Chapter 5

Integer Security

Everything good is the transmutation of something evil:
every god has a devil for a father.

—Friedrich Nietzsche. *Sämtliche Werke: Kritische Studienausgabe*, vol. 10, selection 5[1], number 68

Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs. This is primarily because boundary conditions for integers, unlike other boundary conditions in software engineering, have been intentionally ignored. Most programmers emerging from colleges and universities understand that integers have fixed limits, but because these limits were either deemed sufficient, or because testing the results of each arithmetic operation was considered prohibitively expensive, violating integer boundary conditions has gone almost entirely unchecked in commercial software.

Security changes everything. It is no longer acceptable to assume a program will operate normally given a range of expected inputs when an attacker is looking for input values that produce an abnormal effect. Digital integer representations are, of course, imperfect. A software vulnerability may result when a program evaluates an integer to an unexpected value (that is, a value other than the one obtained with pencil and paper) and then uses the value as an array index, size, or loop counter.

Because integer range checking has not been systematically applied in the development of most C and C++ software systems, security flaws involving

```
1. int main(int argc, char *const *argv) {
2.   unsigned short int total;
3.   total = strlen(argv[1])+strlen(argv[2])+1;
4.   char *buff = (char *)malloc(total);
5.   strcpy(buff, argv[1]);
6.   strcat(buff, argv[2]);
7. }
```

Figure 5–1. Vulnerable program

integers are certain to exist, and some portion of these are likely to be vulnerabilities.

Figure 5–1 contains an example of a vulnerable program. The vulnerability results from a failure in how integer operations are managed. If you are unsure of why this program is vulnerable or how this vulnerability can be exploited to run arbitrary code, you should read the remainder of this chapter. Because integer vulnerabilities result from limitations in how they're represented, integer representation is examined first.

5.1 Integers

This section describes integer representations, types, and ranges. If you are already familiar with machine-level representation and manipulation of integer values, you can ignore this section and proceed to Section 5.2.

Integer Representation

The major consideration in the digital representation of integers is negative integers. Representation methods include *signed-magnitude*, *one's complement*, and *two's complement* [Shiflet 02].

Signed-magnitude representation uses the high-order bit to indicate the sign: 0 for positive, 1 for negative. The remaining low-order bits indicate the magnitude of the value. For example, the binary value 0010 1001 shown in Figure 5–2 represents +41 when the most significant bit is cleared and –41 when it is set.

One's complement representation replaced signed magnitude because the circuitry required to implement signed-magnitude arithmetic was too complicated. Negative numbers are represented in one's complement form by comple-

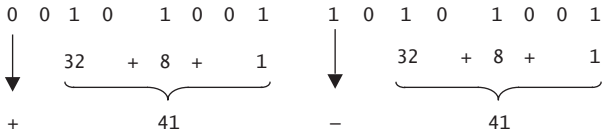

Figure 5–2. Signed-magnitude representation of +41 and –41

menting (taking the opposite value of) each bit, as shown in Figure 5–3 (a). Each 1 is replaced with a 0 and each 0 is replaced with a 1. Even the sign bit is reversed.

Both signed-magnitude and one's complement representations have two representations for zero, which makes programming awkward: tests are required for +0 and –0.

Two's complement representation is the dominant representation and is used almost universally in modern computers. The two's complement form of a negative integer is created by adding one to the one's complement representation, as shown in Figure 5–3 (b). Two's complement representation has a single (positive) value for zero. The sign is still represented by the most significant bit, and the notation for positive integers is identical to their signed-magnitude representations.

Integer Types

C and C++ provide a variety of integer types to allow a close correspondence with the underlying machine architecture. The integer types categories are shown in Table 5–1.

There are two broad categories of integer types: *standard* and *extended*. The standard integer types include all the well-known integer types that have existed from the early days of K&R C. Extended integer types are defined in the C99 standard to specify integer types with fixed constraints.

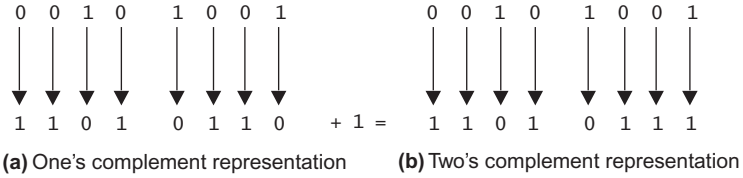

Figure 5–3. Integer representations

Table 5–1. Integer Types

	Standard	Extended
Signed	Standard signed integer types	Extended signed integer types
Unsigned	Standard unsigned integer types	Extended unsigned integer types

Signed and Unsigned Types. Integers in C and C++ are either *signed* or *unsigned*. Both standard and extended integer types include signed and unsigned types; for each signed type there is an equivalent unsigned type. Signed integers are used to represent positive and negative values, the range of which depends on the number of bits allocated to the type and the encoding technique. On a computer using two's complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1} - 1$. When one's complement or sign-magnitude representations are used, the lower bound is $-2^{n-1} + 1$, while the upper bound remains the same.

Figure 5–4 shows the two's complement representation for 4-bit signed integers. Note that incrementing a signed integer at its maximum value (7) results in the minimum value for that type (–8).

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

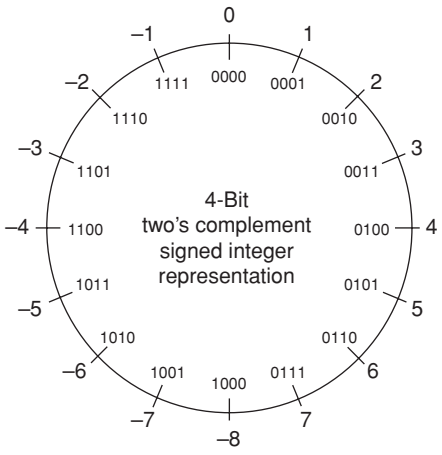

Figure 5–4. Signed integer representation (4-bit, two's complement)

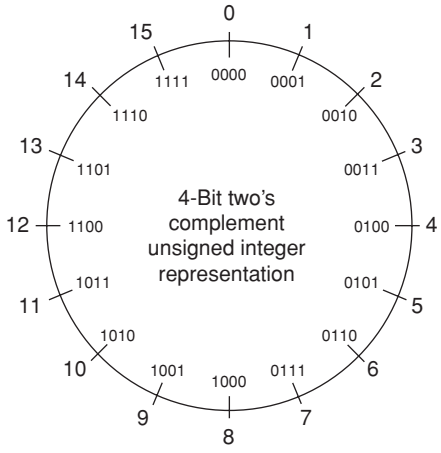

Figure 5–5. Unsigned integer representation (4-bit, two's complement)

the number of bits used to represent the unsigned type. For each signed integer type, there is a corresponding unsigned integer type.

Figure 5–5 shows the two's complement representation for 4-bit unsigned integers. Again, note that incrementing a signed integer at its maximum value (15) results in the minimum value for that type (0).

Standard and Extended Types. Standard integers include the following types, in increasing length order (for example, `long long int` cannot be shorter than `long int`):

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

Extended integer types are implementation defined and include the following types:

- `int#_t`, `uint#_t`, where # represents an exact width (for example, `int8_t`, `uint24_t`)
- `int_least#_t`, `uint_least#_t`, where # represents a width of at least that value (for example, `int_least32_t`, `uint_least16_t`)

- `int_fast#_t`, `uint_fast#_t`, where # represents a width of at least that value for fastest integer types (for example, `int_fast16_t`, `uint_fast64_t`)
- `intptr_t`, `uintptr_t` are integer types wide enough to hold pointers to objects
- `intmax_t`, `uintmax_t` are integer types with the greatest width

Compilers that adhere to the C99 standard support *all* standard types and *most* extended types.

Other Standard Integer Types. In addition to the standard and extended integer types, the C99 specification also defines a number of standard types that are used for special purposes. For example, the following types are defined in the standard header `<stddef.h>`:

- `ptrdiff_t` is the signed integer type of the result of subtracting two pointers
- `size_t` is the unsigned integer type of the 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

You should use these types where appropriate, but understand how they are defined, particularly when combined in operations with differently typed integers.

Platform-Specific Integer Types. In addition to the integer types defined in the C99 standard types, vendors often define platform-specific integer types. The Microsoft Windows API defines a large number of integer types, including: `__int8`, `__int16`, `__int32`, `__int64`, `ATOM`, `BOOLEAN`, `BOOL`, `BYTE`, `CHAR`, `DWORD`, `DWORDLONG`, `DWORD32`, `DWORD64`, `WORD`, `INT`, `INT32`, `INT64`, `LONG`, `LONGLONG`, `LONG32`, `LONG64`, and so forth.[1]

As a Windows programmer you will frequently come across these types. Again, it is okay to use these types, but you should understand how they are defined.

1. See http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/_langref_Data_Type_Ranges.asp and http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_data_types.asp for more information on these data types.

Integer Ranges

The minimum and maximum values for an integer type depend on the type's representation, signedness, and number of allocated bits. Figure 5–6 shows the ranges of some integers that use two's complement representation.

The C99 standard sets minimum requirements for these ranges. Table 5–2 shows the maximum and minimum extents of integer types as required by C99 and as implemented by the Visual C++ .NET and `gcc` compilers on IA-32. Integer ranges are compiler dependent but greatly influenced by the target machine architecture, as demonstrated by the use of a single column in Table 5–2 to represent integer sizes for both compilers.

Comair

A good example of a software failure resulting from integer limits came on Saturday, December 25, 2004, when Comair halted all operations and grounded 1,100 flights after a crash of its flight crew scheduling software.

