp-0E0h]
4. sbb ecx, dword ptr [ebp-0F0h]
```

The **sbb** instruction subtracts the low-order 32 bits

NOTE: Assembly code generated by Visual C++ for Windows 2000

Introductory Example

Accepts two string arguments and calculates their combined length (plus an extra byte for the terminating null character)

```
int main(int argc, char *const *argv) {
   unsigned short int total;
   total = strlen(argv[1]) +
               strlen(argv[2]) + 1;
   char *buff = (char *) malloc(total);
   strcpy(buff, argv[1]);
   strcat(buff, argv[2]);
}
```

Memory is allocated to store both strings

The 1st argument is copied into the buffer and the 2nd argument is concatenated to the end of the 1st argument

Vulnerability

An attacker can supply arguments such that the sum of the lengths of the strings cannot be represented by the **`unsigned short int total`**.

The **`strlen()`** function returns a result of type **`size_t`**, an **`unsigned long int`** on IA-32.

- As a result, the sum of the lengths + 1 is an **`unsigned long int`**.
- This value must be truncated to assign to the **`unsigned short int total`**.

If the value is truncated, **`malloc()`** allocates insufficient memory and **`strcpy()`** and **`strcat()`** will overflow the dynamically allocated memory.