Computer Security

Robust and secure programming in C

Marius Minea

marius@cs.upt.ro

12 October 2017

In this lecture

Write correct code

minimizing risks

with proper error handling

avoiding security pitfalls

portable

some C-specific, some general

It all starts with numbers

Math is perfect, computer have limits
  sometimes easier to reach than one might think

Numeric types differ in C and mathematics.

In math: $\mathbb{Z} \subset \mathbb{R}$, both are *infinite*, $\mathbb{R}$ is dense/uncountable.

In C: `int`, `float`, `double` are *finite!*
  both have *limited range*, reals have *finite precision*

Important to remember this! (overflows, precision loss)

Integer overflows (CERT rules INT-30C, INT-32C)

(Almost) all operations can give results that don't fit

For `unsigned`: called *wrapping*

Check before: `if (y > UINT_MAX - x) //bad`
or after: `sum = x + y; if (sum < x) //bad`

Interval midpoint, e.g. binary search:
Bad code: `unsigned m = (lo + hi) / 2;`
Good code: `unsigned m = lo + (hi - lo) / 2;`

Dangerous: product in `malloc`

```
char *p = malloc(x * y * sizeof(int));
```

if product wraps, array will overflow on use

Signed integer overflow

Worse, result is undefined according to standard

could silently wrap (like for unsigned), or generate trap (exception), or have different behavior in different contexts.

Same kind checks as for unsigned, but need to do *before*

Signedness and overflows (cont.)

WARNING `char` may be `signed` or `unsigned`
(implementation dependent, check `CHAR_MIN`: 0 or `SCHAR_MIN`)
$\Rightarrow$ different `int` conversion if bit 7 is 1 (`'\xff'` = `-1`)
`getchar/putchar` work with `unsigned char` converted to `int`

CAREFUL when comparing / converting `signed` and `unsigned`

```
if (-5 > 4333222111u) printf("-5 > 4333222111 !!!\n");
```

because `-5` converted to `unsigned` has higher value

Correct comparison between `int i` and `unsigned u`:

```
if (i < 0 || i < u)    or    if (i >= 0 && i >= u)
```

(compares `i` and `u` only if `i` is nonnegative)

ERRORS with bitwise operators

DON'T right-shift a negative int!

~~`int n = ...; for ( ; n; n >>= 1 ) ...`~~

May loop forever if `n` negative; the topmost bit inserted is usually the sign bit (implementation-defined). Use `unsigned` (inserts a 0).

DON'T shift with more than bit width (behavior undefined)
(in some implementations, shifts with count modulo bitwidth)

Watch out for overflows and imprecision!

`int` (even `long`) may have small range (32 bits: $\pm$ 2 billion)
Not enough for computations with large integers (factorial, etc.)
Use `double` (bigger range) or arbitrary precision libraries (bignum)

Floating point has limited precision: beyond 1E16, `double` does not distinguish two consecutive integers!

A decimal value may not be precisely represented in base 2:
may be periodic fraction: $1.2_{(10)} = 1.(0011)_{(2)}$
`printf("%f", 32.1f);` writes 32.099998

Due to precision loss in computation, result may be inexact
$\Rightarrow$ replace `x==y` test with `fabs(x - y)` < *small epsilon*
(depending on the problem)

Differences smaller than precision limit cannot be represented:
$\Rightarrow$ for `x < DBL_EPSILON` (ca. 10^{-16}) we have `1 + x == 1`

Evaluation order and side effects

precedence: between *different* operators
know precedence table, use parantheses for clear code

associativity: same operator (left-assoc. or right-assoc.)

evaluation order: of operands to *same* operator
unspecified for most operators

DON'T use side effects in complex expressions!
`2 * f(x) + g(x)` : multiplication before addition (precedence)
Unspecified which part of sum is evaluated first (`f` or `g`)

```
if (getchar() == '/' && getchar() == '*') // comment star
else // have I read one char or two ???
```

Error-checking patterns

Get errors out of the way first

```
int errcode = function(args);
if (errcode != 0) get_out("error message");
// continue normal processing
```

Don't just check some error codes, there may be others!

```
if (errcode == ERR1) handle("msg1");
else if (errcode == ERR2) handle ("msg2");
else if (errcode) handle("some other error");
else // ok
```

Always checking for successful (correct) input!

Reading the desired data might not succeed for two reasons:

system: no more data (end-of-file), read error, etc.

user: data not in needed format (illegal char, not number, etc.)

