

Buffer overflow vulnerabilities

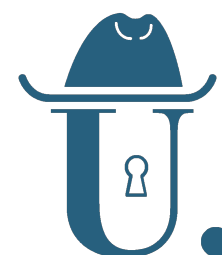
Questões de Segurança em Engenharia de Software (QSES)

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



Introduction

What is a buffer overflow?

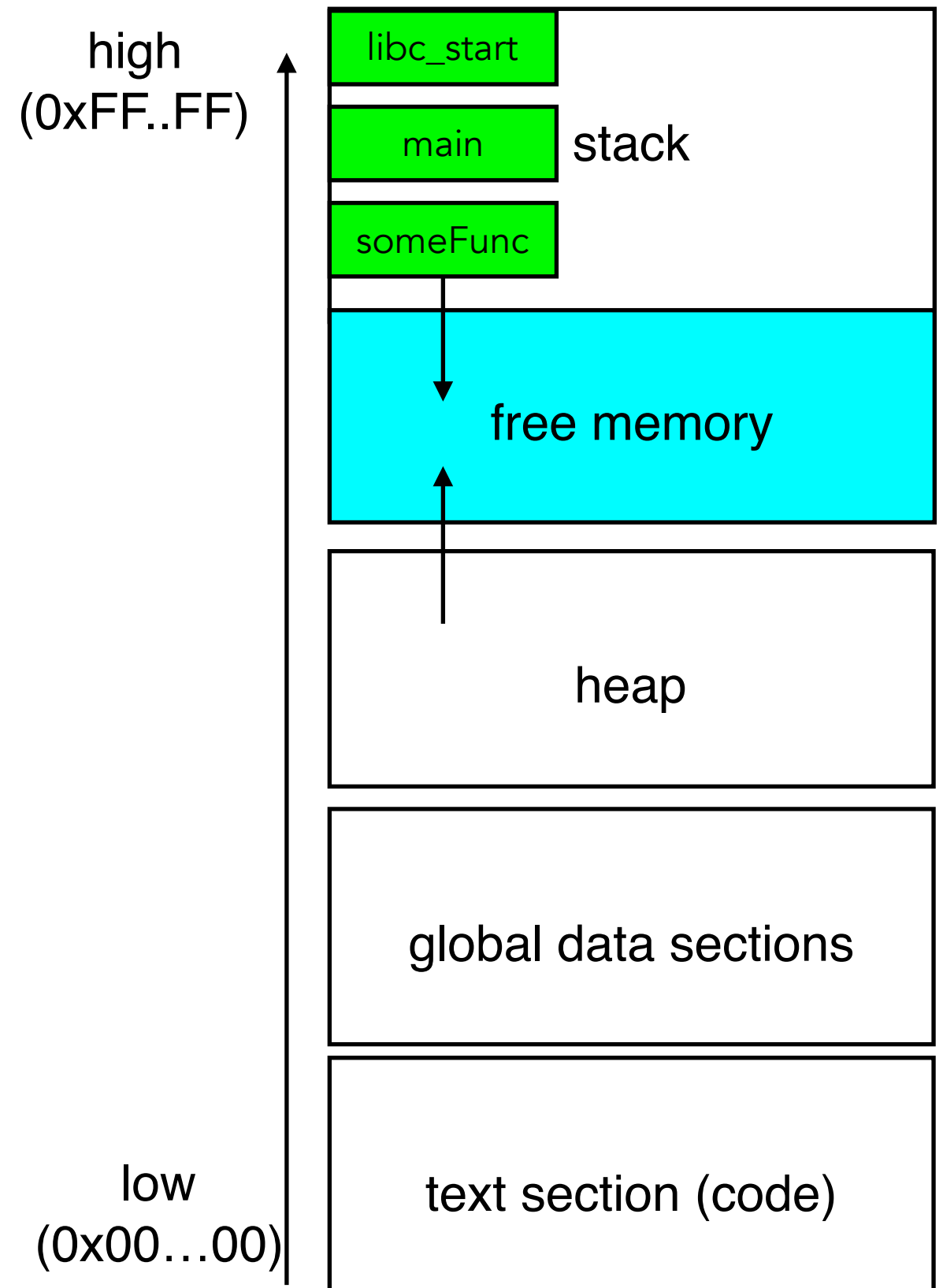
- [CWE-119](#) - Improper Restriction of Operations within the Bounds of a Memory Buffer
 - *“The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.”*
- This is a general definition for buffer overflow, 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 (proper “overflow”)
 - 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** (.text section in the [ELF](#) format)
- **Global data** sections
 - global variables (e.g. **.data** section in ELF for initialized variables, **.bss** for non-initialized variables)
 - constants (**.rodata** in ELF)
 - resolution of dynamic symbols (**.plt** and **.got** in ELF)
 - ...
- **Heap**
 - dynamically allocated memory
 - grows “upwards”
- **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



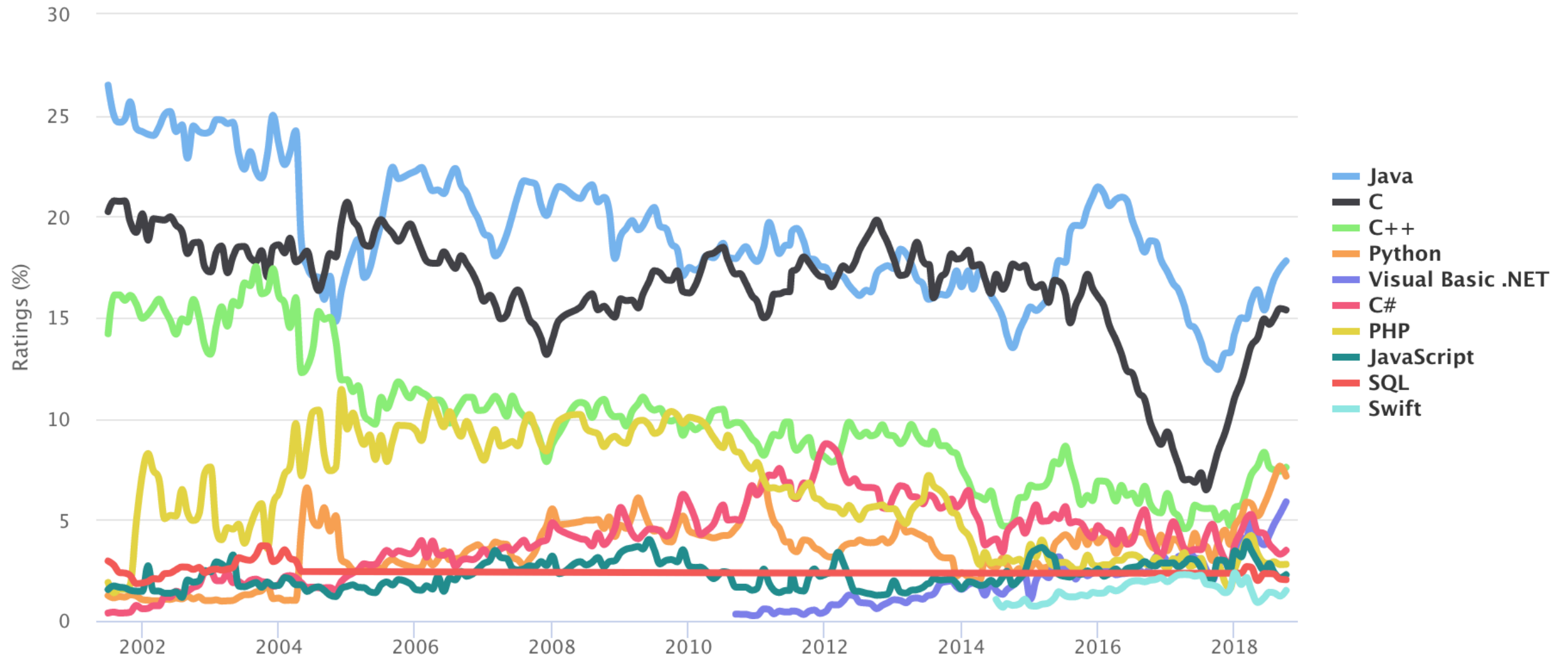
The C language

- Buffer overflows are normally associated with the C language and “relatives” (C++ and Objective-C).
- These languages are used for for implementing software such as:
 - 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++

TIOBE Programming Community Index

Source: www.tiobe.com



- C and C++, together with Java, have been taking in the top 3/4 positions in the [TIOBE index](https://www.tiobe.com) for programming language popularity for many years
 - The rankings are derived from search engine query statistics for programming languages.

