Buffer overflow vulnerabilities

— part 1 —

Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt

What is a buffer overflow?

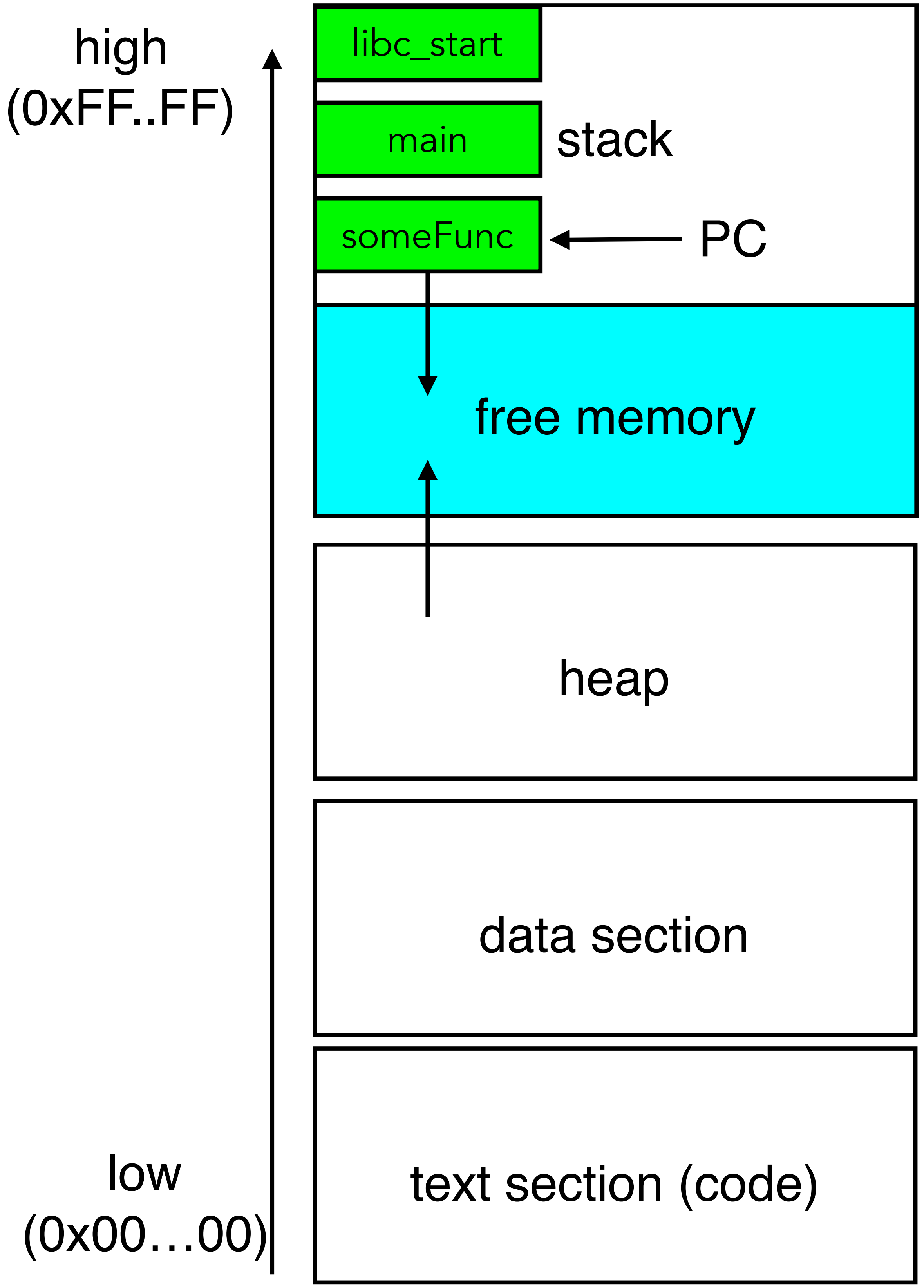
- **CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer**
    - “*The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.* “
- This is a general definition for buffer overflow, that makes no distinction for:
    - the **type of operation**: read or write
    - the **memory area**: stack, heap, … (**Q:** heap? stack?)
    - the **position of invalid memory position relative to buffer**: before (“underflow”) or after
    - the **reason for invalid access**: iteration, copy, pointer arithmetic
- A number of CWEs are specific instances of CWE-119 (next).

