Theoretical Exercise 1 – TDT4205

Fagervik, Jørgen
Håkonsen, Iver

January 27, 2021

Parameter passing

The value returned from the `main` function is 23. The arguments to `increment_with_value` are `int`s, which means that the function will be called by value, not by reference. For the function to work as the name intends, the type of the parameter `a` instead has to be `int *`.

```
#include <stdio.h>

int a = 23;
void increment_with_value(int *a, int b) {
    *a += b;
}

int main(void) {
    increment_with_value(&a, 1);
    return a; // 24
}
```

Symbols

`b` is a local variable inside the function `increment_with_value`, and will be allocated on the stack when the function is called.

C arrays

a. The call to `strcpy` copies the 14 chars stored in `t` into `s`, but `s` only fits 12 chars. `strcpy` does not do bounds checking at all, and writes 2 chars beyond, and overflows into `foo`. `foo` is an `int`, so it at least two bytes wide, and the values `'2'` and `'3'` are written into it, i.e., `foo` is now `0x3332`, or 13106 in decimal. This leads to `foo = 13106` being printed.

b. This problem is known as buffer overflow, which happens when a program either reads or writes beyond the allocated space of an array.

c. Some patched versions of `gcc` has this option set already, and it has to be disabled with `-fno-stack-protector`. In stock versions of `gcc` it has to be enabled with `-fstack-protector`.

d. If line 5 was replaced with `static int foo = 0;`, the problem in this particular case would be solved. `foo` would be allocated in static memory instead of on the stack, and the stack smashing would not hit it. It does not solve the underlying problem of stack smashing.

Functions and variables

a.

Symbol	Memory segment
`rec()`	text
`c`	Initialized Data Segment, read-only area
`d`	Uninitialized Data Segment (bss)
`counter`	Uninitialized Data Segment (bss)
`a`	Stack

b. Running the program causes a segmentation fault, caused by a stack overflow.