The software failure was due to a 16-bit counter that limits the number of changes to 32,768 in any given month. Storms earlier in the month caused many crew reassignments, and the 16-bit value was exceeded.

Compiler- and platform-specific integral limits are documented in the `limits.h` header file. Familiarize yourself with these limits, but remember that these values are platform specific. For portability, use the named constants and not the actual values in your code.

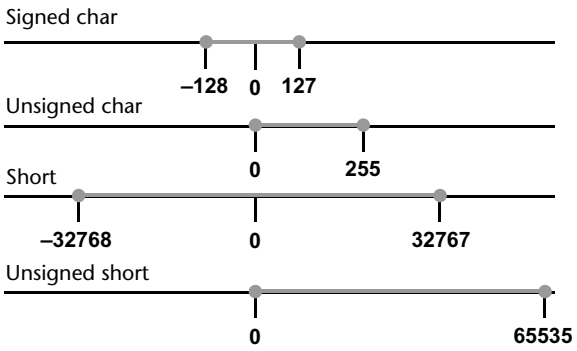

Figure 5–6. Integer ranges (not to scale)

Table 5–2. Maximum and Minimum Extents of Integer Types

Constant	C99 minimum value	Visual C++ .NET and GNU CC on Intel x86	Description
CHAR_BIT	8	8	Number of bits for smallest object that is not a bit-field (byte)
SCHAR_MIN	–127 // $-(2^7 - 1)$	–128	Minimum value for an object of type `signed char`
SCHAR_MAX	+127 // $2^7 - 1$	+127	Maximum value for an object of type `signed char`
UCHAR_MAX	255 // $2^8 - 1$	255	Maximum value for an object of type `unsigned char`
SHRT_MIN	–32,767 // $-(2^{15} - 1)$	–32,768	Minimum value for an object of type `short int`
SHRT_MAX	+32,767 // $2^{15} - 1$	+32,767	Maximum value for an object of type `short int`
USHRT_MAX	65,535 // $2^{16} - 1$	65,535	Maximum value for an object of type `unsigned short int`
INT_MIN	–32,767 // $-(2^{15} - 1)$	–2,147,483,648	Minimum value for an object of type `int`
INT_MAX	+32,767 // $2^{15} - 1$	+2,147,483,647	Maximum value for an object of type `int`
UINT_MAX	65,535 // $2^{16} - 1$	4,294,967,295	Maximum value for an object of type `unsigned int`
LONG_MIN	–2147483647 // $-(2^{31} - 1)$	–2,147,483,648	Minimum value for an object of type `long int`
LONG_MAX	+2,147,483,647 // $2^{31} - 1$	+2,147,483,647	Maximum value for an object of type `long int`
ULONG_MAX	4,294,967,295 // $2^{32} - 1$	4,294,967,295	Maximum value for an object of type `unsigned long int`
LLONG_MIN[a]	–9223372036854775807 // $-(2^{63} - 1)$	–9223372036854775808	Minimum value for an object of type `long long int`
LLONG_MAX[b]	+9223372036854775807 // $2^{63} - 1$	+9223372036854775807	Maximum value for an object of type `long long int`
ULLONG_MAX[c]	18446744073709551615 // $2^{64} - 1$	18446744073709551615	Maximum value for an object of type `unsigned long long int`

a. The default constant for `gcc` is `LONG_LONG_MIN`. For Visual C++ it is `I64_MIN`.
b. The default constant for `gcc` is `LONG_LONG_MAX`. For Visual C++ it is `I64_MAX`.
c. The default constant for `gcc` is `ULONG_LONG_MAX`. For Visual C++ it is `_UI64_MAX`.

5.2 Integer Conversions

Type conversions occur explicitly in C and C++ as the result of a cast or implicitly as required by an operation. While conversions are generally required for the correct execution of a program, they can also lead to lost or misinterpreted data. This section describes how and when conversions are performed and identifies their pitfalls.

Implicit conversions are, in part, a consequence of the C language ability to perform operations on mixed types. For example, most C programmers would not think twice before adding an `unsigned char` to a `signed char` and storing the result in a `short int`. This is because the C compiler generates the code required to perform the required conversions implicitly.

The C99 standard rules define how C compilers handle conversions. These rules, which are described in the following sections, include *integer promotions*, *integer conversion rank*, and *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 an `int`; otherwise, it is converted to an `unsigned int`.

Integer promotions are applied as part of the *usual arithmetic conversions* (discussed later in this section) to certain argument expressions, operands of the unary +, –, and ~ operators, and operands of the shift operators. The following code fragment illustrates the use of integer promotions:

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

Integer promotions require the promotion value of each variable (`c1` and `c2`) to `int` size. The two `int`s are added and the sum truncated to fit into the `char` type.

Integer promotions are performed to avoid arithmetic errors resulting from the overflow of intermediate values. On line 5 of Figure 5–7, the value of `c1` is added to the value of `c2`. The sum of these values is then added to the value of `c3` (according to operator precedence rules). The addition of `c1` and `c2` would result in an overflow of the `signed char` type because the result of the operation exceeds the maximum size of `signed char`. Because of integer promotions, however, `c1`, `c2`, and `c3` are each converted to integers and the overall expression is successfully evaluated. The resulting value is then truncated and stored

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

Figure 5–7. Preventing arithmetic errors with implicit conversions

in `cresult`. Because the result is in the range of the `signed char` type, the truncation does not result in lost data.

Integer Conversion Rank

Every integer type has an *integer conversion rank* that determines how conversions are performed. The following rules for determining integer conversion rank are defined in C99.

- No two different signed integer types have the same rank, even if they have the same representation.
- The rank of a signed integer type is greater than the rank of any signed integer type with less precision.
- The rank of `long long int` is greater than the rank of `long int`, which is greater than the rank of `int`, which is greater than the rank of `short int`, which is greater than the rank of `signed char`.
- The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type, if any.
- The rank of any standard integer type is greater than the rank of any extended integer type with the same width.
- The rank of `char` is equal to the rank of `signed char` and `unsigned char`.
- The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation defined but still subject to the other rules for determining the integer conversion rank.
- For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.

The integer conversion rank is used in the usual arithmetic conversions to determine what conversions need to take place to support an operation on mixed integer types.

Conversions from Unsigned Integer Types

Conversions occur between signed and unsigned integer types of any size and can result in lost or misinterpreted data when a value cannot be represented in the new type.

Conversions of smaller unsigned integer types to larger unsigned integer types are always safe and typically accomplished by zero-extending the value.

When a large unsigned integer is converted to a smaller unsigned integer type, the larger value is truncated and the low-order bits are preserved. When a large unsigned integer is converted to a smaller signed integer type, the value is also truncated, and the high-order bit becomes the sign bit. In both cases, data may be lost if the value cannot be represented in the new type.

When unsigned integer types are converted to the corresponding signed integer type (for example, an `unsigned char` to a `char`), the bit pattern is preserved, so no data is lost. The high-order bit, however, becomes the sign bit. If the sign bit is set, both the sign and magnitude of the value changes.

Table 5–3 summarizes conversions from unsigned integer types. Conversions that can result in lost data are light gray, and ones that can result in the incorrect interpretation of data are dark gray.

Conversions from Signed Integer Types

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 conversion is typically made by sign-extending the signed integer. A signed integer is converted to a shorter signed integer by truncating the high-order bits.

When signed integer types are converted to unsigned, there is no lost data because the bit pattern is preserved. However, the 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, signed integer. In line 3 of Figure 5–8, the value of `c` is compared to the value of `1`. Because of integer promotions, `c` is converted to an unsigned integer with a value of `0xFFFFFFFF` or 4,294,967,295.

Table 5–4 summarizes conversions from signed integer types. Conversions that can result in lost data are light gray, and ones that can result in the incorrect interpretation of data are dark gray.

Table 5–3. Conversions from Unsigned Integer Types

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

```
1. unsigned int l = ULONG_MAX;
2. char c = -1;

3. if (c == l) {
4.   printf("Why is -1 = 4,294,967,295???\n");
5. }
```

Figure 5–8. Integer promotion error

Signed or Unsigned Characters

An additional complication in C and C++ conversion is that the character type `char` can be signed or unsigned (depending on the compiler and machine). When a `signed char` with its high-bit set is saved in an integer, the result is a negative number. In some cases this can lead to an exploitable vulnerability, so in general it is best to use `unsigned char` instead of `char` or `signed char` for

Table 5–4. Conversions from Signed Integer Types

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 bit pattern; high-order bit loses function as sign bit

buffers, pointers, and casts when dealing with character data that may have values greater than 127 (`0x7f`). For example, when processing e-mail messages, certain versions of `sendmail` create tokens from address elements (user, host, domain). The code that performs this function (`prescan()` in `parseaddr.c`) contains logic to check that the tokens are not malformed or overly long. In certain cases, a variable in `prescan()` is set to the special control value –1, which may alter the program logic to skip the length checks. Using an e-mail message with a specially crafted address containing `0xFF`, an attacker can bypass the length checks and overwrite the saved instruction pointer on the stack. When `prescan()` evaluates a character with the value `0xFF` as an `int`, the value is interpreted as –1, causing the length checks to be skipped. This vulnerability is described in VU#897604 and elsewhere.[2]

2. See http://www.kb.cert.org/vuls/id/897604.

Usual Arithmetic Conversions

Many operators that accept arithmetic operands perform conversions using the *usual arithmetic conversions*. After integer promotions are performed on both operands, the following rules are applied to the promoted operands.

1. If both operands have the same type, no further conversion is needed.
2. If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
3. If the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.
4. If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.
5. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

Specific operations can add to or modify the semantics of the usual arithmetic operations.