Specific types of buffer overflow

- **CWE-120**: **Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')**
- **CWE-121 — Stack-Based Buffer Overflow** — “[…] *the buffer being overwritten is allocated on the **stack*** […]”
- **CWE-122 — Heap-Based Buffer Overflow** — “[…] *the buffer that can be overwritten is allocated in the **heap** portion of memory* […]“
- **CWE-123: Write-what-where Condition** - “*ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow*”.
- **CWE-124: Buffer Underwrite ('Buffer Underflow')**
- **CWE-125: Out-of-bounds Read**
- **CWE-126: Buffer Over-read**
- **CWE-127: Buffer Under-read**

Memory address space of a process

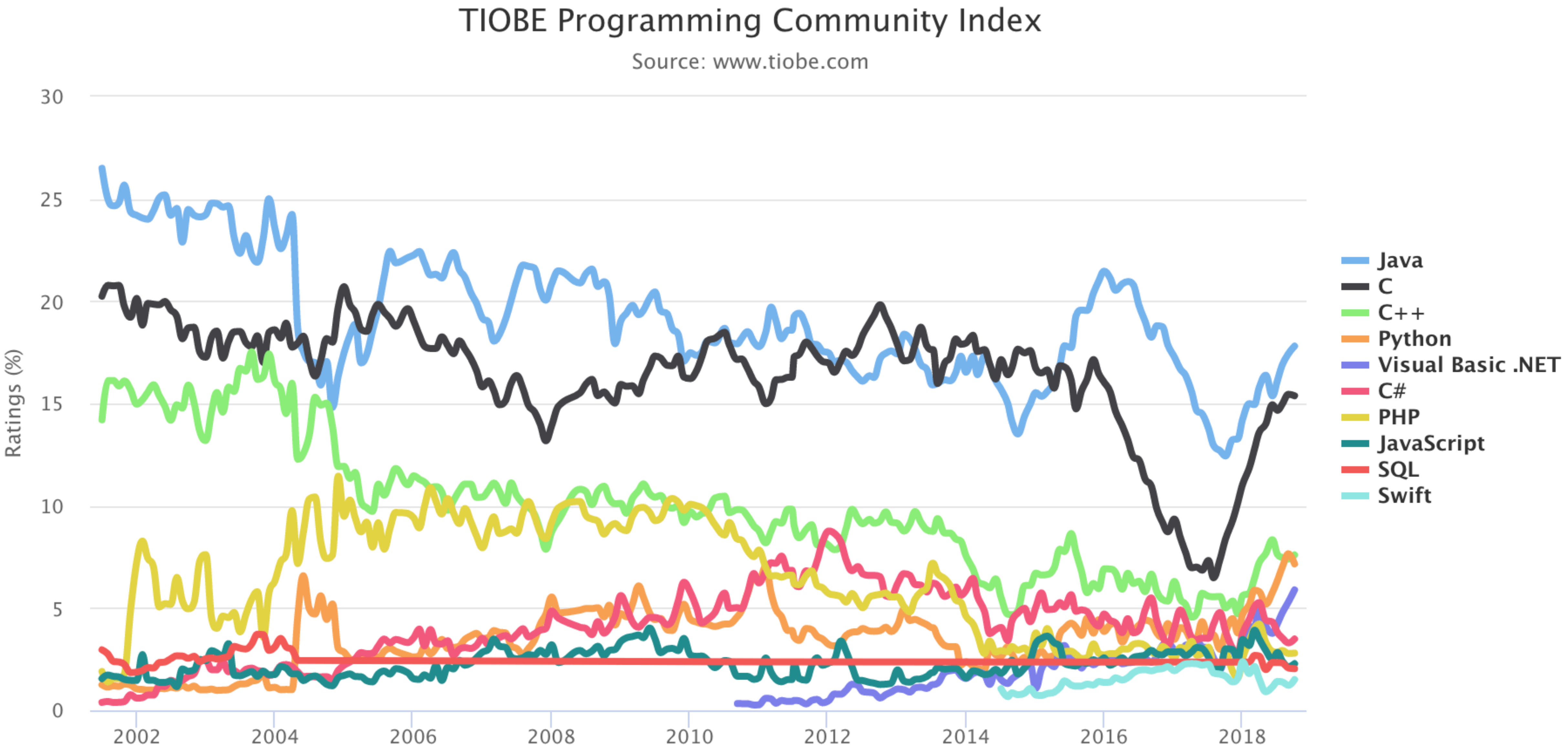
- "Text" section = **code**
- Data segment
  - global variables
  - constants
- Stack
  - contains stack frames, one per active function, grows "downwards"
  - each stack frame is used to hold data for a function activation
  - in multithreaded programs each thread has its independent stack and program counter
- Heap
  - dynamically allocated memory
  - grows "upwards"