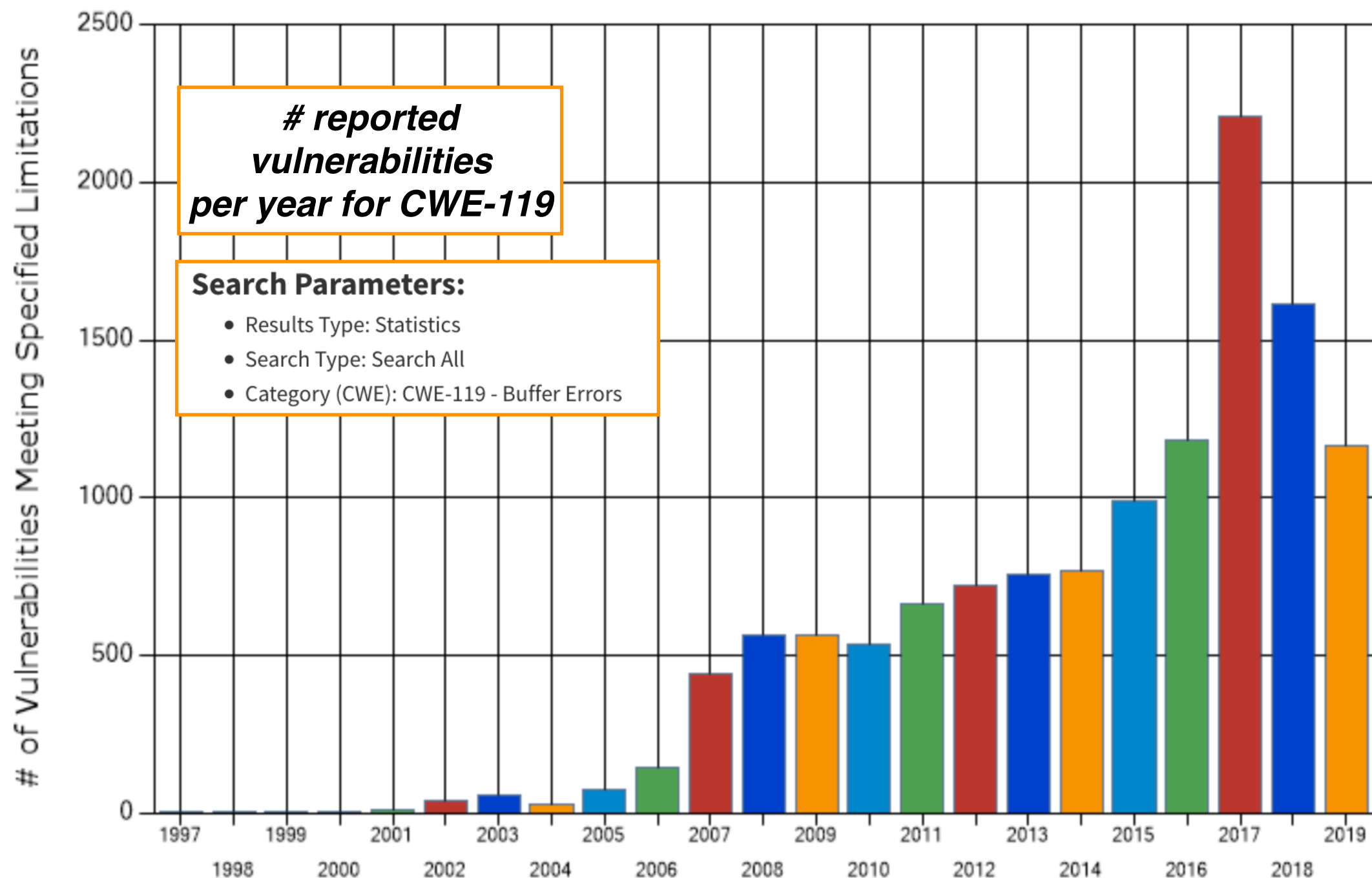
“Popularity” of buffer overflows

Source: [NIST NVD](#)

Search Parameters:

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

Total Matches By Year

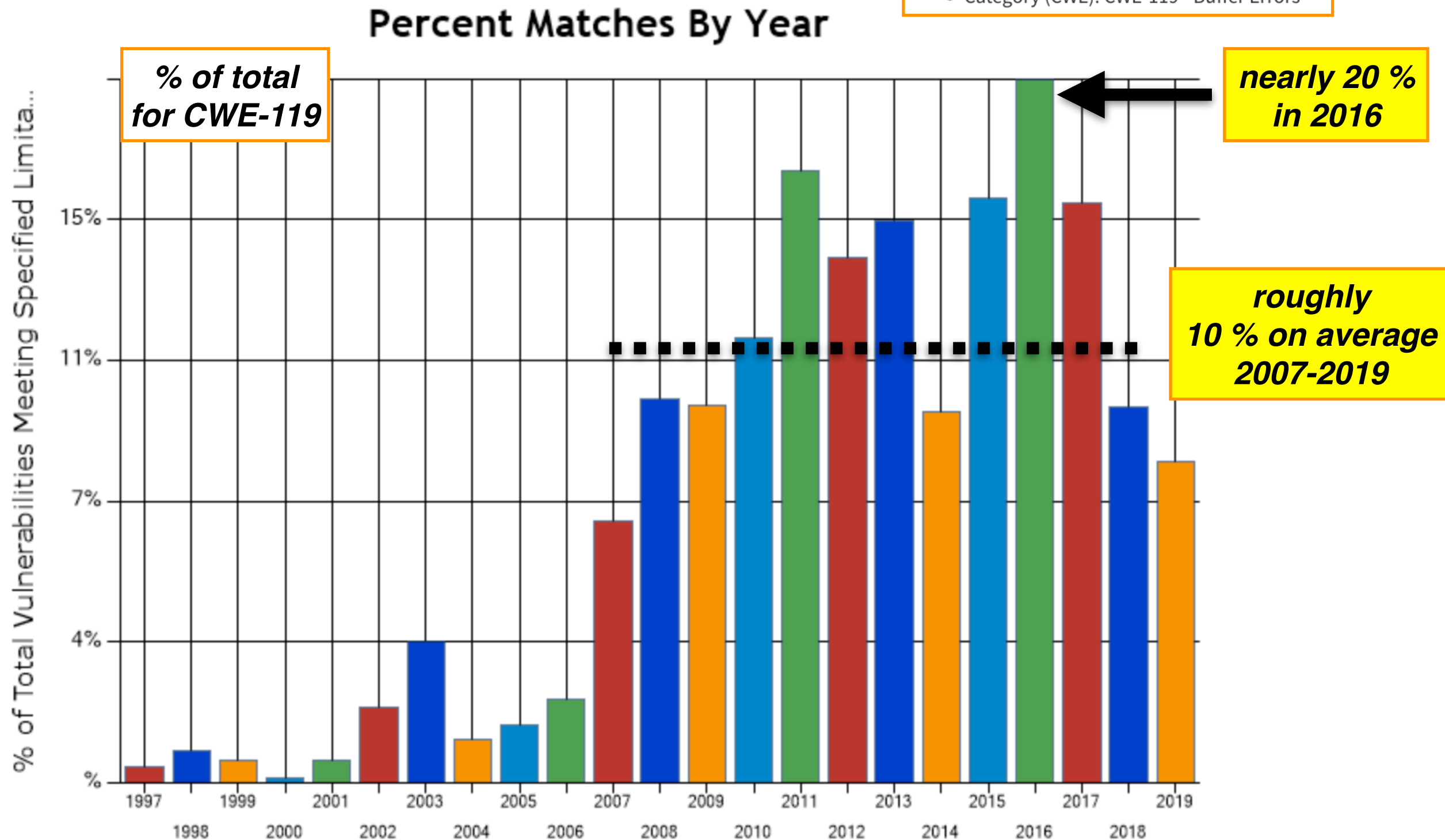


“Popularity” of buffer overflows (2)

Source: [NIST NVD](#)

Search Parameters:

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



C and memory safety

C and memory safety

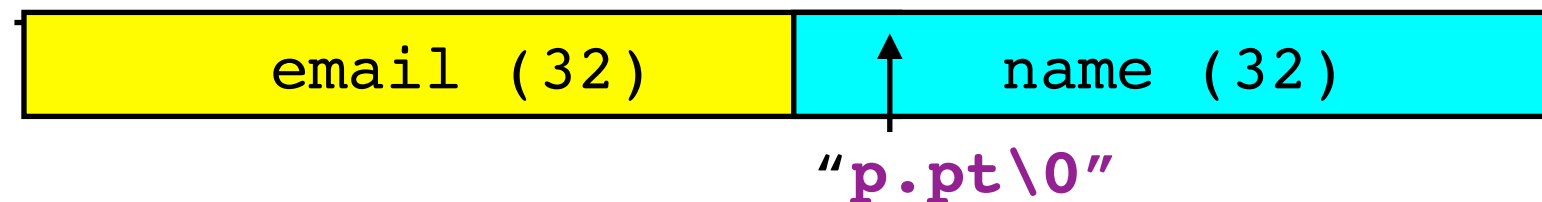
- **C makes it “easy” to access memory in invalid and unchecked manner**
 - No bounds-checking in access to buffers / pointers.
 - Arbitrary pointer arithmetic/casts are allowed
 - Dynamic memory allocation is explicitly managed by the programmer.
 - The effect of accessing invalid memory has no defined semantics (including null-pointer accesses).
- **Lack of memory safety in a program may result in:**
 - Program crash — “segmentation faults”, when protected memory is accessed.
 - Corruption of stack and/or heap, even beyond “logical program data” (e.g. stack return address, frame pointer, heap internal data, ...)
 - Non-deterministic behavior — sensitivity to runtime conditions, choice of compiler and code generation options

Stack overflow example

```
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) !



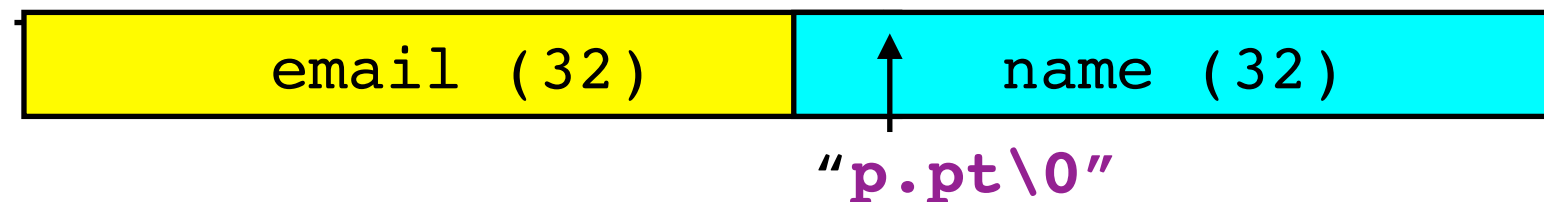
- Variables may be allocated contiguously in the stack (or nearby in the general case)
- **gets** reads an arbitrary number of bytes until a newline, '\n' or EOF is found.
- In this case, second **gets** call may overflow the capacity of **email**.

Stack overflow example (2)

```
char name[32];  
char email[32];  
strcpy(name, argv[1]);  
strcpy(email, argv[2]);  
printf("Name: %s Email: %s\n", name, email);
```

```
./stack_overflow Eduardo 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) !