5.3 Integer Error Conditions

Integer operations can resolve to unexpected values as a result of an *overflow condition*, *sign error*, or *truncation error*. This section explains what these conditions are and how they occur.

Integer Overflow

An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.[3] Integer overflows are closely related to the underlying representation.

3. Decreasing an integer beyond its minimum value is often referred to as an integer *underflow*, although technically this term refers to a floating point condition.

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. It is impossible to determine whether an overflow represents an error condition without understanding the context. For example, a signed overflow condition is not an error when adding two unsigned numbers.

Integer promotion rules dictate that integers smaller than size `int` are promoted to `int` or `unsigned int` before being operated on. This means that integers smaller than `int` are unlikely to overflow because the actual operations are performed on the promoted values. However, these operations may result in a truncation error if the value is assigned to a variable of the size of the original operands.

Figure 5–9 shows the consequences of overflows on signed and unsigned integers. The signed integer `i` is assigned its maximum value on line 3 of 2,147,483,647 and incremented on line 4. This operation results in an integer overflow, and `i` is assigned the value –2,147,483,648 (the minimum value for an `int`). The result of the operation (2,147,483,647 + 1 = –2,147,483,648) is clearly an arithmetic error. Integer overflows also occur when incrementing an unsigned integer already at its maximum value (lines 6–8), decrementing a signed integer already at its minimum value (lines 9–11), or decrementing an unsigned integer already at its minimum value (lines 12–14).

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

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

 6. j = UINT_MAX; // 4,294,967,295;
 7. j++;
 8. printf("j = %u\n", j); /* j = 0 */

 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 */
```

Figure 5–9. Signed and unsigned integer overflows

Although unsigned integer overflows appear to be an error condition, there is some debate about this. The following excerpt is from the C99 standard:

> A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo to the number that is one greater than the largest value that can be represented by the resulting type.

This is a good example of why C is considered a low-level language (that is, one that maps closely to the underlying hardware). Unfortunately, modulo behavior is nonintuitive and can lead to vulnerabilities. Because modulo behavior is sanctioned by the C99 specification, compiler developers are typically disinterested in generating code that detects unsigned integer overflows.[4]

Sign Errors

Sign errors occur when converting from signed to unsigned integer types. When a signed integer is converted to an unsigned integer of equal size, the bit pattern of the original integer is preserved. When a signed integer is converted to an unsigned integer of greater size, the value is first sign-extended and then converted. In both cases, the high-order bit loses its function as a sign bit. If the value of the signed integer is not negative, the value is unchanged. However, when the value of the signed integer is negative, the result is typically a large positive value as shown in Figure 5–10.

```
1.  int i = -3;
2.  unsigned short u;

3.  u = i;
4.  printf("u = %hu\n", u);  /* u = 65533 */
```

Figure 5–10. Sign error

4. The ISO/IEC 10967-1 standard for language-independent arithmetic (LIA-1) allows for the detection of *signed* integer overflows. However, an implementation that defines signed integer types as also being modulo does not need to detect integer overflow. It only needs to detect integer divide-by-zeros.

Why Are So Many Integers Signed?

Historically, most integer variables in C code are declared as signed rather than unsigned integers. On the surface, this seems odd. Most integer variables are used as sizes, counters, or indices that only require non-negative values. So why not declare them as unsigned integers that have greater range of positive values?

One possible explanation is the lack of an exception handling mechanism in C. As a result, C programmers have developed various mechanisms for returning status from application program interface (API) functions. Although C programmers could return status in a "call by reference" argument, the preferred mechanism is for the return value of the function to provide status. This allows a user of the function to test the return status directly in an `if-else` statement rather than by allocating a variable for the return status. This works fine when the function does not normally return a value, but what if the function already returns a value?

A common solution is to identify an invalid return value and use this to represent an error condition. As already noted, most applications of integers produce values in the non-negative range, so it is thereby possible to represent error conditions in a return value as a negative number. To store these values, however, a programmer must declare these values as signed instead of unsigned—possibly adding to the profusion of signed integers.

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. Normally, the low-order bits of the original value are preserved and the high-order bits are lost. When an unsigned value is converted to a signed value of the same length, the bit pattern is preserved. This causes the high-order bit to become a sign bit. As a result, values above the maximum value for the signed integer type are converted to negative numbers, as shown in Figure 5–11. This is a truncation error and not a sign error because the signed type has one less bit to represent the result.

5.4 Integer Operations

Integer operations can result in exceptional conditions in which the result of the operation is not the expected value. Unexpected integer values can cause unexpected programmatic behavior and, ultimately, security vulnerabilities.

Providing a detailed analysis of all integer operations is outside the scope of this book, so we will examine a few—addition, subtraction, multiplication, and division—in detail and provide references for other integer operations. It is generally safe to say that *most* integer operations can result in exceptional conditions, including all those listed here.

In this section, we describe the high-level semantics of integer operations (as defined by the C99 specification) and identify exceptional conditions that can result in vulnerabilities. We also examine *precondition*, *error detection*, and *postcondition* techniques for detecting or preventing exceptional conditions.

Preconditions can be used to determine that an error will occur before performing an operation. For example, a divide-by-zero error has the precondition that the divisor is equal to zero. As a result, these errors can be easily prevented by testing for the precondition.

The error detection technique requires determining if an error occurred performing an operation. Integer errors that result from limitations in how the machine represents integers are typically detected by the processor, which also provides the mechanisms for reporting these errors. The operating system and compiler, in turn, provide the mechanisms for C and C++ application-specific handling.

The postcondition technique performs the operation and then tests the resulting value to determine if it is within valid limits. This approach is ineffective if an exceptional condition can result in an apparently valid value.

In this section, we examine the mechanisms used by IA-32 to report integer errors and the mechanisms provided by Windows/Visual C++ and Linux/GCC to support application-specific handling.

```
1.  unsigned short int u = 32768;
2.  short int i;

3.  i = u;
4.  printf("i = %d\n", i); /* i = -32768 */

5.  u = 65535;
6.  i = u;
7.  printf("i = %d\n", i); /* i = -1 */
```

Figure 5–11. Integer truncation error

Machine-Level Integer Arithmetic

The C programming language is often described as a systems programming language because it is a relatively low-level language that deals with machine-level constructs such as characters, numbers, and addresses. Developers often joke that C is a structured assembly language. This is not true, but C would probably be more secure if it were. In practice, C programmers are often unaware of C semantics and how C source code is implemented in assembly language. There are many examples of compilers generating significantly different machine code for the same machine architecture. Examining the assembly language instructions generated for integer operations in C and C++ is useful in understanding what can go wrong, how these problems can be avoided, and the pros and cons of different avoidance strategies.

Addition

Addition can be used to add two arithmetic operands or a pointer and an integer. If both operands are of the arithmetic type, the usual arithmetic conversions are performed on them. The result of the binary + operator is the sum of the operands. Incrementing is equivalent to adding one. When an expression that has integer type is added to a pointer, the result is a pointer.

Integer addition can result in an overflow if the resulting value cannot be represented in the number of bits allocated to the representation of the integer. The actual number of bits allocated depends on the machine architecture and the operand types. For example, assume two operands of `char` type are added on a machine architecture in which a `char` is represented using 8 bits, a `short` with 16 bits, an `int` with 32 bits, and a `long long` with 64 bits. Because of integer promotions, both operands are converted from `signed char` to the `signed int` type before the addition is performed, but no overflow occurs because `SCHAR_MAX + SCHAR_MAX` is always less than `INT_MAX` and `SCHAR_MIN + SCHAR_MIN` is always greater than `INT_MIN`. However, no such guarantee exists if the two values being added are `int` or `long long` type because the integer promotions are not performed.

It is possible that `SCHAR_MAX + SCHAR_MAX` will exceed the size of a `signed char`, resulting in a *truncation error* during the *assignment* operation. However, no overflow occurs during the addition operation.

Error Detection. Signed and unsigned overflow conditions resulting from an addition operation are detected and reported on IA-32. The instruction set includes an `add` instruction that takes the form `add destination,source`. This instruction adds the first (destination) operand to the second (source) operand and stores the result in the destination operand. The destination operand can be

a register or memory location, and the source operand can be an immediate, register, or memory location. For example, `add ax, bx` adds the 16-bit `bx` register to the 16-bit `ax` register and leaves the sum in the `ax` register.

IA-32 instructions, including the `add` instruction, set flags in the flags register as shown in Figure 5–12. Flags related to integer overflow include the following:

- An overflow flag that indicates a signed arithmetic overflow
- A carry flag that indicates an unsigned arithmetic overflow

There are no distinctions between the addition of signed and unsigned integers at the machine level except the interpretation of the overflow and carry flags. Each addition operation can identify signed and unsigned overflow, as well as overflow of the low-order nibble.

Figure 5–13 shows the assembly language instructions generated by the Visual C++ .NET compiler for various addition operations. Lines 1–3 show the addition of two signed characters and lines 4–6 show the addition of two unsigned characters. Both operands in each case are moved into 32-bit registers (`eax` or `ecx`), which results in their promotion to 32 bit (the size of an `int`). The `signed char` values are sign extended to preserve the sign, while the `unsigned char` values are zero extended to avoid changing the magnitude.

Lines 7–8 show the addition of two unsigned `int` values. These are 32-bit values (hence the doubleword pointers). The exact same code is generated for signed integer values.

Lines 9–12 show the code generated for adding two signed `long long` integer operands. This operation needs to be performed with two separate addition

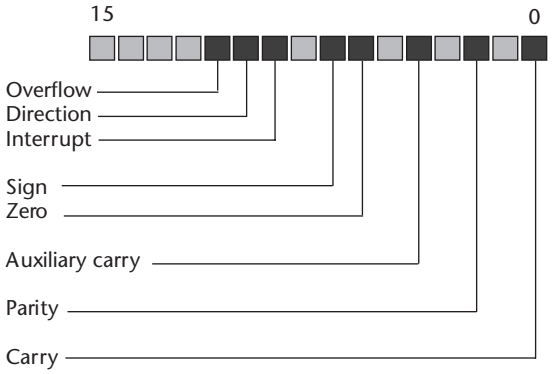

Figure 5–12. Layout of the flags register [Intel 04]