The C language

- Buffer overflows are normally associated with the C language and “relatives” C++ and Objective-C.
- These languages (especially C and C++) are used for for implementing critical software :
    - Operating system kernels and utilities — Linux, Windows, MacOS, …
    - Core building blocks of the Internet — Apache, Webkit, OpenSSL, …
    - Embedded system programming—Arduino, ROS,micro-controller programming in general, …
    - VMs/runtime systems for other languages — Java, Python, PHP, …

Popularity of C and C++

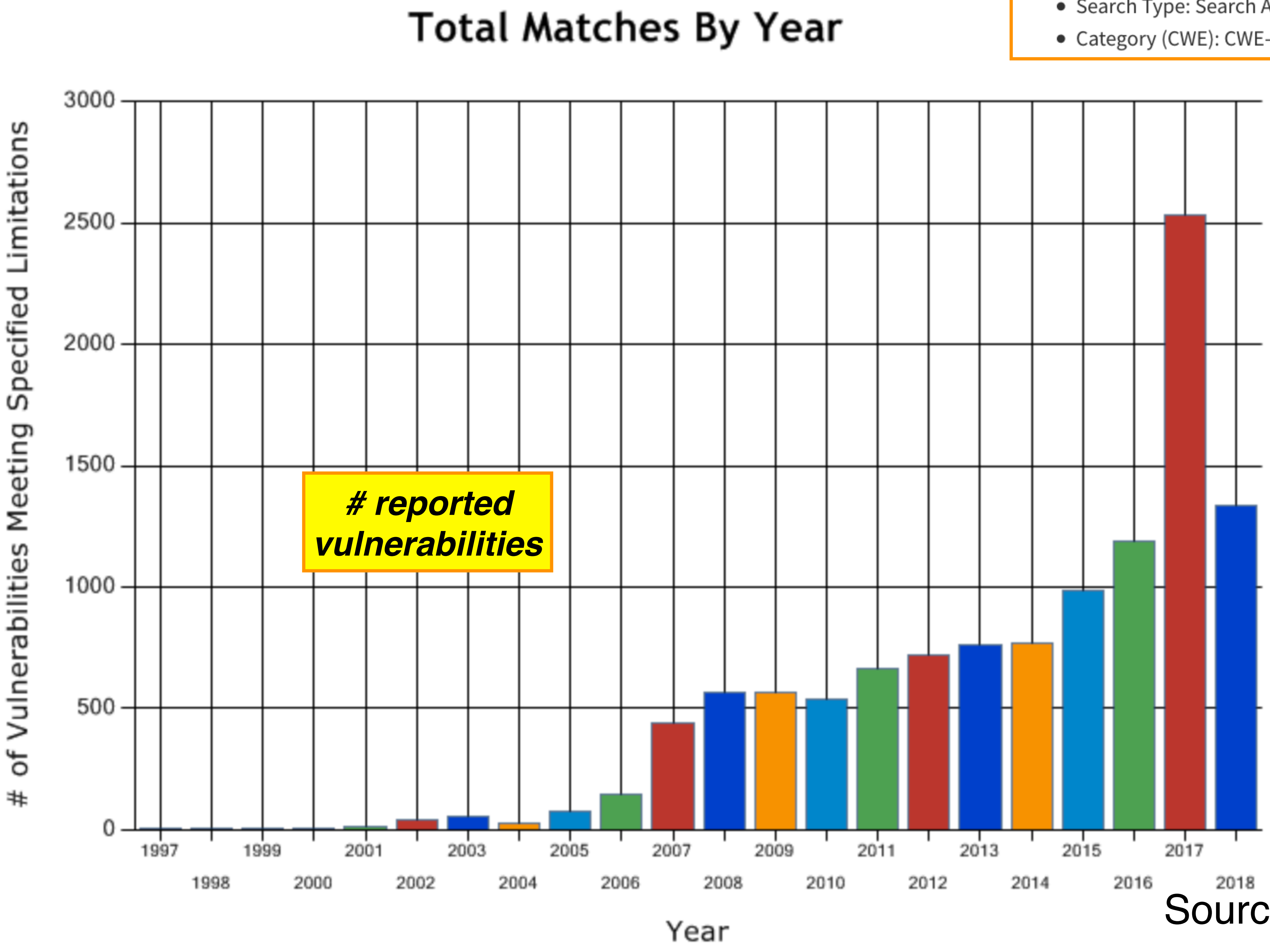


- C and C++, together with Java, have been taking the top 3 positions in the [TIOBE index](#) for programming language popularity for many years
  - The rankings are derived from search engine query statistics for programming languages

“Popularity” of buffer overflows

Search Parameters:

- Results Type: Statistics
- Search Type: Search All
- Category (CWE): CWE-119 - Buffer Errors



