Andreas,Hallvard & Snorre - Group 3

TDT4186 Operating Systems - Theoretical Exercises 1 - Group 3

1.1 Parameter passing

Consider the following C program:

```c
#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;
}
```

Without compiling and running the program, indicate which value is returned by the main function? Briefly explain your answer.

Answer:
It will return 23. The increment in “increment_with_value” is not returned and will therefore not affect “a”.

1.2 Symbols

If we compile the program shown above using `gcc -std=c11 -Wall -o test test.c` and execute `nm test` afterwards, the `nm` output does not contain a memory address for variable `b`.
Briefly explain why `b` is not listed.

Answer:
Because b is never saved as a variable, it's just an internal variable in the increment_with_value-function.

1.3 C arrays

Consider the following C program:

```
   #include <stdio.h>
2  #include <string.h>

4  int main(void) {
     int foo = 0;
6    char s[12];
     char *t = "01234567890123";
8
     printf("foo %p\n  s %p\n", &foo, s);
10   strcpy(s, t);
     printf("foo = %d\n", foo);
12 }
```

a. *Without compiling and running the program*, give the value printed for `foo`.

Answer:

The value printed for foo is 13106.

b. Describe briefly the problem that shows up in the given code which results in this output.

Answer:

The memory of s is smaller than the size of t, making the strcpy not valid and it overflows foo. Here one could use strncopy(s,t,12) to prevent overflow.

c. Modern C compilers protect against the problems shown in this example. For `gcc` or `clang`, find out which command line option can be used to enable this protection.

Answer:

gcc: the various `-fstack-protector` flags

clang: -fsanitize=address, -fsanitize=bounds, SafeCode

d. What would the output be if line 5 was replaced by

`static int foo = 0;`

Briefly explain whether this change would solve the underlying problem.

Answer:

This will stop the overflow from affecting foo, but the overflow will then affect t and thus not solve the underlying problem.

1.4 Functions and variables

Consider the following C program:

```c
#include <stdio.h>

const int c = 1;
int d, counter = 0;

unsigned int rec(unsigned int number) {
  counter++;
  return rec(counter);
}

int main(void) {
  int a = rec(c);
  printf("%d\n", a);
  return 0;
}
```

a. Which memory segments are the function `rec()`, variables `c`, `d`, `counter`, `and a` as well as parameter `a` located in?

<u>Answer:</u>

rec() = Text segment
c = Data
d = Data
counter = Data
a = Stack
parameter a = Stack

b. What happens if you execute the compiled program? What changes if you add a local variable `char array[1000]` to function `rec`?

<u>Answer:</u>
It is an infinite loop and it will not stop before it reaches stack overflow. If we add char array[1000] it will use more memory and will reach stack overflow with fewer iterations.