```
sc1 + sc2
 1. movsx       eax, byte ptr [sc1]
 2. movsx       ecx, byte ptr [sc2]
 3. add         eax, ecx


uc1 + uc2
 4. movzx       eax, byte ptr [uc1]
 5. movzx       ecx, byte ptr [uc2]
 6. add         eax, ecx

ui1 + ui2
 7. mov         eax, dword ptr [ui1]
 8. add         eax, dword ptr [ui2]

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]
```

Figure 5–13. Assembly code generated by Visual C++ for addition

instructions on IA-32. The `add` instruction adds the low-order 32 bits. The `adc` instruction adds the high-order 32 bits and the value of the carry bit. As noted earlier, the carry flag denotes an unsigned arithmetic overflow. However, for both 64-bit signed and unsigned integers, all the bits in the low-order doubleword are included in the magnitude. Therefore, if the carry bit is set, the value must be carried over to the high-order 32 doubleword.

Unsigned overflows can be detected using either the `jc` instruction (jump if carry) or the `jnc` instruction (jump if not carry). These conditional jump instructions are placed after the `add` instruction in the 32-bit case or `adc` instruction in the 64-bit case. Similarly, signed overflows can be detected using either the `jo` instruction (jump if overflow) or the `jno` instruction (jump if not overflow) after execution of the `add` instruction (32-bit case) or `adc` instruction (64-bit case).

Precondition. 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 `int` type and `ULLONG_MAX` for addition of `unsigned long long` type.

Addition of signed integers is more complicated, as shown in Table 5–5.

As you test for these preconditions, make sure that the test itself does not overflow. The tests in Table 5–5 are guaranteed not to overflow for appropriately signed values.

Postcondition. Another solution to detecting integer overflow is to perform the addition and then evaluate the results of the operation. For example, to test for overflow of signed integers, let `sum = lhs + rhs`. If `lhs` is non-negative and `sum < rhs`, an overflow has occurred. Similarly, if `lhs` is negative and `sum > rhs`, an overflow has occurred. In all other cases, the addition operation succeeds without overflow. For unsigned integers, if the sum is smaller than either operand, an overflow has occurred.

Subtraction

Subtraction is another additive operation. For subtraction, both operands must have arithmetic type or be pointers to compatible object types. It is also possible to subtract an integer type from a pointer to an object type. Decrementing is equivalent to subtracting one.

Error Detection. The IA-32 instruction set includes `sub` (subtract) and `sbb` (subtract with borrow). The `sub` instruction subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or memory location, and the source operand can be an immediate, register, or memory location.

The `sbb` instruction is usually executed as part of a multi-byte or multi-word subtraction in which a `sub` instruction is followed by an `sbb` instruction. The `sbb` instruction adds the source operand (second operand) and the carry flag and subtracts the result from the destination operand (first operand). The

Table 5–5. Addition of Signed Integers of Type `int`

LHS	RHS	Exceptional condition
Positive	Positive	Overflow if `INT_MAX - LHS < RHS`
Positive	Negative	None possible
Negative	Positive	None possible
Negative	Negative	Overflow if `LHS < INT_MIN - RHS`

result of the subtraction is stored in the destination operand. The carry flag represents a borrow from a previous subtraction.

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

The IA-32 instruction generated by the Visual C++ compiler for subtraction follows the same patterns as addition. Figure 5–14 shows the use of the `sub` and `sbb` instructions in the subtraction of `signed long long` integers `sll2` from `signed long long` integer `sll1`.

Precondition. To test for overflow for unsigned integers, make sure that the LHS < RHS.

For signed integers of the same sign, exceptional conditions cannot occur.

For signed integers of mixed signs, the following applies:

- If LHS is positive and RHS is negative, check that the `lhs` > `INT_MAX` + `rhs` for `signed int` type.
- If LHS is non-negative and RHS is negative, check that `lhs` < `INT_MAX` + `rhs`.

For example, `0 – INT_MIN` causes an overflow condition because the result of the operation is one greater than the maximum representation possible.

Postcondition. To test for overflow of signed integers, let `difference` = `lhs` – `rhs` and apply the following:

- If `rhs` is non-negative and `difference` > `lhs`, an overflow has occurred.
- If `rhs` is negative and `difference` < `lhs`, an overflow has occurred.
- In all other cases, no overflow occurs.
- For unsigned integers, an overflow occurs if `difference` is greater than `lhs`.

```
sll1 - sll2
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]
```

Figure 5–14. Assembly code generated by Visual C++ for subtraction

Multiplication

Multiplication is prone to overflow errors because relatively small operands, when multiplied, can overflow a given integer type.

One solution, which is used by the IA-32 `mul` instruction, is to allocate storage for the product that is twice the size of the larger of the two operands. For example, the product of two 8-bit operands can always be represented by 16 bits, and the product of two 16-bit operands can always be represented by 32 bits.

Each operand of the binary * operator has arithmetic type. The usual arithmetic conversions are performed on the operands.

Error Detection. The IA-32 instruction set includes both a `mul` (unsigned multiply) and `imul` (signed multiply) instruction. The `mul` instruction performs an unsigned multiplication of the first (destination) operand and the second operand (source) operand and stores the result in the destination operand.

The `mul` instruction is shown in Figure 5–15 in C-style pseudocode. The `mul` instruction accepts 8-, 16-, and 32-bit operands and stores the results in 16-, 32-, and 64-bit destination registers, respectively. If the high-order bits are required to represent the product of the two operands, the carry and overflow flags are set. If the high-order bits are not required (that is, they are equal to zero), the carry and overflow flags are cleared.

The IA-32 instruction set also includes `imul`, a signed form of the `mul` instruction with one-, two-, and three-operand forms [Intel 04]. The carry and overflow flags are set when significant bits (including the sign bit) are carried into the upper half of the result and cleared when they do not.

The three forms of the `imul` instruction are similar to the `mul` instruction in that the length of the product is calculated as twice the length of the oper-

```
 1. if (OperandSize == 8) {
 2.   AX = AL * SRC;
 3. else {
 4.   if (OperandSize == 16) {
 5.     DX:AX = AX * SRC;
 6.   }
 7.   else {  // OperandSize == 32
 8.     EDX:EAX = EAX * SRC;
 9.   }
10. }
```

Figure 5–15. IA-32 unsigned multiplication [Intel 04]

ands. With the one-operand form, the product is stored exactly in the destination. With the two- and three-operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the carry or overflow flag must be tested to ensure that no significant bits are lost.

The two- and three-operand forms can also be used with unsigned operands because the lower half of the product is the same regardless of whether the operands are signed or unsigned. The carry and overflow flags, however, cannot be used to determine whether the upper half of the result is nonzero.

Figure 5–16 shows the assembly instructions generated by the Visual C++ .NET compiler for `signed char`, `unsigned char`, and `unsigned int` multiplication. The assembly instructions generated for signed integer multiplication are not shown but are identical to those generated in the unsigned integer case. In all four cases, the two-operand form of the `imul` instruction is used. As previously noted, this form of the `imul` instruction can be used with both signed and unsigned numbers. Signed integers smaller than `int` size are sign extended to 32 bits and unsigned integers are zero extended—accomplishing the integer promotions.

Figure 5–17 shows the assembly code generated by `g++` for the same multiplication operations. Unlike the Visual C++ compiler, `g++` uses the byte form of the `mul` instruction for integers of type `char`, regardless of whether the type is

```
sc_product = sc1 * sc2;
 1. movsx       eax, byte ptr [sc1]
 2. movsx       ecx, byte ptr [sc2]
 3. imul        eax, ecx
 4. mov         byte ptr [sc_product], al

uc_product = uc1 * uc2;
 5. movzx       eax, byte ptr [uc1]
 6. movzx       ecx, byte ptr [uc2]
 7. imul        eax, ecx
 8. mov         byte ptr [uc_product], al

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
```

Figure 5–16. Assembly code generated by Visual C++