- The **strcpy** function copies data onto destination buffer until **'\0'** is found.

Stack overflow (3) - off-by-one error

```
#include <stdio.h>
#define N 8
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. `ArrayIndexOutOfBoundsException` 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 — off-by-one error

```
$ 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
```

sum	0x7fffffffdd14
number[0]	
number[1]	
number[2]	
number[3]	
number[4]	
i	0x7fffffffdd00

- In this execution: 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 $(5+1)$ to `sum`, obtaining $15+6 = 21$

Heap memory management

- **Dynamically-allocated memory must be explicitly managed by the programmer**
 - no garbage collection
 - In C: **malloc** and variants (**calloc/realloc**) + **free** are functions the programmer must use to explicitly 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

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?)
// free(b) - not done - memory leak!
```

- **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?

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

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

Numerical overflow example

```
...
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=2⁶³-1** bytes, hence it returns **NULL**

■ Several vulnerabilities in the example program:

- **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

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)

```
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

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

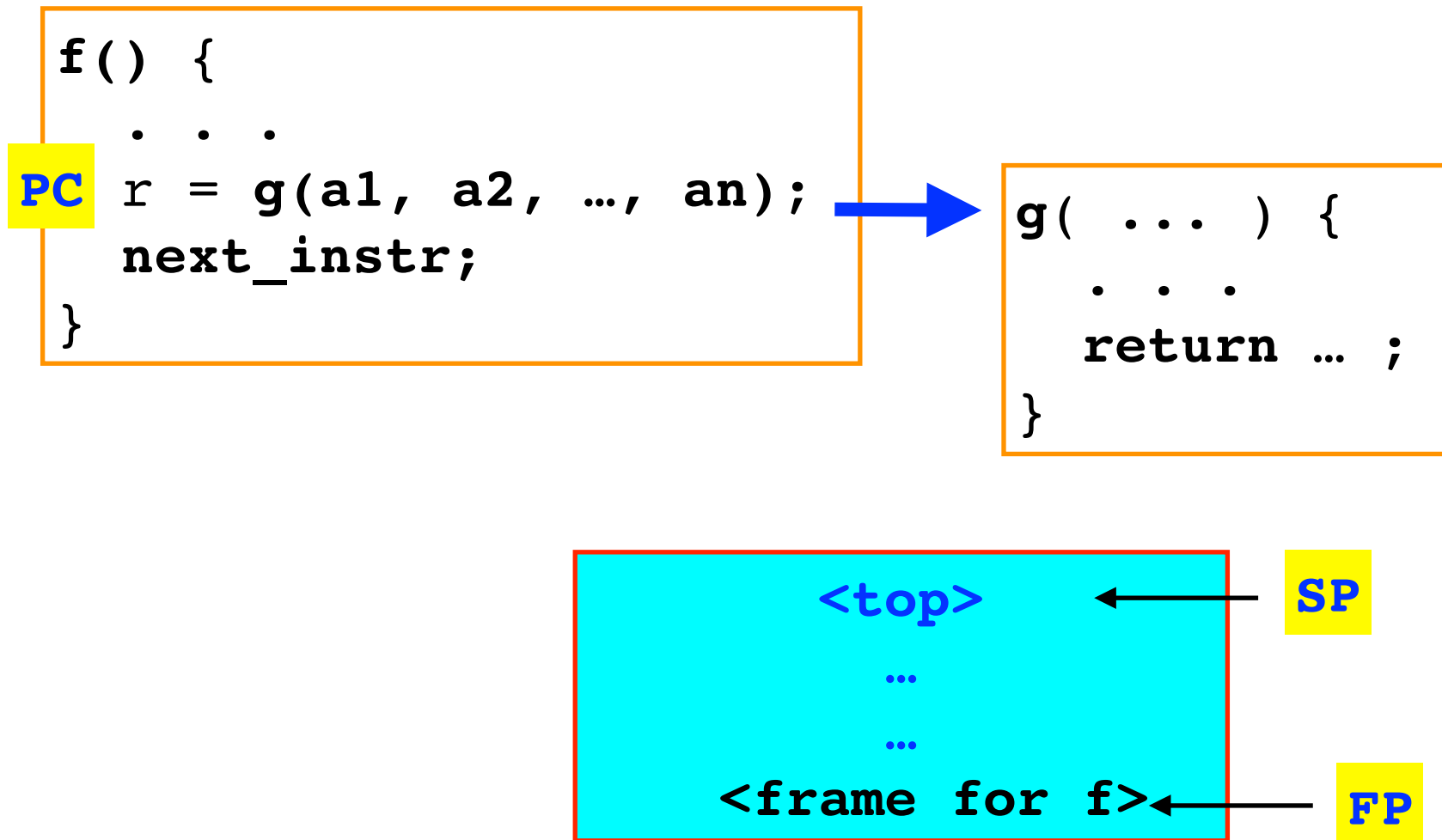
generated code

```
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