Total Matches By Year

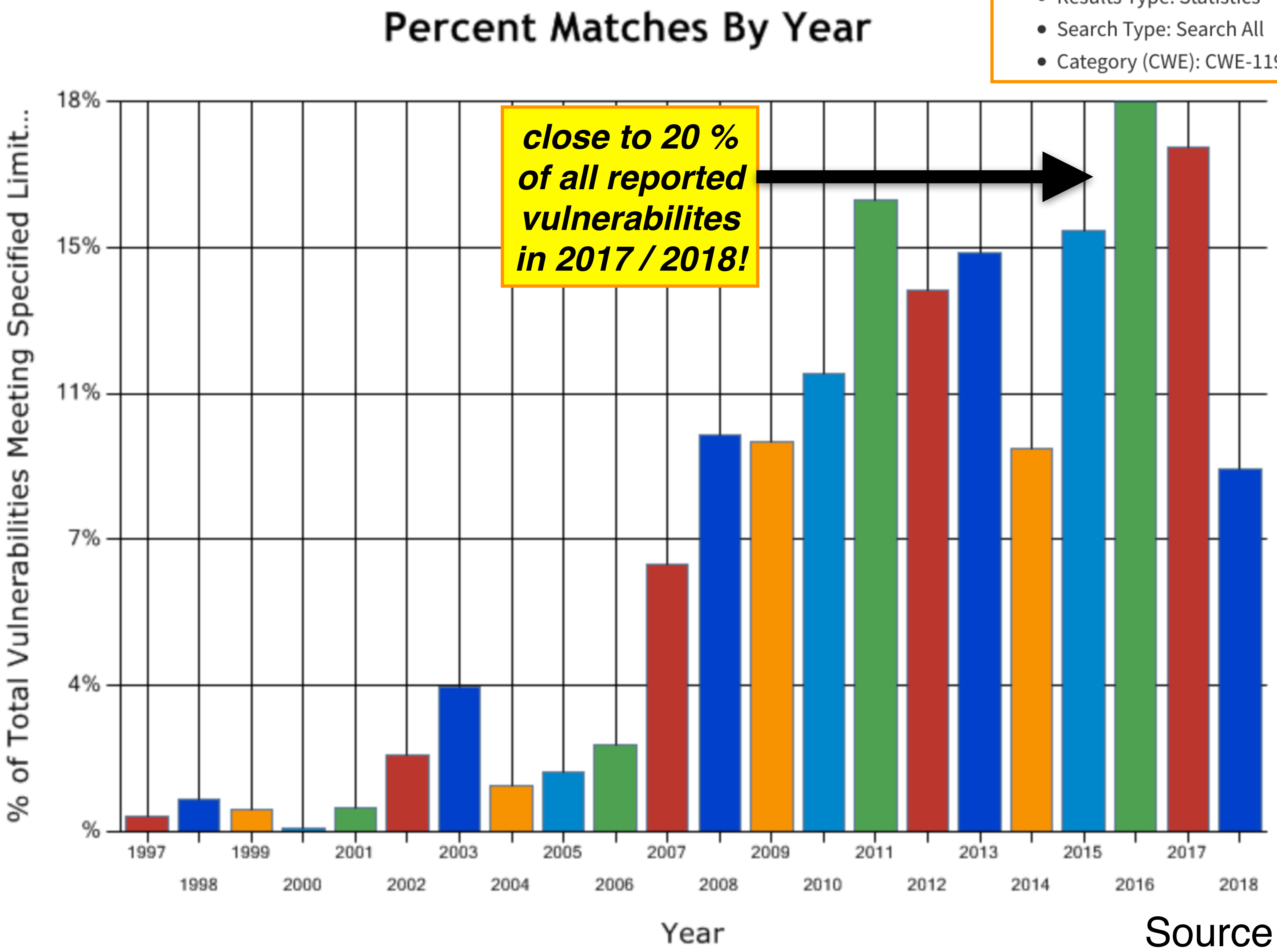
 # reported vulnerabilities

Source: NIST NVD

“Popularity” of buffer overflows (2)

Search Parameters:

- Results Type: Statistics
- Search Type: Search All
- Category (CWE): CWE-119 - Buffer Errors



close to 20 % of all reported vulnerabilites in 2017 / 2018!

Source: [NIST NVD]

C vulnerabilities

common issues &
a few examples

C vulnerabilities

- **C is not memory-safe (= buffer overflows out-of-the-box)**
  - Read/write access to out-of-bounds or logically undefined/ inacessible memory, beyond memory pertaining to program variables, that can arbitrarily affect the stack, internal heap data, …
- **C is not type-safe**
  - The types of data associated to program variables (memory locations) can be re-interpreted at will.
  - In particular arbitrary casts are allowed and unchecked.
- Programs written in memory/type-safe languages trap the execution & raise runtime errors when memory and type safety are violated.
- … but C has either “liberal” semantics or, sometimes even worse, **undefined behavior** in these cases and others

C vulnerabilities (2)

- **Undefined behavior** gives “room” for a C compiler to generate the “most convenient” code.
- In particular, behavior may differ:
  - according to the compiler in use, even for different versions of the same compiler
  - according to compiler settings - for instance optimisation settings may sometimes lead to quite unexpected / unsafe behavior!
  - depending also on the underlying processor architecture and operating system

C vulnerabilities (3)

- Dynamically-allocated memory must be explicitly managed by the programmer
  - no garbage collection
  - no built-in constructs at the language level for C: **malloc and variants** plus **free** are functions the programmer must use explictly manipulate the heap
  - C++ does have the built-in **new** and **delete** operators, but these are really equivalent to **malloc** and **free** in memory terms
- Strings are represented by null-terminated character sequences
  - many string-related functions easily lead to buffer overflows (strcpy, gets, printf, scanf, …)
  - the source of many (security) problems

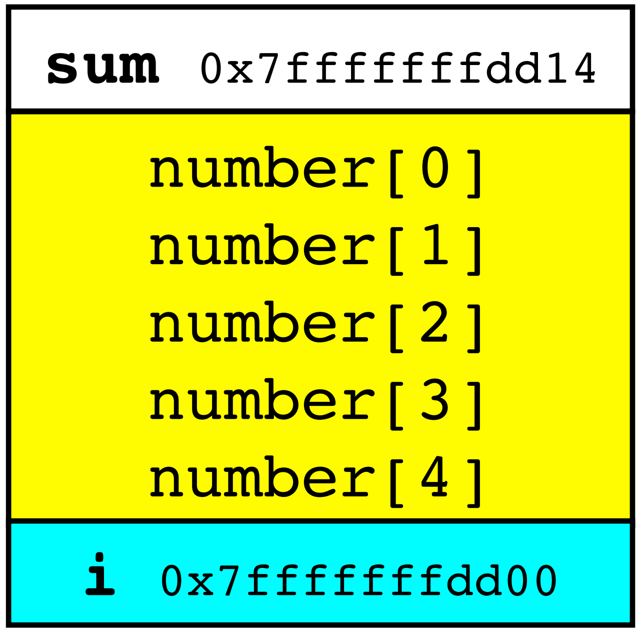
Stack overflow example

```c
#include <stdio.h>
#define N 5
int main(int argc, char** argv) {
  int sum = 0;
  int numbers[N]; // fill as { 1, 2, 3, 4, 5 }
  for (int i=0; i < N; i++)
    numbers[i] = i+1;
  for (int i = 0; i <= N; i++)
    sum += numbers[i];
  printf("Sum=%d\n", sum);
  return 0;
}
```