```
sc_product = sc1 * sc2;
uc_product = uc1 * uc2;
 1. movb -10(%ebp), %al
 2. mulb -9(%ebp)
 3. movb %al, -11(%ebp)

si_product = si1 * si2;
ui_product = ui1 * ui2;
 4. movl  -20(%ebp), %eax
 5. imull -24(%ebp), %eax
 6. movl  %eax, -28(%ebp)
```

Figure 5–17. Assembly code generated by g++

signed or unsigned, and the `imul` instruction for word-length integers, again regardless of whether the type is signed or unsigned.

Precondition. To prevent an overflow when multiplying unsigned integers, check that A > MAX_INT/B (that is, A * B > MAX_INT).

Another approach that doesn't involve division (and can be more efficient) is to cast both operands to the next larger size and then multiply. As we have already pointed out, the result of the multiplication will always fit in 2*n bits, where n is the number of bits in the larger of the two operands. Integers of 32 bits should be cast to 64 bits, and smaller integers can be cast to 32 bits.

For unsigned integers, you can check high-order bits in the next larger integer and, if any are set, throw an error.

For signed integers, an overflow has not occured if the high-order bits and the sign bit on the lower half of the product are all zeros or ones. This means that for a 64-bit product, for example, the upper 33 bits would need to be all zeros or ones.

Multiplying 64 bits is more difficult because these values cannot be easily extended to larger integers. For details, review the source code to David LeBlanc's SafeInt class [LeBlanc 04].

Postcondition. The postcondition approach can also be used to test for multiplication overflow, although the approach is more complicated than it is for addition because the product can require twice the number of bits required by the larger operand.

For 16-bit (word-length) signed integers, checking for overflow is simplified by casting both the LHS and RHS operands to 32-bit values and storing

their product in a 32-bit destination field. An overflow can be detected when the product right-shifted by 16 bits is not the same as the product right-shifted by 15 bits.

For positive results, this test detects whether the magnitude has overflowed into the sign bit of the low-order 16 bits. For negative results, this test detects whether the magnitude has overflowed into the high-order bits.

Division

Integer division overflows are not obvious—you expect the quotient to be less than the dividend. However, an integer overflow condition occurs when the `MIN_INT` value for a 32- or 64-bit signed integer is divided by –1. For example, in the 32-bit case, `-2,147,483,648/-1` should be equal to `2,147,483,648.` Because `2,147,483,648` cannot be represented as a signed 32-bit integer, the resulting value is `-2,147,483,648/-1 = -2,147,483,648.`

Division is also prone to problems when mixed sign and type integers are involved. For example, a reasonable expectation is that –1/4294967295 would equal zero. If the denominator is an `unsigned int` and the numerator is signed and 32 bits or less, both inputs are cast to an `unsigned int` and then the division is performed. The result is that –1/4294967295 is equal to one and not zero.

Error Detection. The IA-32 instruction set includes the following division instructions: `div`, `divpd`, `divps`, `divsd`, `divss`, `fdiv`, `fdivp`, `fidiv`, and `idiv`. The `div` instruction divides the (unsigned) integer value in the `ax`, `dx:ax`, or `edx:eax` registers (dividend) by the source operand (divisor) and stores the quotient in the `ax` (`ah:al`), `dx:ax`, or `edx:eax` registers. The `idiv` instruction performs the same operations on (signed) values. The results of the `div`/`idiv` instructions depends on the operand size (dividend/divisor), as shown in Table 5–6. The quotient range shown is for the signed (`idiv`) instruction.

Nonintegral results are truncated toward zero. The remainder is always less than the divisor in magnitude. Overflow is indicated by the divide error exception rather than with the carry flag.

Table 5–6. The `idiv` Instruction

Operand size	Dividend	Divisor	Quotient	Rem.	Quotient range
Word/byte	`ax`	r/m8	`al`	`ah`	–128 to +127
Doubleword/word	`dx:ax`	r/m16	`ax`	`dx`	–32,768 to +32,767
Quadword/doubleword	`edx:eax`	r/m32	`eax`	`edx`	-2^{31} to $2^{32}-1$

Figure 5–18 shows the Intel assembly instructions generated by the Visual C++ compiler for signed and unsigned division. As expected, signed division uses the `idiv` instruction and unsigned division uses the `div` instruction. Because the divisor in both the signed and unsigned case is a 32-bit value, the dividend is interpreted as a quadword. In the signed case, this is handled by doubling the size of the `si_dividend` in register `eax` by means of sign extension and storing the result in registers `edx:eax`. In the unsigned case, the `edx` register is cleared using the `xor` instruction before calling the `div` instruction to make sure that there is no residual value in this register.

Unlike the `add`, `mul`, and `imul` instructions, the Intel division instructions `div` and `idiv` do not set the overflow flag; they generate a division error if the source operand (divisor) is zero or if the quotient is too large for the designated register. A divide error results in a fault on interrupt vector 0. A fault is an exception that can generally be corrected and that, once corrected, allows the program to restart with no loss of continuity. When a fault is reported, the processor restores the machine state to the state before the beginning of execution of the faulting instruction. The return address (saved contents of the `cs` and `eip` registers) for the fault handler points to the faulting instruction, rather than the instruction following the faulting instruction [Intel 04].

Precondition. Integer division overflows can be prevented by checking to see whether the numerator is the minimum value for the integer type and the denominator is –1. Division by zero can be prevented, of course, by ensuring that the divisor is nonzero.

```
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

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
```

Figure 5–18. Assembly code generated by Visual C++

Postcondition. Normal C++ exception handling does not allow an application to recover from a hardware exception or fault such as an access violation or divide by zero [Richter 99]. Microsoft does provide a facility called *structured exception handling* (SEH) for dealing with hardware and other exceptions. Structured exception handling is an operating system facility that is distinct from C++ exception handling. Microsoft provides a set of extensions to the C language that enable C programs to handle Win32 structured exceptions.

Figure 5–19 shows how Win32 structured exception handling can be used in a C program to recover from divide-by-zero and overflow faults resulting from division operations. The same figure also shows how SEH *cannot* be used to detect overflow errors resulting from addition or other operations

Lines 5–9 contain a `__try` block containing code that will cause a divide-by-zero fault when executed. Lines 10–15 contain an `__except` block that catches and handles the fault. Similarly, lines 16–25 contain code that will cause an integer overflow fault at runtime and corresponding exception handler to recover from the fault. Lines 26–36 contain an important counter example. The code in the `__try` block results in an integer overflow condition. However, the same exception handler that caught the overflow exception after the division fault will not detect this overflow because the addition operation does not generate a hardware *fault*.

Figure 5–20 shows the implementation of the / operator in C++ that uses structured exception handling to detect division error. The division is nested in a `__try` block. If a division error occurs, the logic in the `__except` block is executed. In this example, a C++ exception is thrown. This is possible because C++ exceptions in Visual C++ are implemented using structured exceptions. This also makes it possible to implement this same / operator using C++ exceptions, as shown in Figure 5–21.

In the Linux environment, hardware exceptions such as division errors are managed using signals. In particular, 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 using the `signal()` call as follows:

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

The `signal()` call accepts two parameters: the signal number and the signal handler's address. But because a division error is a fault, 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 shown in Figure 5–22 throws a C++ exception that can then be caught by the calling function.

```
 1. #include <windows.h>
 2. #include <limits.h>

 3. int main(int argc, char* argv[]) {
 4.   int x, y;

 5.   __try {
 6.     x = 5;
 7.     y = 0;
 8.     x = x / y;
 9.   }
10.   __except (GetExceptionCode() == 11.
                EXCEPTION_INT_DIVIDE_BY_ZERO ?
12.             EXCEPTION_EXECUTE_HANDLER : 13.
                EXCEPTION_CONTINUE_SEARCH){
14.     printf("Divide by zero error.\n");
15.   }

16.   __try {
17.     x = INT_MIN;
18.     y = -1;
19.     x = x / y;
20.   }
21.   __except (GetExceptionCode() == 22.
                EXCEPTION_INT_OVERFLOW ?
23.             EXCEPTION_EXECUTE_HANDLER :
                EXCEPTION_CONTINUE_SEARCH) {
24.     printf("Integer overflow during division.\n");
25.   }

26.   __try {
27.     x = INT_MAX;
28.     x++;
29.     printf("x = %d.\n", x);
30.   }
31.   __except (GetExceptionCode() == 32.
                EXCEPTION_INT_OVERFLOW ?
33.             EXCEPTION_EXECUTE_HANDLER : 34.
                EXCEPTION_CONTINUE_SEARCH) {
35.     printf("Integer overflow during increment.\n");
36.   }

37.   return 0;
38. }
```

Figure 5–19. Structured exception handling in C

```
1. Sint operator /(signed int divisor) {
2.   __try {
3.     return si / divisor;
4.   }
5.   __except(EXCEPTION_EXECUTE_HANDLER) {
6.     throw SintException(ARITHMETIC_OVERFLOW);
7.   }
8. }
```

Figure 5–20. Structured exception handling

```
1. Sint operator /(unsigned int divisor) {
2.   try {
3.     return ui / divisor;
4.   }
5.   catch (...) {
6.     throw SintException(ARITHMETIC_OVERFLOW);
7.   }
8. }
```

Figure 5–21. C++ exception handling

```
1. static void divide_error(int val) {
2.   throw SintException(ARITHMETIC_OVERFLOW);
3. }
```

Figure 5–22. Signal handler

5.5 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. This section describes examples of situations where integer error conditions or faulty logic involving integers can lead to vulnerabilities.