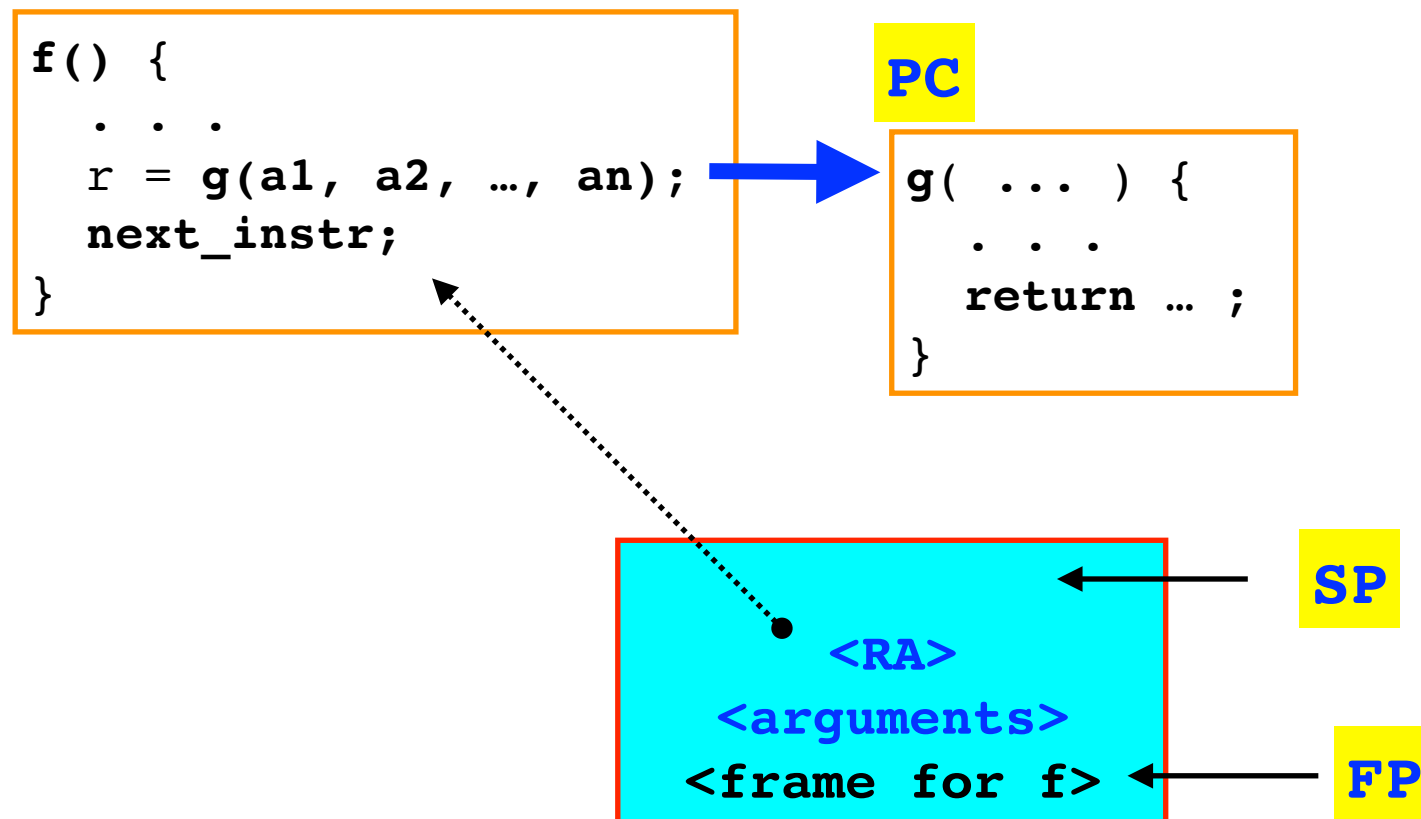
Stack-smashing attacks

Function call



- Let us describe how function calls are generally handled. Details may differ according to calling conventions and compilation options (e.g. for code protection or optimisation).
 - **PC** = program counter, the address of the currently execution instruction
 - **SP** = stack pointer, the address of the current stack location
 - **FP** = frame pointer, the base address for the currently executing function

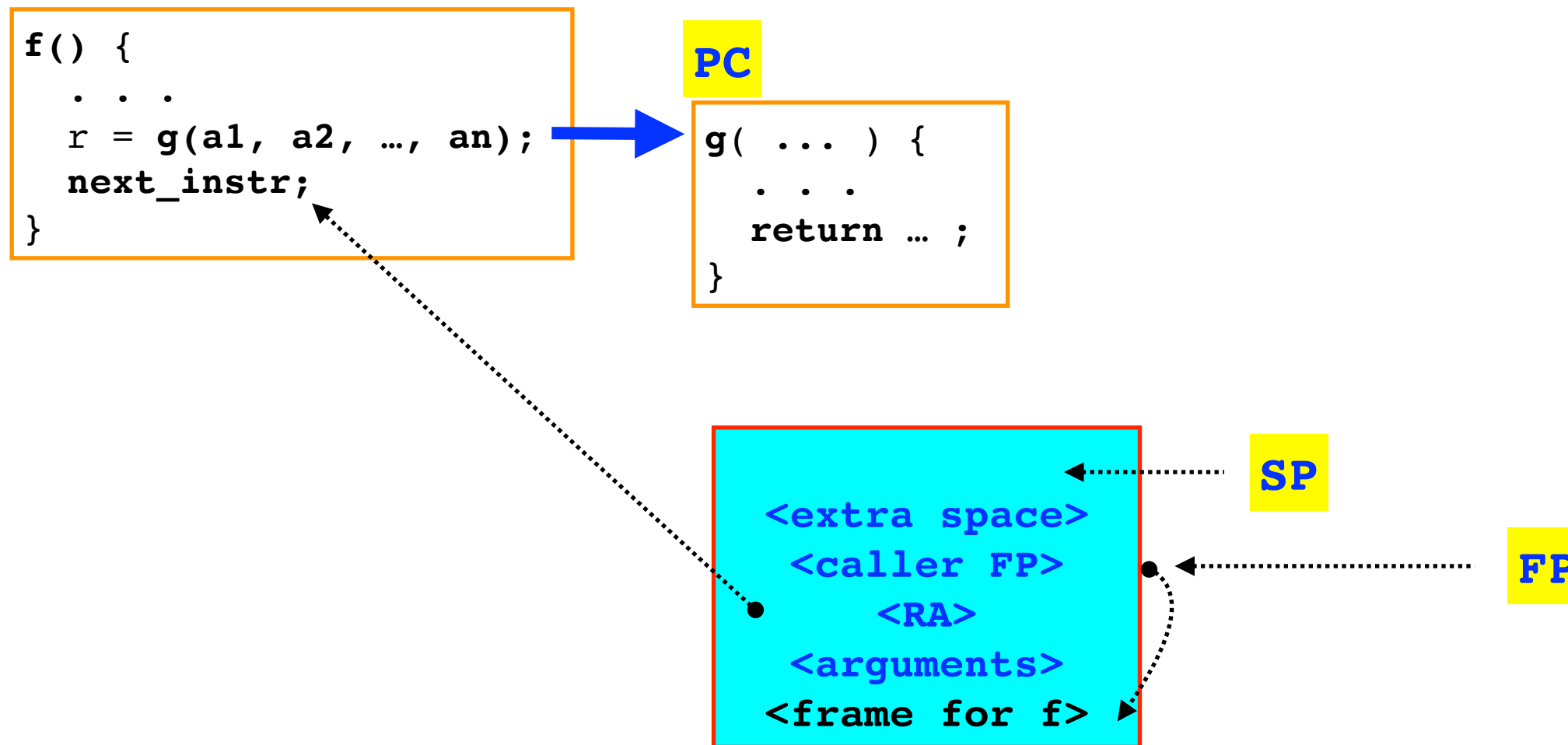
Function calls - initiation by caller



■ Calling function proceeds by:

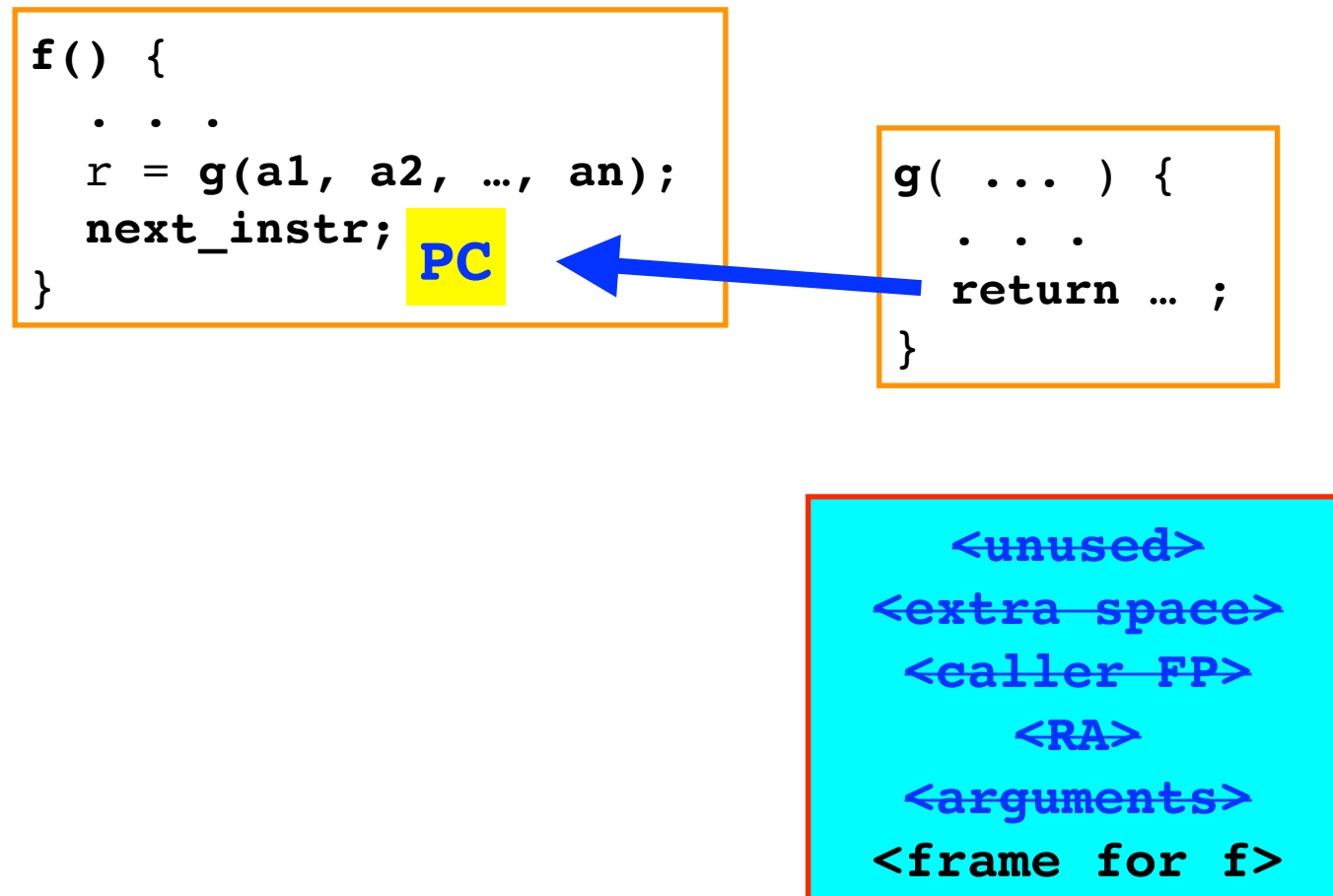
- **1) Passing arguments through registers and/or the stack.** For instance in Linux x86_64 arguments (up to some limit) are passed through registers, in x86_32 only the stack is used in most calls.
- **2) Pushing the return address onto the stack, i.e., the address of the instruction after the call (current PC + some offset).**
- **3) Branching to the called function, changing PC.**

Function calls - initiation by callee



- **On entry, called function** proceeds by:
 - **1)** Pushing the current FP (the callee's) frame pointer onto the stack.
 - **2)** Setting FP to the current stack pointer.
 - **3)** Updating SP such that the necessary space is allocated for local variables/intermediate values as needed.

Function calls - return sequence



■ On return, the callee proceeds by:

- 1) Setting stack to current FP
- 2) Popping (restores) the frame pointer from the stack (callee)
- 3) Pops the return address (callee) and returns to it (callee).
- Some calling conventions push the return value (if any) onto the stack, others use a register .

Simple x86_64 example