- A particular execution may print **`20`**, not **`15`** as expected. A small re-arrangement of variable declarations may lead to other results, but not **`15`** anyway. The code does not print **`15`**, because the second **for** loop has an "**off-by-one**" error: **i** goes from **0** up to **`N=5`**, not **`N-1=4`** ! **The expected behavior is undefined.** Analogous programs written in memory-safe languages would throw a runtime exception signalling the invalid array access (e.g. `ArrayIndexOutBoundsException` in Java).
- There is a **stack overflow** in the access to **`number,`** given that local variables are allocated in the stack. Let's see how using the **GNU debugger (gdb)** …

Stack overflow example (2)

```
$ gcc -g stack_overflow.c -o stack_overflow
$ gdb ./stack_overflow
(gdb) br 8
Breakpoint 1 at 0x40056e: file stack_corruption.c, line 8.
(gdb) r
. . .
Breakpoint 1, main (argc=1, argv=0x7fffffffde08) at
stack_overflow.c:8
8    for (int i = 0; i <= N; i++)
(gdb) p &i
$1 = (int *) 0x7fffffffdd14
(gdb) p &sum
$2 = (int *) 0x7fffffffdd1c
(gdb) p numbers
$3 = {1, 2, 3, 4, 5}
(gdb) p &numbers
$4 = (int (*)[5]) 0x7fffffffdd00
(gdb) p &numbers[5] - &i
$5 = 0
```



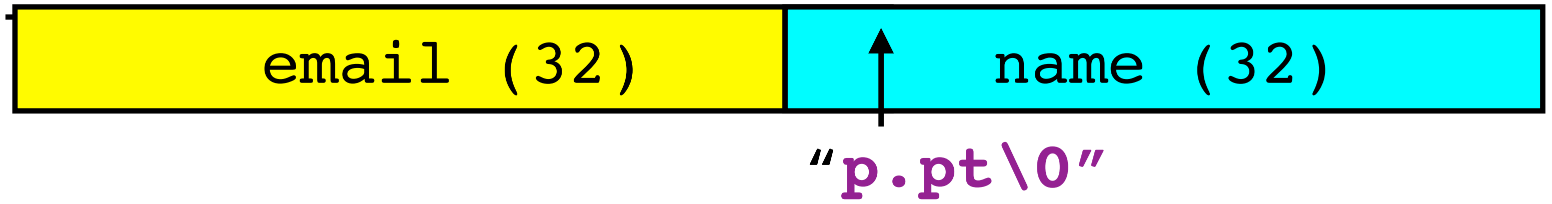
- Position **5** of `numbers` corresponds to the address of **`i`** !
- In the last iteration of the buggy **`for`** loop, **`i = 5`**, so the program will add 5 to **`sum`**, obtaining **`15+5 = 20`**

Stack overflow with string-manipulation functions

```
char name[32];
char email[32];
printf("Enter your name: ");
gets(name);
printf("Enter your email: ");
gets(email);
printf("Name: %s Email: %s\n", name, email);
```

```
Enter your name: Eduardo
Enter your email: very_long_email_I_guess@dcc.fc.up.pt
Name: p.pt Email: very_long_email_I_guess@dcc.fc.up.pt
```

stack overflow (5 bytes) !



email (32)

name (32)

“p.pt\0”

- Variables are allocated contiguously in the stack (or nearby in the general case)
- **`gets`** reads an arbitrary number of bytes until a newline, ‘\0’ or EOF is found.
- In this case, second **`gets`** call may overflow the capacity of **`email`**.