```
 1. void getComment(unsigned int len, char *src) {
 2.   unsigned int size;

 3.   size = len - 2;
 4.   char *comment = (char *)malloc(size + 1);
 5.   memcpy(comment, src, size);
 6.   return;
 7. }

 8. int _tmain(int argc, _TCHAR* argv[]) {
 9.   getComment(1, "Comment ");
10.   return 0;
11. }
```

Figure 5–23. Integer overflow vulnerability

Integer Overflow

Figure 5–23 shows an example of an integer overflow based on a real-world vulnerability in the handling of the comment field in JPEG files [Solar Designer 00].

JPEG files contain a comment field that includes a two-byte length field. The length field indicates the length of the comment, including the two-byte length field. To determine the length of the comment string alone (for memory allocation), the function reads the value in the length field and subtracts two (line 3). The function then allocates the length of the comment plus one byte for the terminating null byte (line 4). There is no error checking to ensure that the length field is valid, which makes it possible to cause an overflow by creating an image with a comment length field containing the value 1. The memory allocation call of zero bytes (1 minus 2 [length field] plus 1 [null termination]) succeeds. The `size` variable is declared as an `unsigned int` (line 2), resulting in a large positive value of `0xffffffff` (from 1 minus 2).

Another real-world example of integer overflow appears in memory allocation. Integer overflow can occur in `calloc()` and other memory allocation functions when computing the size of a memory region. As a result, a buffer smaller than the requested size is returned, possibly resulting in a subsequent buffer overflow.[5]

The following code fragments may lead to vulnerabilities:

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

5. See http://cert.uni-stuttgart.de/advisories/calloc.php.

The `calloc()` library call accepts two arguments: the storage size of the element type and the number of elements. The element type size is not specified explicitly in the case of `new` operator in C++. To compute the size of the memory required, the storage size is multiplied by the number of elements. 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. As a result, the application can write beyond the end of the allocated buffer resulting in a heap-based buffer overflow.

Sign Errors

Figure 5–24 shows a program that is susceptible to a sign error exploit [Horovitz 02]. The program accepts two parameters (the length of data to copy and the actual data) and is flawed because a signed integer is converted to an unsigned integer of equal size. That is, `len` is declared as a signed integer on line 3, which makes it possible to assign a negative value to the variable on line 5. A negative value bypasses the check on line 6 because it is less than the buffer size. In the resulting call to `memcpy()` on line 7, the signed integer value is treated as an unsigned value of type `size_t`.[6] This results in a sign error because the negative length is now interpreted as a large, positive integer with the resulting buffer overflow.

```
 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.    else
10.      printf("Too much data\n");
11.  }
```

Figure 5–24. Implementation containing a sign error

6. The `size_t` type specifies the maximum number of bytes that a pointer references. It is used for a count that must span the full range of a pointer.

This vulnerability can be prevented by restricting the integer `len` to a valid value. This could be accomplished with a more effective range check on line 6 that guarantees `len` greater than 0 but less than `BUFF_SIZE`. It could also be accomplished by declaring `len` as an unsigned integer on line 3—eliminating the conversion from a signed to unsigned type in the call to `memcpy()` and preventing the sign error from occurring.

A real-world example was documented in NetBSD Security Advisory 2000-002.[7] NetBSD 1.4.2 and prior versions used integer range checks of the following form:

```
if (off > len - sizeof(type-name))
  goto error;
```

where both `off` and `len` are signed integers. Because the `sizeof` operator, as defined by the C99 standard, returns an unsigned integer type (`size_t`), the integer promotion rules require that `len - sizeof(`*`type-name`*`)` be computed as an unsigned value. If `len` is less than the value returned by the `sizeof` operator, the subtraction operation underflows and yields a large positive value, which allows the integer range check to be bypassed.

An alternative form of the integer range check that eliminates the problem in this case is as follows:

```
if ((off + sizeof(type-name)) > len) {
   goto error;
```

The programmer still must ensure that the addition operation does not result in an overflow by guaranteeing that the value of `off` is within a defined range.

Truncation Errors

Figure 5–25 duplicates the vulnerable program shown in Figure 5–1 in the introduction to this chapter. The program accepts two string arguments and calculates their combined length (plus an extra byte for the terminating null character) on line 3. The program allocates enough memory on line 4 to store both strings. The first argument is copied into the buffer on line 5 and the second argument is concatenated to the end of the first argument on line 6.

At first glance, you wouldn't expect a vulnerability to exist because the memory is dynamically allocated as required to contain the two strings. How-

7. See ftp://ftp.netbsd.org/pub/NetBSD/misc/security/advisories/NetBSD-SA2000-002.txt.asc.

```
1.  int main(int argc, char *const *argv) {
2.    unsigned short int total;
3.    total = strlen(argv[1])+strlen(argv[2])+1;
4.    char *buff = (char *) malloc(total);
5.    strcpy(buff, argv[1]);
6.    strcat(buff, argv[2]);
7.  }
```

Figure 5–25. Truncation error involving the sum of two lengths

ever, an attacker can supply arguments such that the sum of the lengths of these strings cannot be represented by the `unsigned short` integer `total`. As a result, the value is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type. For example, if the first argument is 65,500 characters and the second argument is 36 characters, the sum of the two lengths + 1 will be 65,537. The `strlen()` function is specified to return a result of type `size_t`, which is typically an `unsigned long` integer. As both 65,500 and 36 are `unsigned long` integers, the sum of the three values is also an `unsigned long` integer. For an `unsigned long` integer value to be assigned to the variable `total`, an `unsigned short` integer, a demotion is required.

Assuming 16-bit `short` integers, the result is (65500 + 37) % 65536 = 1. The `malloc()` call successfully allocates the requested byte and the `strcpy()` and `strcat()` invocations create a buffer overflow condition.

Figure 5–26 contains another example of a how a truncation error may lead to a vulnerability [Howard 03a]. The formal parameter `cbBuf` is declared

```
 1.  bool func(char *name, long cbBuf) {
 2.    unsigned short bufSize = cbBuf;
 3.    char *buf = (char *)malloc(bufSize);
 4.    if (buf) {
 5.      memcpy(buf, name, cbBuf);
 6.      if (buf) free(buf);
 7.      return true;
 8.    }
 9.    return false;
10.  }
```

Figure 5–26. Implementation vulnerable to a truncation error exploit

as a `long` and used as the size in the `memcpy()` operation on line 5. The `cbBuf` parameter is also used to initialize `bufSize` on line 2, which, in turn, is used to allocate memory for `buf` on line 3.

At first glance, this function appears to be immune from a buffer overflow because the size of the destination buffer for `memcpy()` is dynamically allocated. But the problem is that `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. Therefore, a truncation error will occur on line 2 for any values of `cbBuf` between 65,535 and 2,147,483,647. This would only be an error and not a vulnerability if `bufSize` were used for both the calls to `malloc()` and `memcpy()`. However, 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.[8]

5.6 Nonexceptional Integer Logic Errors

Many exploitable software flaws do not require an exceptional condition to occur but are simply a result of poorly written code.

Figure 5–27 shows a vulnerability caused by using a signed integer as an index variable. The `insert_in_table()` function inserts `value` at position `pos` in an array of integers. Storage for the array is allocated on the heap on line 4 the first time the function is called. Lines 6–8 ensure that `pos` is not greater than 99. The value is inserted into the array at the specified position on line 9.

Although no exceptional condition can occur, there is a vulnerability resulting from the lack of 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 on line 6, but a negative value would not.

The following assignment statement from line 9:

```
table[pos] = value;
```

is equivalent to

```
*(table + (pos * sizeof(int))) = value;
```

8. Compiling this example with the `/W3` or `/W4` option of Visual C++ results in a "possible loss of data" warning.

```
 1. int *table = NULL;

 2. int insert_in_table(int pos, int value){
 3.   if (!table) {
 4.     table = (int *)malloc(sizeof(int) * 100);
 5.   }
 6.   if (pos > 99) {
 7.     return -1;
 8.   }
 9.   table[pos] = value;
10.   return 0;
11. }
```

Figure 5–27. Negative indices

The `sizeof(int)` resolves to 4 when using Visual C++ on IA-32. If a negative `pos` value is passed as an argument, `value` would be written to a location `pos` x 4 bytes before the start of the actual buffer. This flaw could be eliminated two ways: by declaring the formal argument `pos` as an `unsigned integer` or by checking upper and lower bounds on line 6.

5.7 Mitigation Strategies

All integer vulnerabilities result from *integer type range errors*. For example, integer overflows occur when integer operations generate a value that is out of range for a particular integer type. Truncation errors occur when a value is stored in a type that is too small to represent the result. Sign errors occur when a negative value is saved in a signed type that cannot accommodate the value's range. Even the logic errors described in this chapter are the result of improper range checking.

Because all integer vulnerabilities are type range errors, *type range checking—if properly applied—can eliminate all integer vulnerabilities*. Languages such as Pascal and Ada allow range restrictions to be applied to any scalar type to form subtypes. Ada, for example, allows range restrictions to be declared on derived types using the range keyword:

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

The range restrictions are then enforced by the language runtime. C and C++, on the other hand, are not nearly as good at enforcing type safety. Fortunately, there are some avoidance strategies that can be used to reduce or eliminate the risk from integer-type range errors.

Range Checking

As you may have expected, the burden for integer range checking in C and C++ is placed squarely on the programmer's shoulders. Sometimes it's relatively easy, sometimes it's not. It's relatively easy, for example, to check an integer value to make sure it is within the proper range before using it to index an array.

The sign-error exploit in Figure 5–24, for example, showed how an out-of-range value (a negative integer) could be used to bypass a check on the upper bounds of an array and cause a buffer overflow. Figure 5–28 shows a version of this program that has improved implicit type checking and explicit range checks. Changes from less secure versions are shown in **bold**. The implicit type check results from the declaration, on line 3, of `len` as an unsigned integer. In general, it is a good idea to use unsigned types for indices, sizes, and loop counters that should never have negative values. The `memcpy()` call on line 7 is also protected by an explicit range check on line 6 that tests both the upper and lower bounds.

Declaring `len` to be an unsigned integer is insufficient for range restriction because it only restricts the range from `0..MAX_INT`. The range check on line 6 is sufficient to ensure that no out-of-range values are passed to `memcpy()` as long as both the upper and lower bounds are checked. Using both the implicit