```
int main(int argc, char** argv) {  
    long r = foo(5, 2);  
    printf("%ld\n", r);  
    return 0;  
}
```

PC=%rbi

```
long foo(long a, long b) {  
    long s = a + b,  
        d = a - b;  
    return s * d;  
}
```

■ Relevant x86_64 registers in this example:

- **%rip** — program counter
- **%rsp** — stack pointer
- **%rbp** — frame pointer
- **%rsi** and **%rdi** are used to pass arguments (the stack is not used for arguments in this case)

Simple example — call initiation

```
int main(int argc, char** argv) {  
    long r = foo(5, 2);  
    printf("%ld\n", r);  
    return 0;  
}
```

```
int foo(long a, long b) {  
    long s = a + b,  
        d = a - b;  
    return s * d;  
}
```

```
main:  
    ...  
    movl $2, %esi  
    movl $5, %edi  
    call foo  
    ...
```

```
foo:  
    pushq %rbp  
    movq %rsp, %rbp  
    movq %rdi, -24(%rbp)  
    movq %rsi, -32(%rbp)
```

%rip

■ main:

- Uses registers pass both arguments. **%esi** and **%edi** are shorthand for the lower 32 bits of the **%rdi** and **%rsi** general-purpose registers [values 5 and 2 fit on 32-bits]
- The **call** instructions then places the RA on the stack, and updates the PC (**%rip**) to **foo**.

Simple example — call initiation (2)

```
int main(int argc, char** argv) {  
    long r = foo(5, 2);  
    printf("%ld\n", r);  
    return 0;  
}
```

```
long foo(long a, long b) {  
    long s = a + b,  
        d = a - b;  
    return s * d;  
}
```

```
main:  
    ...  
    movl $2, %esi  
    movl $5, %edi  
    call foo  
    ...
```

```
foo:  
    pushq %rbp  
    movq %rsp, %rbp  
    movq %rdi, -24(%rbp)  
    movq %rsi, -32(%rbp)
```

■ foo:

- Saves the FP (**%rbp**) onto the stack (**%rsp**), before resetting it to the current SP (**%rbp**).
- Pushes the arguments (**%rdi** and **%rsi**) onto the stack for convenience in later processing.

Simple example — return sequence

```
int main(int argc, char** argv) {  
    long r = foo(5, 2);  
    printf("%ld\n", r);  
    return 0;  
}
```

```
long foo(long a, long b) {  
    long s = a + b,  
        d = a - b;  
    return s * d;  
}
```

```
main:  
    ...  
    movl $2, %esi  
    movl $5, %edi  
    call foo  
    ...
```

```
foo:  
    ...  
    imulq -16(%rbp), %rax  
    popq %rbp  
    ret
```

■ On return, **foo**:

- Places the result on **%rax** — `imulq ..., %rax`
- Pops the FP (of main) from the stack — `popq %rbp`
- Pops the return address from the stack and returns — `ret`

Simple example — illustration with gdb

```
Breakpoint 1, main (argc=1,
argv=0x7fffffff5f8) at stack_test.c:
10
10  long r = foo(5, 2);
(gdb) p $rbp
$1 = (void *) 0x7fffffff510
(gdb) p $rsp
$2 = (void *) 0x7fffffff4f0
(gdb) p $rip
$3 = (void (*)( )) 0x400574 <main+15>
(gdb) s
```

saved
FP

return address

```
#0  0x00000000400583 in main (argc=1,
argv=0x7fffffff5f8) at stack_test.c:
10
10  long r = foo(5, 2);
(gdb) p $rip
$10 = (void (*)( )) 0x400583 <main+30>
(gdb) p $rbp
$11 = (void *) 0x7fffffff510
```

```
Breakpoint 2, foo (a=5, b=2)
at stack_test.c:4
4  long s = a + b,
(gdb) p $rbp
$4 = (void *) 0x7fffffff4e0
(gdb) p $rsp
$5 = (void *) 0x7fffffff4e0
(gdb) p $rip
$6 = (void (*)( )) 0x400539 <foo+12>
(gdb) p *(void**) $rbp
$7 = (void *) 0x7fffffff510
(gdb) p *(void**) ($rbp+8)
$8 = (void *) 0x400583 <main+30>
(gdb) n
5      d = a - b;
(gdb) n
6  return s * d;
(gdb) p $rip
$9 = (void (*)( )) 0x40055a <foo+45>
(gdb) ret
Make foo return now? (y or n) y
```


Stack smashing attacks — assumptions

- Let us assume for now that;
 - we can perform a buffer overflow on the stack without any protection in place
 - we can place executable code on the stack
 - memory addresses are predictable
- Provided the program has a vulnerability of “interest”, we can think of a **stack-smashing attack**.
- **Idea** — overflow the stack frame of a function such that:
 - malicious code is placed on the stack, and the return address is changed to point to it
 - hence, on function return, the malicious code gets executed

A simple example

```
#include <stdio.h>
int main(int argc, char**argv) {
    char name[128];
    printf("What's your name?\n");
    gets(name);
    printf("Hello %s!\n", name);
    return 0;
}
```

normal execution

```
$ ./hello.bin
What's your name?
Eduardo
Hello Eduardo
```

- Simple “hello” program that:
 - calls a **gets** operation to read a string onto buffer **name**
 - then prints “Hello <username>\n” using 3 **printf** calls

A simple example (2)

```
#include <stdio.h>
int main(int argc, char**argv) {
    char name[128];
    printf("What's your name?\n");
    gets(name);
    printf("Hello %s!\n", name);
    return 0;
}
```

compiler warning!

```
hello.o: In function `main':
hello.c:(.text+0x1a): warning: the `gets' function is
dangerous and should not be used.
```

- Compiler warns us that the **gets** “is dangerous and should not be used”!

A simple example (2)

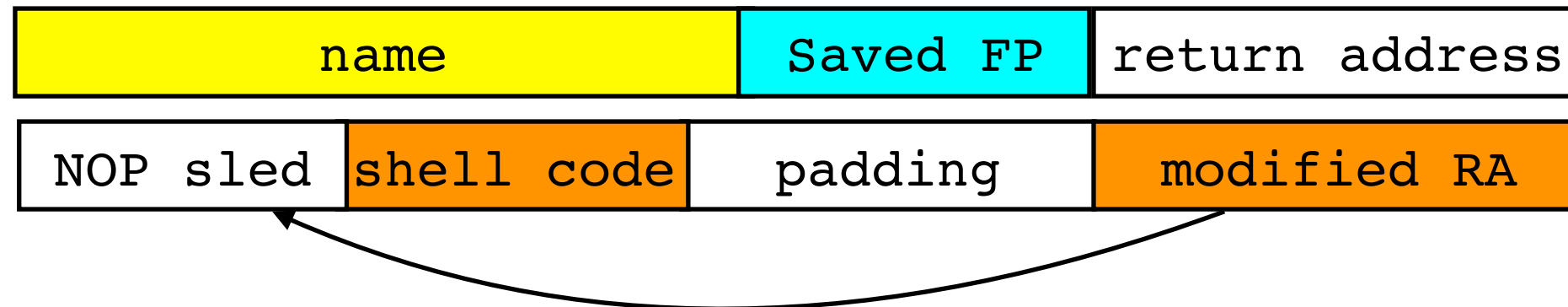
```
#include <stdio.h>
int main(int argc, char**argv) {
    char name[128];
    printf("What's your name?\n");
    gets(name);
    printf("Hello %s!\n", name);
    return 0;
}
```

execution with crash

```
What's your name?
1234567890123456789012345678901234567890
Hello 12345678901234567890...12345678901234567890
Segmentation fault (core dumped)
```

- `gets` call easily leads to a buffer overflow
 - `gets` will read input until a newline (`\n`), doing so without internal information of the size of the input buffer; `gets` receives a pointer to the buffer, not the buffer length information
 - the buffer overflow may causes a crash (“segmentation fault”)

Stack smashing attack — outline



- Call to `gets` may be exploited with malicious input that:
 - fills the buffer with code with a NOP sled (sequence of NOPs) plus “shell code” to open a system shell
 - NOP sled is useful because we may only know the whereabouts of `name` approximately.
 - modifies the return address of `main` to jump to the NOP sled and then in sequence execute the “shell code” instructions.
- **Shell code?** Easy to obtain online.
- **Challenge:** overwrite the RA with the address of `name` var (or approximately) ?

Some famous attacks

- [Morris Worm](#) (1990)
 - “Accidental” attack caused DoS brought down much of the (then-small) Internet
 - More info here: [“The Internet Worm Program: An Analysis”](#), E. H. Spafford (page 9 for stack-based overflow details)
 - Named after [Robert T. Morris](#), convicted at the time. He is now a professor at MIT !
- Other famous attacks:
 - [Code Red worm](#)
 - [SQL Slammer](#)
- Interesting historical account (until 2009): [“Memory Corruption Attacks The \(almost\) Complete History”](#), Haron Meer, Black Hat USA 2010



Robert T. Morris

In 1990, Robert T. Morris (then 25 years old) was the first person convicted under the “Computer Fraud and Abuse Act” after he created the first worm to have a major effect on real-world computer systems. It brought down much of the Internet and related networks for days. Morris apologized in 2008, saying he'd sought to estimate the Internet's size, not cause harm.

Example shell code