A function can report both a *result* and an *error code* as follows:

1. *expand result datatype* to include error code

   `getchar()` : `unsigned char` converted to `int`,

   or `EOF (-1)` which is different from any `unsigned char`

2. return type may have a special *invalid/error value*

   `fgets` returns address where the line was read (first argument)

   or `NULL` (invalid pointer value) when nothing read

3. return *error code* and *store result* at given pointer

   `scanf` *returns no. of items read* (can be 0, or `EOF` at end-of-input)

   takes as arguments *addresses* where it should place read data

Understanding end-of-file

Input functions only detect EOF if trying to read *past* the end
  ⇒ testing `feof(somefile)` can be misleading

e.g., read sequence of whitespace-separated numbers
  if no more input follows, `feof(stdin)` true after read
  if char follows (Enter, space, etc.) `feof(stdin)` still false
    (but there may be no other number left)

⇒ can't and should not use `feof(...)` as test in the read loop

Process input while correct

Checking for end-of-input explicitly is rarely needed.

The point of processing is to *read data*

⇒ thus we must check that data was read successfully:

`while (`*read successful*`)` *use data*

On exit from loop, if `feof(stdin)`, input is finished

else input does not match format ⇒ read next char(s) and report

DO NOT write code of the form

```
while (!feof(stdin))
    scanf("%d", &n);
```

After last good read (number), end-of-input is not yet reached

unless no more separators (whitespace, incl. newline) after it

⇒ next read will not succeed, but is not checked

If read is checked (as it *MUST* be), testing `EOF` is not needed:

```
while (scanf("%d", &n) == 1)
  // process n
```

Check bounds when filling an array

Often, we have to fill an array up to some stopping condition:
- read from input upto a given character (period, `\n`, etc)
- copy from another string or array

Arrays must not be written beyond their length!

Loop should **first** *test array is not full!*

```
for (int i = 0; i < len; ++i) { // limit to array size
   tab[i] = ...;       // assign with value read
   if (normal stopping condition) break/return;
}
// here we can test if maximal length reached
// and report if needed
```

Error handling

MUST check return code of *every* I/O function

Failure modes may not be obvious

`fclose(f)` : all done, can still have error
  no room to flush to disk; USB stick removed; HW failure
    $\Rightarrow$ must report, perhaps chance to recover

Even printing to `stdout` could fail (redirected, no more space)

Meaningful error messages: errno / perror

Error codes

global variable `int errno` declared in `errno.h`
contains code of last error in a library function
(illegal operation, file not found, not enough memory, etc.)

Careful: use `errno` only if sure that there was an error
for safety reset before calling function that may fail

Function `void perror(const char *s)` from `stdio.h`
prints user message `s`, a colon `:` and then the error description
(same as given by `char *strerror(int errnum)` from `string.h`)

String to num: use functions w/ error reporting

Converting strings to numbers `int n = atoi(s);` returns 0 on error, but also for `"0"`

Avoid. Use only when string known to be good.

```
int n; char s[] = "  -102  56 42";
if (sscanf(s, "%d", &n) == 1) ... //number OK
```

but we don't know where processing of string stopped

also does not signal overflow (if number too large)

```
long int strtol(const char *nptr, char **endptr, int base);
```

assigns to `*endptr` the address of first unprocessed char

```
char *end; long n = strtol(s, &end, 10);
```

base 10 or other

also `strtoul` for `unsigned long`, `strtod` for base 10 `double`

set `errno` to `ERANGE` on overflow

Always limit input!

C11 standard removed function ~~gets~~: did not limit size read
$\Rightarrow$ it is impossible to use ~~gets~~ safely
$\Rightarrow$ buffer overflow, memory corruption, security vulnerabilities

Use: `char *fgets(char *s, int size, FILE *stream);`

Reads up to and including newline `\n`, max. `size-1` characters, stores line in array `s`, adds `'\0'` at the end.

NEVER use ~~`%s: scanf("%s",...)`~~. Leads to buffer overflow.

MUST give max. string length in format!

```
char str[30];
if (scanf("%29s", str) != 1) { /* handle error */ }
else { /* word (up to first whitespace) is in s */ }
```

What is the root of the evil ?

C is *low-level*

No (safe) notion of memory object

Can create (almost) arbitrary pointers

Can't really pass an array to a function
  pointer passed instead
  address carries no length information

Must pass array length as separate parameter
  but programmer responsible for passing correct value
  even for heap-allocated chunk, length is available to library, but not checked on use

Unsafe functions and replacements

`strcpy`
  `strncpy` – but was really designed for fixed-length strings

`strcat` seldom a logical reason to use
  `strncat` careful: can write n+1 bytes

`sprintf`
  `snprintf`