```
 1.  #define BUFF_SIZE 10

 2.  int main(int argc, char* argv[]){
 3.    unsigned int len;
 4.    char buf[BUFF_SIZE];
 5.    len = atoi(argv[1]);
 6.    if ((0 < len) && (len < BUFF_SIZE) ){
 7.      memcpy(buf, argv[2], len);
 8.    }
 9.    else
10.      printf("Too much data\n");
11.  }
```

Figure 5–28. Implementation with implicit type and explicit range checking

and explicit checks may be redundant, but we recommend this practice as "healthy paranoia."

In other cases, type range checking is more complicated. For example, if x is assigned the product of a, b, and c, it is necessary to limit the range of a, b, and c so that the value of x cannot exceed the range of values for whichever integer type x is declared. As integer variables are operated on multiple times in combination with other integer values, it becomes increasingly difficult to ensure that an integer type range error does not occur.

While this problem is difficult to solve, there are valid software engineering techniques that can help. First, all external inputs should be evaluated to determine whether there are identifiable upper and lower bounds. If so, these limits should be enforced by the interface. Anything that can be done to limit the input of excessively large or small integers should help prevent overflow and other type range errors. Furthermore, it is easier to find and correct input problems than it is to trace internal errors back to faulty inputs.

Second, typographic conventions can be used in the code to distinguish constants from variables. They can even be used to distinguish externally influenced variables from locally used variables with well-defined ranges.

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

Strong Typing

One way to provide better type checking is to provide better types. Using an unsigned type, for example, can guarantee that a variable does not contain a negative value. However, this solution does not prevent overflow or solve the general case.

Data abstractions can support data ranges in a way that standard and extended integer types cannot. Data abstractions are possible in both C and C++, although C++ provides more support. For example, if an integer was required to store the temperature of water in liquid form using the Fahrenheit scale, we could declare a variable as follows:

```
unsigned char waterTemperature;
```

Using `waterTemperature` to represent an unsigned 8-bit value from 1–255, is sufficient: water ranges from 32 degrees Fahrenheit (freezing) to 212 degrees Fahrenheit (the boiling point). However, this type does not prevent overflow and also allows for invalid values (that is, 1–31 and 213–255).

One solution is to create an abstract 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 the `waterTemperature` does not leave the valid range. If this is done properly, there is no possibility of an integer type range error occurring.

This data abstraction is easy to write in C++ and C. A C programmer could specify `create()` and `destroy()` methods instead of constructors and destructors but would not be able to redefine operators. Inheritance and other features of C++ are not required to create usable data abstractions.

Compiler-Generated Runtime Checks

In a perfect world, C and C++ compilers would flag exceptional conditions as code is generated and provide a mechanism (such as an exception, trap, or signal handler) for applications to handle these events. Unfortunately, the world we live in is far from perfect. A brief description of some of the capabilities that exist today follows.

Visual C++. Visual C++ .NET 2003 includes native runtime checks that catch truncation errors as integers are assigned to shorter variables that result in lost data. For example, the `/RTCc` compiler flag catches those errors and creates a report. Visual C++ also 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.

Although this seems like a useful feature, runtime error checks are not valid in a release (optimized) build, presumably for performance reasons.[9]

GCC. The `gcc` and `g++` compilers include an `-ftrapv` compiler option that provides limited support for detecting integer exceptions at runtime. According to the `gcc` man page, this option "generates traps for signed overflow on addition, subtraction, multiplication operations." In practice, this means that the `gcc` compiler generates calls to existing library functions rather than generating assembler instructions to perform these arithmetic operations on signed integers. If you use this feature, make sure you are using `gcc` version 3.4 or later because the checks implemented by the runtime system before this version do not adequately detect all overflows and should not be trusted.[10]

9. See Visual C++ Compiler Options, /RTC (Run-Time Error Checks), Visual Studio .NET help system.

10. See VU#540517 "libgcc contains multiple flaws that allow integer type range vulnerabilities to occur at runtime" at http://www.kb.cert.org/vuls/id/540517.

Safe Integer Operations

Integer operations can result in error conditions and possible lost data, particularly when inputs to these operations can be manipulated by a potentially malicious user.

The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. However, 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.

An alternative or ancillary approach is to protect each operation. However, because of the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, this approach can be prohibitively labor intensive and expensive to implement.

A more economical solution to this problem is to use a safe integer library for all operations on integers where one or more of the inputs could be influenced by an untrusted source. Figure 5–29 shows examples of when to use safe integer operations.

The first example shows a function that accepts two parameters specifying the size of a given structure and the number of structures to allocate that can be manipulated by untrusted sources. These two values are then multiplied to determine what size memory to allocate. Of course, the multiplication operation

Use Safe Integer Operations

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

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

Don't Use Safe Integer Operations

```
void foo() {
  int i;

  for (i = 0; i < INT_MAX; i++)
    ....
}
```

Figure 5–29. Checking for overflow when adding two signed integers

```
1. Wtype __addvsi3 (Wtype a, Wtype b) {
2.   const Wtype w = a + b;

3.   if (b >= 0 ? w < a : w > a)
4.     abort ();
5.   return w;
6. }
```

Figure 5–30. Checking for overflow when adding two signed integers

could easily overflow the integer variable and provide an opportunity to exploit a buffer overflow.

The second example shows when *not* to use safe integer operations. The integer `i` is used in a tightly controlled loop and is not subject to manipulation by an untrusted source, so using safe integers would add unnecessary performance overhead.

Safe integer libraries use different implementation strategies. The `gcc` library uses postconditions to detect integer errors. SafeInt C++ class tests preconditions to prevent integer errors. The RCSint Class and a library by Michael Howard take advantage of machine-specific mechanisms to detect integer errors. We compare and contrast these four approaches in the remainder of this section.

GCC. The `gcc` runtime system generates traps for signed overflow on addition, subtraction, and multiplication operations for programs compiled with the `-ftrapv` flag. To accomplish this, calls are made to existing, portable library functions that test an operation's postconditions and call the C library `abort()` function when results indicate that an integer error has occurred.

Figure 5–30 shows a function from the `gcc` runtime system that is used to detect overflows resulting from the addition of signed 16-bit integers. The addition operation is performed on line 2 and the results are compared to the operands to determine whether an overflow condition has occurred. For `_addvsi3()`, if `b` is non-negative and `w < a`, an overflow has occurred and `abort()` is called. Similarly, `abort()` is also called if `b` is negative and `w > a`.

C Language Compatible Library. Michael Howard has written parts of a safe integer library that detects integer overflow conditions using architecture-specific mechanisms [Howard 03b].

Figure 5–31 shows a function that performs unsigned addition. Figure 5–32 shows a version of the vulnerable program from Figure 5–25 that has been

```
 1. in bool UAdd(size_t a, size_t b, size_t *r) {
 2.   __asm {
 3.     mov eax, dword ptr [a]
 4.     add eax, dword ptr [b]
 5.     mov ecx, dword ptr [r]
 6.     mov dword ptr [ecx], eax
 7.     jc  short j1
 8.     mov al, 1 // 1 is success
 9.     jmp short j2
10. j1:
11.     xor al, al // 0 is failure
12. j2:
13.   };
14. }
```

Figure 5–31. Unsigned integer addition and multiplication operations

```
 1. int main(int argc, char *const *argv) {
 2.   unsigned int total;
 3.   if (UAdd(strlen(argv[1]), 1, &total) &&
          UAdd(total, strlen(argv[2]), &total)) {
 4.       char *buff = (char *)malloc(total);
 5.       strcpy(buff, argv[1]);
 6.       strcat(buff, argv[2]);
 7.   } else {
 8.     abort();
 9.   }
10. }
```

Figure 5–32. C language compatible library solution

modified (shown in bold) to use the Howard library. The calculation of the total length of the two strings is performed using the `UAdd()` call on lines 3–4 with appropriate checks for error conditions. Even adding one to the sum can result in an overflow and needs to be protected.

Advantages of the Howard approach are that it can be used in both C and C++ programs and it is efficient: assembly language instructions are the same as those generated by a compiler except that they integrate checks for carry and

other conditions that indicate exceptions and set a return code. Drawbacks are usability (an awkward API) and portability: the use of embedded Intel assembly instructions prevents porting to other architectures.

SafeInt Class. SafeInt is a C++ template class written by David LeBlanc [LeBlanc 04]. SafeInt generally implements the precondition approach and tests the values of operands before performing an operation to determine whether errors might occur. The class is declared as a template, so it can be used with any integer type. Nearly every relevant operator has been overridden.

Figure 5–33 shows a section of code from the SafeInt class that checks for overflow in signed integer addition. Figure 5–34 shows a version of the vulnerable program from Figure 5–25 that has been modified (boldface type) to use the SafeInt library. Lines 1–4 show the implementation for the SafeInt + operator, which is invoked twice on line 9 of the main routine. The variables `s1` and `s2` are declared as SafeInt types on lines 7 and 8. In both cases, the SafeInt class is instantiated as an unsigned long type. When the + operator is invoked (twice) on line 9, it uses the safe version of the operator implemented as part of the SafeInt class. The safe version of the operator guarantees that an exception is generated if the result is invalid.