```
// Goal is to execute execve("/bin/sh", ["/bin/sh", 0], 0)
// We need to set rax = 0x3b, rsi = ["/bin/sh", 0], rdx = 0
section .text
    global _start
_start:
    xor     rdx, rdx      # rdx = 0 (3rd parameter)
    mov     qword '///bin/sh', rbx # prepare 1st argument
    shr     $0x8, %rbx    # shift 8 bits => "/bin/sh\0"
    push    rbx           # push "/bin/sh\0" to the stack
    mov     rsp, rdi      # get it on rdi (1st parameter)
    push    rax           # push 0 (2nd array argument referenced by rsi)
    push    rdi           # push "/bin/sh\0" (1st array argument)
    mov     rsp, rsi      # point rsi (2nd argument) to the stack pointer
    mov     $0x3b, al     # low 8 bits of rax - code for execve syscall
    syscall
```

4831d248bb2f2f62696e2f736848c1eb08534889e750574889e6b03b0f05

- **Size:** only 30 bytes.
- Carefully crafted not to contain null (0) values. **Q:** Why?
- Source (**my comments in bold**): <http://shell-storm.org/shellcode/files/shellcode-603.php>

Other attacks

■ **Successful attack may depend on the absence of memory protections we will refer to next:**

- NX/DEP protection data — the stack is executable
- Stack protections (canaries) being disabled!
- ASLR disabled — addresses are predictable on every run

■ **return-to-libc attacks:**

- when stack is not executable, try to change return address to interesting libc code, e.g. a call to `system`
- Easy on some platforms that only use the stack to pass arguments (e.g. Linux/x86-32)

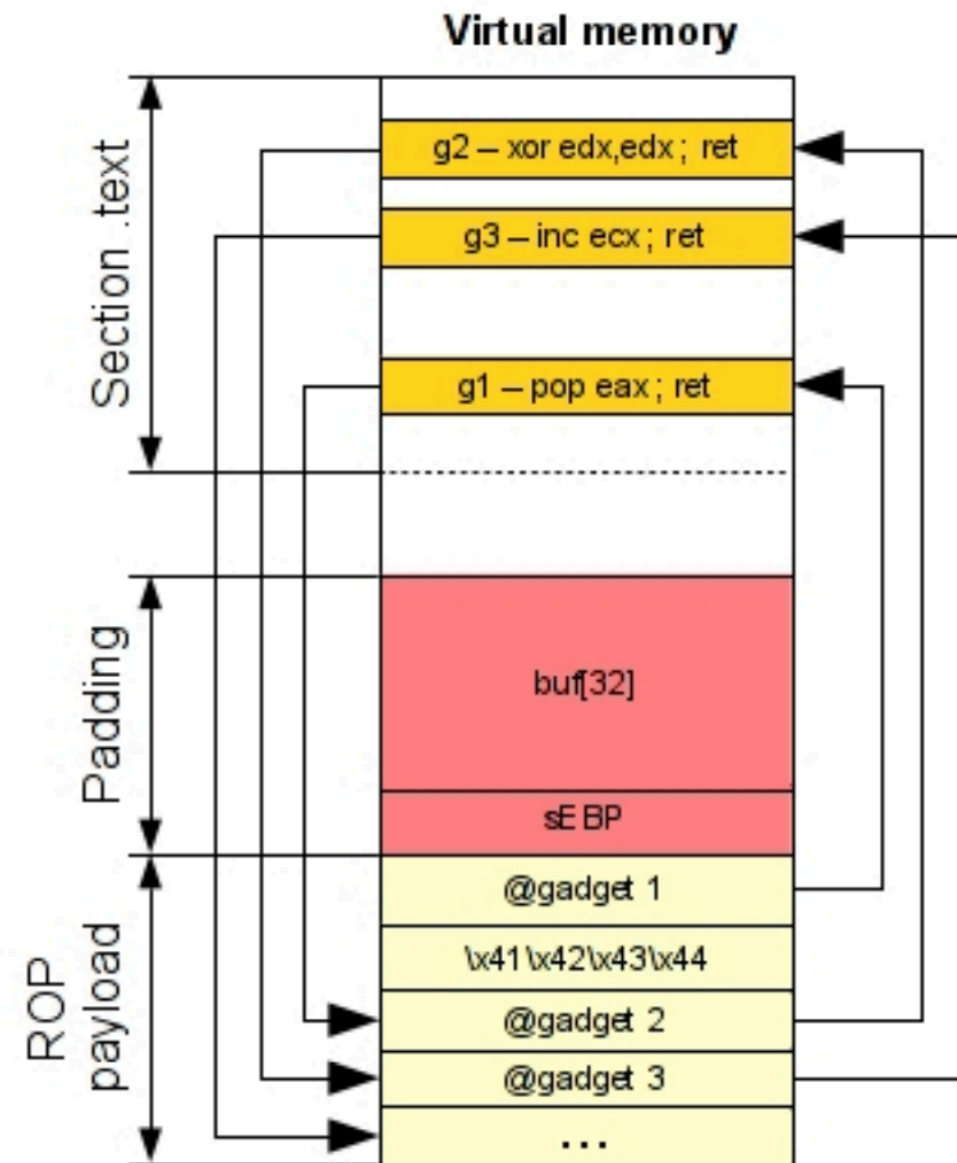
■ **ROP chains**

- ROP chains manipulate the stack (but do not execute code on it) to execute small code fragments (“gadgets”) in a chain with malicious purpose.
- Gadgets are collected from code that is marked as executable, for instance glibc fragments.

ROP chains — illustration

- Gadget1 is executed and returns
- Gadget2 is executed and returns
- Gadget3 is executed and returns
- And so on until all instructions that you want are executed
- So, the real execution is:

```
pop    eax
xor     edx, edx
inc     ecx
```



- Source: “[An introduction to the Return Oriented Programming and ROP chain generation](#)”, J. Salwan, Univ. Bordeaux
- See also: “[Return-Oriented Programming: Systems, Languages, and Applications](#)”, Roemer et al., ACM TISSEC, 2012

Variation