Heap-allocated memory programming errors

```
int n;  unsigned char *a, *b;
n =  . . .;
a = (char*) malloc(n);   // allocate memory for a
memset(a, 'x', n);       // set all positions to 'x'
free(a);                 // free memory
// a is now a dangling reference (to freed up memory)
b = (char*) malloc(2*n);  // allocate memory for b
printf("a == b ? %s\n", a == b ? "yes" : "no");
memset(b, 'X', 2*n);       // set all positions to 'X'
memset(a, 'x', n);         // use dangling reference, set to ‘x’
free(a);                   // double free! (and what about b?)
```

- **Use-after-free: NO !** Pointer **a** should not be used after being freed up, it becomes a **dangling reference**.
- **Free-after-use: YES !** On the other hand **b** is not freed up at the end, we will have a **memory leak** (allocated but not freed up).
- **Double-free: NO**! It is also incorrect to free **a** twice.
- **Q:** what to expect from the execution?

Numerical overflow example

```c
…
int main(int argc, char** argv) {
  long n = atol(argv[1]);
  printf("Allocating %lu (%lx) bytes for n=%ld (%lx)\n",
         (size_t) n, (size_t) n, n, n);
  char* buffer = (char*) malloc(n);
  printf("Allocated buffer: %p\n”, buffer);
  free(buffer);
  return 0;
}
```

```
$ ./integer_overflow  -1
Allocating 18446744073709551615 (ffffffffffffffff) bytes for n=-1 (ffffffffffffffff)
Allocated buffer: 0x0
```

- Integer overflow
  - **`malloc`** takes **`size_t (unsigned long)`** arguments, 64-bit unsigned integers, **`n`** is 64-bit signed integer, the argument conversion causes an overflow
  - **`malloc`** cannot allocate **`UINT_MAX=2^63-1`** bytes, hence it returns **`NULL`**
- The faults in this program are several:
  - `argc / argv[1]` not checked — program crashes without arguments
  - `atol` used to parse `argv[1]` : will return `0` on a parse error, `strtol` should be used instead
  - and if conversion is succesful (as in the example), bounds for `n` are not verified

Heap-allocated memory: dangling references & memory leaks (2)

```
$ ./dangling_reference_example 9
a - line 19 >  78 78 78 78 78 78 78 78 78
a == b ? yes
a - line 25 >  00 00 00 00 00 00 00 00 78
b - line 25 >  00 00 00 00 00 00 00 00 78 00 00 00 00 00 00 00 00 00
a - line 27 >  58 58 58 58 58 58 58 58 58
b - line 27 >  58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58
a - line 29 >  78 78 78 78 78 78 78 78 78
b - line 29 >  78 78 78 78 78 78 78 78 78 58 58 58 58 58 58 58 58 58
a - line 31 >  00 00 00 00 00 00 00 00 78
b - line 31 >  00 00 00 00 00 00 00 00 78 58 58 58 58 58 58 58 58 58
```

- In this execution, both calls to **`malloc`** yield a pointer to the same memory segment (the segment is reused after being freed up for **`a`**)
- Hence **`a`** and **`b`** end up referring to the same memory segment. Using the dangling reference (**`a`**) will necessarily corrupt the memory pointed to by **`b.`**

Lack of type safety

```
long   a = 12345678912345;
double b = 12345678912345.9;
char   c[8] = "1234567";
printf("a: %ld b: %.3lf c: \"%s\"\n", a, b, c);

// Memory and type-safe
a = (long) b; // truncation errors possible but well-defined
b = (double) a;  // long converted to double
strcpy(c,"7654321");
printf("a: %ld b: %.3lf c: \"%s\"\n", a, b, c);

 // Memory-safe but not type-safe
 a = * (long*) &b;
 b = * (double*) &main;
 strcpy(c, (char*) &a);
 printf("a: %ld b: %.3lf c: \"%s\"\n", a, b, c);
```

it “works”

```
a: 12345678912345 b: 12345678912345.900 c: "1234567"
a: 12345678912345 b: 12345678912345.000 c: "7654321"
a: 4802654590752698880 b: 0.000 c: ""
```

Lack of type safety (2)