```
 1. if (!((rhs ^ lhs) < 0)) { //test for +/- combo
 2.   //either two negatives, or 2 positives
 3.   if (rhs < 0) {
 4.     //two negatives
 5.     if (lhs < MinInt() - rhs) { //remember rhs < 0
 6.       throw ERROR_ARITHMETIC_UNDERFLOW;
 7.     }
 8.     //ok
 9.   }
10.   else {
11.     //two positives
12.     if (MaxInt() - lhs < rhs) {
13.       throw ERROR_ARITHMETIC_OVERFLOW;
14.     }
15.     //OK
16.    }
17.  }
18.  //else overflow not possible
19.  return lhs + rhs;
```

Figure 5–33. Checking for overflow when adding two signed integers

```
1. //addition
2.SafeInt<T> operator +(SafeInt<T> rhs) {
3. return SafeInt<T>(addition(m_int,rhs.Value()));
4.}

5. int main(int argc, char *const *argv) {
6.   try{
7.     SafeInt<unsigned long> s1(strlen(argv[1]));
8.     SafeInt<unsigned long> s2(strlen(argv[2]));
9.     char *buff = (char *) malloc(s1 + s2 + 1);
10.    strcpy(buff, argv[1]);
11.    strcat(buff, argv[2]);
12.  }
13.  catch(SafeIntException err) {
14.    abort();
15.  }
16. }
```

Figure 5–34. SafeInt solution

The SafeInt library has several advantages over the Howard approach. Because it is written entirely in C++, it is more portable than safe arithmetic operations that depend on assembly language instructions. It is also more usable: arithmetic operators can be used in normal inline expressions, and SafeInt uses C++ exception handling instead of C-style return code checking. One disadvantage is that SafeInt is larger and slower than the Howard approach.

The precondition approach could also be implemented in C-compatible libraries, although the advantages derived from C++ would not be realized.

RCSint Class. RCSint combines the usability of the SafeInt template class with the performance of the Howard approach. It stands for "reliable, convenient, and secure integers."

Figure 5–35 shows the `si_multiply()` method from the RCSint class. The method multiplies two signed integers of type `int`. The multiplication operation is implemented in embedded assembly language code. If the multiplication succeeds, the method simply returns with the result in the `eax` register. If an overflow condition occurs, the program throws the `ERROR_ARITHMETIC_OVERFLOW` exception.

The RCSint approach supports a C++ interface (including operators) for ease of use and uses Intel assembly instructions for speed. Like the Howard approach, it cannot be easily ported to other hardware architectures.

```
 1. int si_multiply(int si1, int si2) {

 2.   __asm {
 3.     mov  eax, dword ptr [si1]
 4.     mov  ecx, dword ptr [si2]
 5.     imul ecx
 6.     jno  nof
 7.   };

 8.   throw SintException(ERROR_ARITHMETIC_OVERFLOW);
 9. nof:
10.   return;
11. }
```

Figure 5–35. Checking for overflow when multiplying two signed integers

Arbitrary Precision Arithmetic

There are many arbitrary precision arithmetic packages available, primarily for scientific computing. However, these can also solve the problem of integer type range errors, which result from a lack of precision in the representation.

GNU Multiple Precision Arithmetic Library (GMP). GMP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It was designed to provide the fastest possible arithmetic for applications that require higher precision than what is directly supported by the basic C types.

GMP emphasizes speed over simplicity or elegance. It uses sophisticated algorithms, full words as the basic arithmetic type, and carefully optimized assembly code.

Java BigInteger. Newer versions of the Java JDK contain a `BigInteger` class in the `java.math` package. It provides arbitrary-precision integers as well as analogs to all of Java's primitive integer operators. While this does little for C and C++ programmers, it does illustrate that the concept is not entirely foreign to language designers.

Testing

Checking the input values of integers is a good start, but it does not guarantee that subsequent operations on these integers will not result in an overflow or

other error condition. Unfortunately, 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. For example, 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 these integer variables or, in cases where 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 for the following.

- Integer type ranges are properly checked.
- Input values are restricted to a valid range based on their intended use.
- Integers that cannot assume negative values (for example, ones used for indices, sizes, or loop counters) are declared as unsigned and properly range-checked for upper and lower bounds.
- All operations on integers originating from untrusted sources are performed using a safe integer library.

Also, make sure that your "safe" integer library is really safe and protects against all error conditions identified or referenced in this chapter.

5.8 Notable Vulnerabilities

This section describes notable vulnerabilities resulting from the incorrect handling of integers in C and C++.

XDR Library

There is an integer overflow present in the `xdr_array()` function distributed as part of the Sun Microsystems XDR (external data representation) library. This overflow has been shown to lead to remotely exploitable buffer overflows in multiple applications, leading to the execution of arbitrary code. Although the library was originally distributed by Sun Microsystems, multiple vendors

have included the vulnerable code in their own implementations. The initial vulnerability research and demonstration was performed by Internet Security Systems (ISS) and has been described in

- SunRPC `xdr_array` buffer overflow

  http://www.iss.net/security_center/static/9170.php
- Sun Alert Notification ID: 46122

  http://sunsolve.sun.com/search/document.do?assetkey=1-26-46122-1
- CERT Advisory CA-2002-25

  http://www.cert.org/advisories/CA-2002-25.html
- CERT Vulnerability Note VU#192995

  http://www.kb.cert.org/vuls/id/192995

The XDR libraries are used to provide platform-independent methods for sending data from one system process to another, typically over a network connection. Such routines are commonly used in remote procedure call (RPC) implementations to provide transparency to application programmers who need to use common interfaces to interact with many different types of systems. The `xdr_array()` function in the XDR library provided by Sun Microsystems contains an integer overflow that can lead to improperly sized dynamic memory allocation. Subsequent problems such as buffer overflows can result, depending on how and where the vulnerable `xdr_array()` function is used.

Windows DirectX MIDI Library

The Microsoft Windows DirectX library, `quartz.dll`, contains an integer overflow vulnerability that allows an attacker to execute arbitrary code or crash any application using the library, causing a denial of service. This vulnerability has been described in:

- eEye Digital Security advisory AD20030723

  http://www.eeye.com/html/Research/Advisories/AD20030723.html
- Microsoft Advisory MS03-030

  http://www.microsoft.com/technet/security/bulletin/ms03-030.mspx
- CERT Advisory CA-2003-18

  http://www.cert.org/advisories/CA-2003-18.html

- CERT Vulnerability Note VU#561284
  http://www.kb.cert.org/vuls/id/561284

Windows operating systems include multimedia technologies called DirectX and DirectShow. According to Microsoft Advisory MS03-030,

> DirectX consists of a set of low-level Application Programming Interfaces (APIs) that are used by Windows programs for multimedia support. Within DirectX, the DirectShow technology performs client-side audio and video sourcing, manipulation, and rendering.

DirectShow support for MIDI files is implemented in a library called `quartz.dll`. Because this library does not adequately validate the tracks value in the `MThd` section of MIDI files, a specially crafted MIDI file could cause an integer overflow, leading to heap memory corruption.

Any application that uses DirectX or DirectShow to process MIDI files could be affected by this vulnerability. Of particular concern, Internet Explorer (IE) loads the vulnerable library to process MIDI files embedded in HTML documents. An attacker could therefore exploit this vulnerability by convincing a victim to view an HTML document (for example, Web page, HTML e-mail message) containing an embedded MIDI file. Note that a number of applications (for example, Outlook, Outlook Express, Eudora, AOL, Lotus Notes, Adobe PhotoDeluxe) use the IE HTML rendering engine (WebBrowser ActiveX control) to interpret HTML documents.

Bash

The GNU Project's Bourne Again Shell (`bash`) is a drop-in replacement for the UNIX Bourne shell (`/bin/sh`). It has the same syntax as the standard shell but provides additional functionality such as job control, command-line editing, and history.

Although `bash` can be compiled and installed on almost any UNIX platform, its most prevalent use is on Linux, where it has been installed as the default shell for most applications. The bash source code is freely available from many sites on the Internet.

A vulnerability exists in `bash` versions 1.14.6 and earlier where `bash` can be tricked into executing arbitrary commands. This vulnerability is described in CERT Advisory CA-1996-22.[11]

11. See http://www.cert.org/advisories/CA-1996-22.html.

There is a variable declaration error in the `yy_string_get()` function in the `parse.y` module of the `bash` source code. This function is 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. 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.

However, –1 is used in other parts of the parser to indicate the end of a command. Thus, the character code 255 decimal (377 octal) serves as an unintended command separator for commands given to `bash` via the `-c` option. For example,

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

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

5.9 Summary

Integer vulnerabilities result from lost or misrepresented data. The key to preventing these vulnerabilities is to understand the nuances of integer behavior in digital systems and carefully apply this knowledge in the design and implementation of your systems.

Limiting integer inputs to a valid range can prevent the introduction of arbitrarily large or small numbers that can be used to overflow integer types. Many integer inputs have well-defined ranges (for example, an integer representing a date or month). Other integers have reasonable upper and lower bounds. For example, because Jeanne Calment, believed by some to be the world's longest living person, died at age 122, it should be reasonable to limit an integer input representing someone's age from 0–150. For some integers it can be difficult to establish an upper limit. Usability advocates would argue against imposing arbitrary limits, introducing a trade-off between security and usability. However, if you accept arbitrarily large integers, you must ensure that operations on these values do not cause integer errors that then result in integer vulnerabilities.

Ensuring that operations on integers do not result in integer errors requires considerable care. Programming languages such as Ada do a good job

of enforcing integer type ranges, but if you are reading this book, you are probably not programming in Ada. Ideally, C and C++ compilers will one day provide options to generate code to check for overflow conditions. But until that day, it is a good idea to use one of the safe integer libraries discussed in this chapter as a safety net.

As always, it makes sense to apply available tools, processes, and techniques in the discovery and prevention of integer vulnerabilities. Static analysis and source code auditing are useful for finding errors. Source code audits also provide a forum for developers to discuss what does and does not constitute a security flaw and to consider possible solutions. Dynamic analysis tools, combined with testing, can be used as part of a quality assurance process, particularly if boundary conditions are properly evaluated.

If integer type range checking is properly applied and safe integer operations are used for values that can pass out of range (particularly due to external manipulation), it is possible to prevent vulnerabilities resulting from integer range errors.

5.10 Further Reading

David LeBlanc covers some additional problem areas with integers such as comparison operators [LeBlanc 04]. Blexim discusses integer overflows and signedness bugs, as well as giving some real-world examples [blexim 02].