```
#include <stdio.h>
int main(int argc, char**argv) {
    char name[32];
    gets(name);
    printf("Hello ");
    printf(name);
    printf("\n");
    return 0;
}
```

execution leaking information in the stack

What's your name?

%p %p %p

Hello 0x400720 0x7ffff7dd59e0 0x206f6c6c

- **CWE-134**: “Use of Externally-Controlled Format String” , commonly known as **format-string vulnerability!** We introduce a “format string” for name! The **printf** call looks up the arguments for “print-out” even if there are really none, causing memory to be dumped and possibly overwritten.
- Information disclosure of memory contents itself may be helpful for stack-smashing attack.
- But printf may also write onto the stack (%n modifier) — see for instance “[Exploiting Format String Vulnerabilities](#)”, by “scut” and “team teso”, 2001

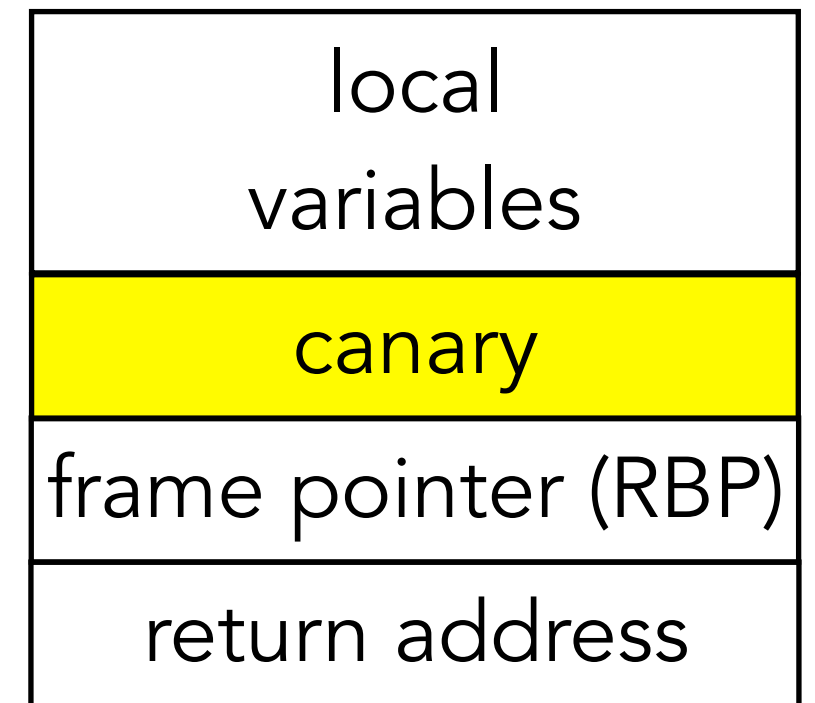
Handling buffer overflows

memory protections

Memory protections

- Prevention of buffer overflows
 - Use of stack canaries
 - Data execution prevention /non-executable flag (DEP/NX)
 - Address Space Layout Randomization (ASLR)

Canaries



- **Stack corruption detection**

- Protect the stack with a canary value.
- On return, canary is checked causing termination if value differs.

- **It does not protect against local variable overriding!**

- **Mechanism can be defeated if canary is known or can be guessed**

- Canary is constant :) or generated with a PRNG that is weak or whose seed can be guessed. Cryptographic-strength PRNG makes this harder

- **... or if attacker finds a way to determine the canary's position and read its value from the stack.**

- **Performance overhead**

- extra code required per function call, even if compiler tries to be smart / developer has a choice of options, e.g. e.g. GCC has several [-fstack-protector-XXXX flavors](#) (see next slide)

- There are memory protections that can enabled **for the heap too**, e.g, also [in GCC](#)

Stack protections — GCC

- Our examples have been compiled so far using the **-fno-stack-protector** switch, that disables stack canaries.
 - Older **GCC** versions (e.g. tested on 5.3) doesn't really require the switch, as it does not emit code for stack canaries. Recent versions (e.g. 7.3) do so by default. Recent versions of the **clang** compiler also do.
- Some GCC stack protection settings (also typically accepted in clang):
 - **-fstack-protector**: stack protection added for “*vulnerable objects*”, including “*functions that call alloca and functions with buffers larger than 8 bytes*” (from the GCC 7.3 manual)
 - **-fstack-protector-strong**: “*includes additional functions to be protected*”, e.g. “*those that have local array definitions*”
 - **-fstack-protector-all**: protects all functions
 - **-fstack-protector-explicit**: “only protects those functions which have the `stack_protect` attribute.

Example stack protection code generated by GCC (5.3)

gcc -fstack-protector ...

main:

On entry

pushq %rbp

movq %rsp, %rbp

subq \$144, %rsp

movq %fs:40, %rax # Canary value onto rax

movq %rax, -8(%rbp) # pushed onto the stack

...

On exit

movq -8(%rbp), %rdx # pops canary location

xorq %fs:40, %rdx # compare with original value

je .L3

call __stack_chk_fail # stack check failed

.L3:

leave # normal return

ret

Data execution prevention (DEP)

- Our code has also been compiled with the **-z execstack** switch, passed on to the GNU program linker (ld)
 - This lets data the stack and heap segments be executable.
 - The NX (non-executable) bit is set for these memory segments.
- Provided canaries can be defeated, return-to-libc / ROP attacks are feasible.

Address-space layout randomisation (ASLR)

■ ASLR

- OS randomly arrange positions of key areas in the memory layout (stack, heap, data, code) including library code.
- Addresses of variables, functions are different on every run of a program.
- This applies to a program but also possibly linked libraries to libc address functions.

■ We disabled ASLR (in Linux) by setting the value in `/proc/sys/kernel/randomize_va_space` to 0.

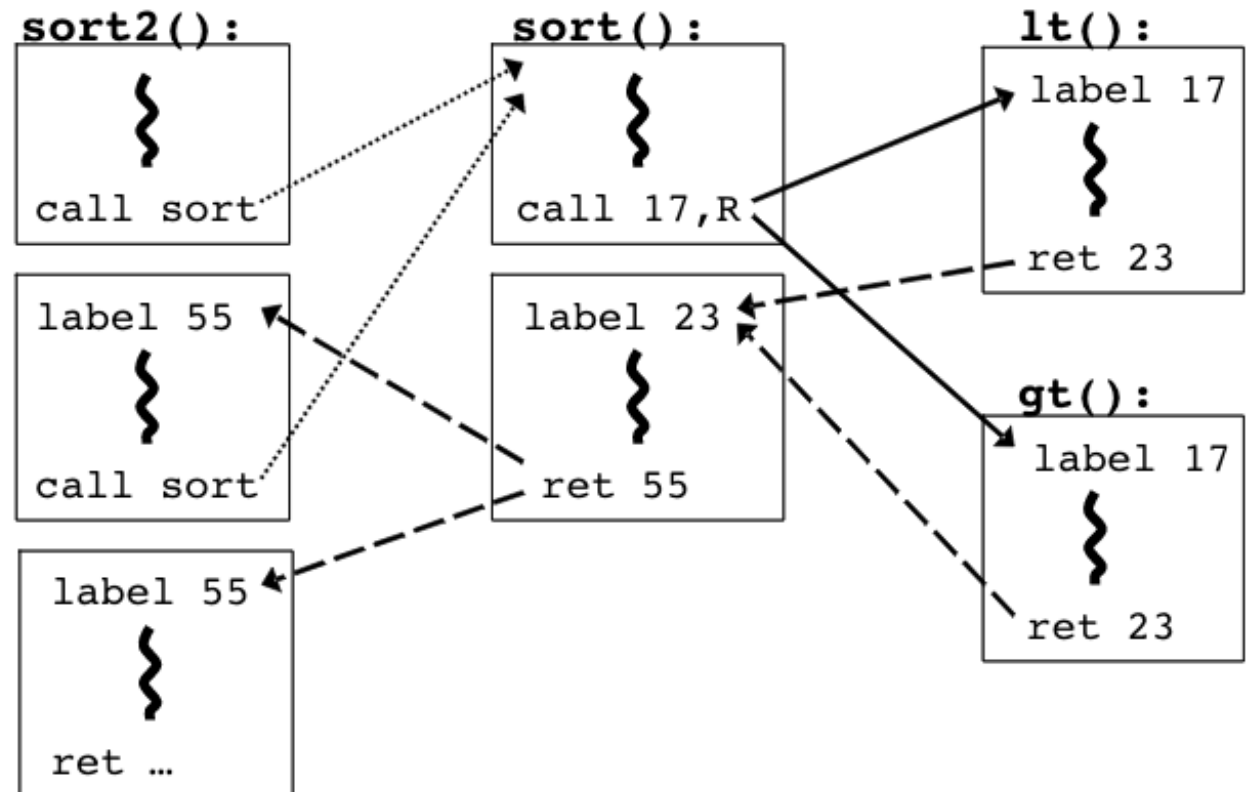
■ Adversary cannot rely on fixed memory layout, but it may use leaks and gamble on relative addresses (e.g. for stack data).

Control flow integrity

- Canaries/NX-bits/ASLR are mechanisms for trying to **detect / defeat** control-flow hijack.
- **Control flow integrity** seeks to **ensure** that the control flow of a program is (*really*) as expected:
 - **Functions:** if **f** calls **g** at instruction **I** then when **g** returns execution resumes (the PC is restored) in **f** at instruction **I+1**.
 - **More generally:** each control branch taken during program execution (not just function calls/returns) corresponds to the program's intended behavior.
- CFI schemes work by:
 - **determining possible branches statically**, e.g., according to individual procedure CFGs, call-graphs.
 - instrumenting code to **verify branches during execution** are as expected

Control flow integrity (2)

```
bool lt(int x, int y) {  
    return x < y;  
}  
bool gt(int x, int y) {  
    return x > y;  
}  
  
sort2(int a[], int b[], int len)  
{  
    sort( a, len, lt );  
    sort( b, len, gt );  
}
```



From: "[Control Flow Integrity: Principles, Implementations, and Applications](#)", M. Abadi et al. , CCS 2005

■ CFI instrumentation scheme - overview:

- Maintain a "shadow stack" to monitor control flow through CFI IDs.
- Branch target locations have associated CFI IDs.
- Branch instructions push the ID of their target onto the "shadow stack", that is checked at the branch target.

Other protections

- “Fortified” source code in libraries
 - Idea: fortify security-sensitive library calls.
 - We’ll have a brief look at GCC/GLIBC’s `_FORTIFY_SOURCE` flag.
- **Runtime sanitizers** — useful during development
 - **Idea:** monitor program execution to detect errors and possibly trap execution.
 - Two example gcc/clang sanitizer plugins: Undefined Behavior Sanitizer, and Address Sanitizer

The glibc `_FORTIFY_SOURCE` flag

`_FORTIFY_SOURCE` (since glibc 2.3.4)

Defining this macro causes some lightweight checks to be performed to detect some buffer overflow errors when employing various string and memory manipulation functions (for example, `memcpy(3)`, `memset(3)`, `strcpy(3)`, `strncpy(3)`, `strcat(3)`, `strncat(3)`, `sprintf(3)`, `snprintf(3)`, `vsprintf(3)`, `vsnprintf(3)`, `gets(3)`, and wide character variants thereof).

- We may employ glibc's `_FORTIFY_SOURCE`.
- During compilation:
 - Signals buffer overflows over variables with size known at compile-time.
- During execution:
 - Performs runtime checks that also try to detect buffer overflows.
- Let us take a look at an example from “[Enhance application security with FORTIFY_SOURCE](#)”, Siddharth Sharma, Red Hat blogs

glibc's `_FORTIFY_SOURCE` flag (2)

```
// Known size for both source and destination
char buffer[5];
. . .
strcpy(buffer, "deadbeef");
```

```
$ gcc -D_FORTIFY_SOURCE=1 -O fortify_test.c -o fortify_test
In file included from /usr/include/string.h:635:0,
                 from fortify_test.c:9:
In function 'strcpy',
    inlined from 'main' at fortify_test.c:16:3:
/usr/include/bits/string3.h:110:10: warning: call to
__builtin___strcpy_chk will always overflow destination buffer
    return __builtin___strcpy_chk (__dest, __src, __bos
(__dest));
```