A closer look:

```
long a = 12345678912345;
  a: addr: 0x7ffe6d109b28 | data:  59 5b ce 73 3a 0b 00 00 | 12345678912345
double b = 12345678912345.9;
  b: addr: 0x7ffe6d109b20 | data:  cd b3 b6 9c e7 74 a6 42 | 12345678912345.900
char c[8] = "1234567";
  c: addr: 0x7ffd08ba6f70 | data:  31 32 33 34 35 36 37 00 | "1234567"
a = (double) b;
  a: addr: 0x7ffe6d109b28 | data:  59 5b ce 73 3a 0b 00 00 | 12345678912345
b = a;
  b: addr: 0x7ffe6d109b20 | data:  00 b2 b6 9c e7 74 a6 42 | 12345678912345.000
strcpy(c,"7654321");
  c: addr: 0x7ffe6d109b10 | data:  37 36 35 34 33 32 31 00 | "7654321"
a = * (long*) &b;
  a: addr: 0x7ffe6d109b28 | data:  00 b2 b6 9c e7 74 a6 42 |
4802654590752698880
b = * (double*) &main
  b: addr: 0x7ffe6d109b20 | data:  55 48 89 e5 48 81 ec 90 | -0.000
strcpy(c, (char*) &a);
  c: addr: 0x7ffe6d109b10 | data:  00 36 35 34 33 32 31 00 | ""
```

NULL pointer access example

```
#include <stdio.h>

typedef struct {
  int data;
} Foo;

int flawed_function(Foo* pointer) {
  int v = pointer -> data; // dereference before check
  if (pointer == NULL) // actual check
    return -1;
  return v;
}

int main(int argc, char** argv) {
  printf("result = %d\n", flawed_function(NULL)); // What to expect?
  return 0;
}
```

- Dereferencing a **`NULL`** pointer is undefined behavior, but what do you expect / prefer from this code? Crash or no crash?
- **`NULL`** is actually **0** (only a matter of programming style to use **`NULL`**)

NULL pointer access example (2)

Using gcc 6.3 on Linux x86_64 without code optimisation:

```
$ gcc null_pointer_example.c -o null_pointer_example_no_opt
$ ./null_pointer_example_no_opt
Segmentation fault (core dumped)
```

Now enabling optimisation level 2 (-O2):

```
$ gcc null_pointer_example.c -O2 –o null_pointer_example_with_opt
$ ./null_pointer_example_with_opt
-1
```

- Compiling the program without optimisation leads to a **segmentation fault**. The execution is trapped due to access to an invalid memory segment.
- Compiling the program with optimisation leads to a "normal" execution without crash !
- Why so? We must look at the generated code.

NULL pointer access example(3)

```c
int flawed_function(Foo* pointer) {
  int v = pointer -> data; // dereference before check
  if (pointer == NULL) // actual check
    return -1;
  return v;
}

int main(int argc, char** argv) {
  printf("result = %d\n", flawed_function(NULL)); // What to expect?
  return 0;
}
```

`gcc -O2` becomes “equivalent” to

```c
int main(int argc, char** argv) {
  printf(“%d\n", -1);
  return 0;
}
```

generated code

```
subq $8, %rsp
movl $-1, %esi
movl $.LC1, %edi
xorl %eax, %eax
call printf
```

- Since **`flawed_function`** is small in size, GCC decides to inline its (intermediate representation) code within **`main`**. Given that the argument is **`NULL`**, **`pointer->data`** is undefined behavior, hence a C compiler can do whatever it pleases.
- GCC decides to treat **`v=pointer->data`** is **dead code** since according to the data flow **-1** should be returned! Under that assumption the result must “logically” be **-1 !**
- **Variations:**
  - Using **`-O2 -fno-inline`** we get the segmentation fault instead!
  - Other GCC versions may handle it differently - check the Compiler Explorer site

Common programming mistakes

- Data manipulation
  - “Off-by-one” (OBO) errors (1st example)
  - Lack of input validation, buffer/array length in particular
  - Type conversion errors
  - Bad use of pointers
  - Numeric overflows
- Use of dangerous API calls, particularly string-related functions
  - gets, printf, scanf, …
- Heap management errors
  - use-after-free, no free-after-use, double-free