```
$ ./fortify_test
Buffer Contains: `???? , Size Of Buffer is 5
*** buffer overflow detected ***: ./fortify_test terminated
===== Backtrace: =====
/lib64/libc.so.6(+0x77de5)[0x7ffff7a92de5]
```

In this example the buffer overflow is detected at compile-time, given that the size of involved buffers and data contents can be deduced. The buffer overflow is also signalled during execution.

gcc's FORTIFY_SOURCE flag (3)

```
char buffer[5]; // known size
strcpy(buffer, argv[1]); // argv[1] size not known
```

```
$ gcc -D_FORTIFY_SOURCE=1 -O fortify_test2.c -o fortify_test2
```

```
$ ./fortify_test2 abcd
```

```
$ ./fortify_test2 abcde
```

```
*** buffer overflow detected ***: ./fortify_test2 terminated
```

```
===== Backtrace: =====
```

```
/lib64/libc.so.6(+0x77de5)[0x7ffff7a92de5]
```

```
/lib64/libc.so.6(__fortify_fail+0x37)
```

```
. . .
```

```
./fortify_test2[0x400499]
```

In this example there are no warnings at compile-time, given that the size of the program argument string is only known at runtime.

But the size of the destination buffer is known, hence the buffer overflow can be detected at runtime. Under the hood The `strcpy(buffer, argv[1])` call is replaced by `strcpy_chk(buffer, argv[1], 5)`

Undefined Behavior Sanitizer

```
int sum = 0;
int numbers[N];

for (int i = 0; i <= N; i++)
    sum += numbers[i];
```

```
stack_overflow.c:9:12: runtime error: index 5 out of bounds for type 'int [5]'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior stack_overflow.c:9:12 in
Abort trap: 6
```

- **UBSan** is a gcc/clang plugin for detecting undefined behavior during execution of a program.
 - enabled using `-fsanitize=undefined` switch during compilation
 - undefined behavior errors are reported during execution, but program execution is also halted if `-fno-sanitize-recover` is specified during compilation
- The array overflow example we saw previously is now signalled.

Undefined Behavior Sanitizer (2)

```
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;  
}
```

```
null_pointer_deref.c:8:22: runtime error: member access within null  
pointer of type 'Foo'  
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior  
null_pointer_deref.c:8:22 in  
Abort trap: 6
```

- **UBSan** is a gcc/clang plugin for detecting undefined behavior during execution of a program.
 - enabled using `-fsanitize=undefined` switch during compilation
 - undefined behavior errors are reported during execution, but program execution is also halted if `-fno-sanitize-recover` is specified during compilation
- The null-pointer dereference example we saw previously now always halts (regardless of whether optimisation is turned on or off).

Address Sanitizer

```
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-after-free
free(a); // double free! (and what about b?)
```

```
==17390==ERROR: AddressSanitizer: heap-use-after-free on
address 0x6020000000d0 at pc 0x000106cf1fa6 bp 0x7fffe8f0def0
sp 0x7fffe8f0dee8
```

- [AddressSanitizer](#) is a runtime memory error detector.

Handling buffer overflows

secure programming

Secure programming

- Secure programming techniques
 - Argument validation / defensive programming
 - Avoid inherently dangerous API calls / use safe variants of those, in particular string manipulation functions in C
 - Manage dynamically allocated (heap) memory correctly
 - ...
- Established advice for secure programming
 - [SEI CERT C Coding Standard](#)
- **Validation:** Use code reviewing tools and testing to find vulnerabilities and fix them

Insecure API calls

- Examples of C functions involving string manipulation
 - For input/output: `gets` `scanf` `fscanf`
 - General string manipulation: `strcpy` `strcat` `sprintf`
- Some of these calls have bounded-length variants
 - A length argument indicates the maximum amount of memory to consider
 - Examples: `fgets` `strncpy` `snprintf`
- Bounded-length variants are not entirely safe, e.g.
 - No guarantee of null-termination for the target buffer (e.g. `strncpy`).
 - Undefined behavior when buffers overlap.
 - Some “safe” variants of string operations counter for these cases (see next slide).

C11 - ISO/IEC TR 24731

From: <http://en.cppreference.com/w/c/string/byte/strncpy>

strncpy, strncpy_s

Defined in header <string.h>

<code>char *strncpy(char *dest, const char *src, size_t count);</code>	(1)	(until C99)
<code>char *strncpy(char *restrict dest, const char *restrict src, size_t count);</code>		(since C99)
<code>errno_t strncpy_s(char *restrict dest, rsize_t destsz, const char *restrict src, rsize_t count);</code>	(2)	(since C11)

2) Same as (1), except that the function does not continue writing zeroes into the destination array to pad up to count, it stops after writing the terminating null character (if there was no null in the source, it writes one at `dest[count]` and then stops). Also, the following errors are detected at runtime and call the currently installed **constraint handler** function:

- src or dest is a null pointer
- destsz or count is zero or greater than `RSIZE_MAX`
- count is greater or equal destsz, but destsz is less or equal `strlen_s(src, count)`, in other words, truncation would occur
- overlap would occur between the source and the destination strings

■ A safer set functions in C11 - [ISO/IEC TR 24731](#)

■ Further reference:

- [“On Implementation of a Safer C Library, ISO/IEC TR 24731”](#), Laverdière-Papineau et al., 2006
- [“Security Development Lifecycle \(SDL\) Banned Function Calls”](#), Michael Howard, Microsoft Developer Network

Safe string manipulation functions

- Insecure \Rightarrow (more) secure:
 - strcat \Rightarrow strlcat
 - strcpy, strncpy \Rightarrow strlcpy (note: strncpy does not ensure NULL termination)
 - strncat \Rightarrow strlcat
 - strncpy \Rightarrow strlcpy
 - sprintf \Rightarrow snprintf
 - vsprintf \Rightarrow vsnprintf
 - gets \Rightarrow fgets
- Microsoft library versions
 - strcpy_s, strncpy_s (eq. to strlcpy), strcat_s

SEI CERT C — a few examples

STR07-C. Use the bounds-checking interfaces for string manipulation

Created by Confluence Administrator, last modified by Jill Britton on Aug 10, 2017

The C Standard, Annex K (normative), defines alternative versions of standard string-handling functions designed to be safer replacements for existing functions. For example, it defines the `strcpy_s()`, `strcat_s()`, `strncpy_s()`, and `strncat_s()` functions as replacements for `strcpy()`, `strcat()`, `strncpy()`, and `strncat()`, respectively.

[STR07-C](#)

MEM00-C. Allocate and free memory in the same module, at the same level of abstraction

[MEM00-C](#)

FIO20-C. Avoid unintentional truncation when using `fgets()` or `fgetws()`

[FIO20-C](#)

C/C++ source code analysis

- Historical tools — limited “grep-like” analysis, but security-oriented:
 - [RATS](#) (Rough Auditing Tool For Security) for C, C++, Perl, PHP, Python. *“As its name implies, the tool performs only a rough analysis of source code.”*
 - [FlawFinder](#): *“a simple program that examines C/C++ source code and reports possible security weaknesses [...] is not a sophisticated tool. It is an intentionally simple tool, but people have found it useful.”*
- Modern, more powerful C/C++/Objective-C analysers
 - [Clang Static Analyzer](#)
 - [Facebook Infer](#)
 - [Sonar Source C/C++](#) (commercial)
 - ◆ SonarSource makes plugins for other mainstream languages free for use in the community edition though

RATS/Flawfinder

- test.c:32: [5] (buffer) *gets*: Does not check for buffer overflows ([CWE-120](#), [CWE-20](#)). Use *fgets()* instead.

```
gets(f);
```

- test.c:56: [5] (buffer) *strncat*: Easily used incorrectly (e.g., incorrectly computing the correct maximum size to add) [MS-banned] ([CWE-120](#)). Consider *strcat_s*, *strlcat*, *snprintf*, or automatically resizing strings. Risk is high; the length parameter appears to be a constant, instead of computing the number of characters left.

```
strncat(d,s,sizeof(d)); /* Misuse - this should be
```

- Even if FlawFinder / RATS perform rough analysis, their generated reports include:
 - The **location** of the problems
 - **Description of the potential vulnerability** and corresponding CWE reference
 - **Suggestion for change** in the code

Clang static analyzer - screenshots

```
33
34 int flawed_function(Foo* pointer) {
35     int v = pointer -> data; // dereference before check
```

3. Entered call from 'mxain'

4. Access to field 'data' results in a dereference of a null pointer (loaded from variable 'pointer')

```
39 }
40
```

```
int main3(int argc, char** argv) {
    int n;    char *a, *b;
    --argc; ++argv;
    n = argc == 1 ? atoi(argv[0]) : 10;
    a = (char*) malloc(n); // allocate memory for a
    memset(a, 'x', n);      // set all positions to 'x'
    PRINT(a, n);           // print contents
    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");
    PRINT(a, n), PRINT(b, 2*n); // print contents of invalid a &
    initial b
    memset(b, 'X', 2*n);      // set
    PRINT(a, n); PRINT(b, 2*n); // print
    memset(a, 'x', n);        // use dangling reference, set to 'x'
    PRINT(a, n); PRINT(b, 2*n); // print again
```

1. Assuming 'argc' is not equal to 1

2. Memory is allocated

3. Memory is released

4. Use of memory after it is freed