# *Valgrind* tutorial

*Valgrind* is a tool which can find memory leaks in your programs, such as buffer overflows and bad memory management. This document will show per example how *Valgrind* responds to buggy code and how *Valgrind* reports these bugs.

# Dangling pointers

```
1  #include <stdint.h>
2
3  uint8_t * allocate_memory();
4
5  int main(int argc, char * argv[]) {
6          uint8_t * buffer = allocate_memory();
7          buffer[0] = 0;
8          return 0;
9  }
10
11 /*
12  * Returns a pointer to newly allocated 1kB memory.
13  */
14 uint8_t * allocate_memory() {
15         uint8_t memory[1000];
16         return memory;
17 }
```

Function `allocate_memory` has `memory`, a 1,000 byte array, as a local variable. A pointer to the first element of this array is returned, such that the calling function may use this allocated memory. But, after returning from the function, the stack that was created for the `allocate_memory` function has been freed along with its local variable `memory`. In line 7, the `main` function will write to a place in memory that can be used by other functions, which may lead to unexpected behavior.

If you use the *GCC* compiler, you will see that *GCC* warns you about the possible bug in your code:

```
> gcc local_pointers.c -o local_pointers
local_pointers.c: In function allocate_memory:
local_pointers.c:16:2: warning: function returns address of
local variable [enabled by default]
```

Running *Valgrind* on the program yields the following output:

```
> valgrind --tool=memcheck ./local_pointers
Invalid write of size 1
==6276==    at 0x40057D: main (in /../local_pointers)
==6276==  Address 0x7fefffc60 is just below the stack ptr.
To suppress, use: --workaround-gcc296-bugs=yes
```

# Memory leaks

```c
#include <stdint.h>
#include <stdlib.h>

uint8_t * allocate_memory();

int main(int argc, char * argv[]) {
        uint8_t * buffer = allocate_memory();
        buffer[0] = 0;
        return 0;
}

/*
 * Returns a pointer to newly allocated 1kB memory.
 */
uint8_t * allocate_memory() {
        uint8_t * memory = (uint8_t *) malloc(sizeof(uint8_t) *
            1000);
        return memory;
}
```

Function `allocate_memory` creates 1kB of space on the heap by a call to function `malloc`. After the `main` function is executed, the allocated 1kB will remain unavailable to other processes. So, always use the `free` function to deallocate the allocated memory so other processes can use it again.

Note that *GCC* does not warn you about this bug. Running *Valgrind* on the program yields the following output:

```
> valgrind --tool=memcheck ./no_free
==11389== HEAP SUMMARY:
==11389==     in use at exit: 1,000 bytes in 1 blocks
==11389==   total heap usage: 1 allocs, 0 frees, 1,000 bytes allocated
==11389==
==11389== LEAK SUMMARY:
==11389==    definitely lost: 1,000 bytes in 1 blocks
==11389==    indirectly lost: 0 bytes in 0 blocks
...
```

As you can see, *Valgrind* reports that 1kB of memory on the heap is still in use after the program `no_free` is done executing.

## Actions based on uninitialized data

```
1 int main(int argc, char * argv[]) {
2         int i;
3         while (i < 10) {
4                 // Do something here
5                 ++i;
6         }
7         return 0;
8 }
```

Function `main` creates a variable `i`, which is never initialized. Based on the value of `i`, the `while`-loop will iterate for a certain amount of time. Because the value of `i` is not known, an unknown number of iterations will occur.

Running *Valgrind* on the program yields the following output:

```
> valgrind --tool=memcheck ./uninitialized_data
==11650== Conditional jump or move depends on uninitialised value(s)
==11650==    at 0x400501: main (in /./uninitialized_data)
```

As you can see, *Valgrind* warns you that a conditional jump (such as `if`, `switch`, `for`, `while` statements) or move is based on data with uninitialized values.

## Writing past the array boundary

```c
#include <stdlib.h>
#include <string.h>

int main(int argc, char * argv[]) {
        char * course_name = "TI2725-C"; // "TI2725-C" + '\0'

        char course_name_cpy1[8];
        strcpy(course_name_cpy1, course_name);

        char * course_name_cpy2 = malloc(sizeof(char) * 8);
        strcpy(course_name_cpy2, course_name);

        free(course_name_cpy2);
        return 0;
}
```

The `main` function copies the string `TI2725-C` into two variables, one on the stack and one on the heap. Note that the string consists of 9 characters, as the strings ends with `\0`, the `NULL`-terminating character.

The array `course_name_cpy1` has only allocated memory on the stack for eight characters. When calling the `strcpy` function, the ninth character of the string will be written in the first byte next to the eight allocated bytes.

Running *Valgrind* on the program yields the following output:

```
> valgrind --tool=memcheck ./out_of_bounds
==12190== Invalid write of size 1
==12190==    at 0x4C2D812: strcpy (in /./vgpreload_memcheck-amd64-linux.so)
==12190==    by 0x400685: main (in /./out_of_bounds)
==12190==  Address 0x51fd048 is 0 bytes after a block of size 8 alloc'd
==12190==    at 0x4C2CD7B: malloc (in /./vgpreload_memcheck-amd64-linux.so)
==12190==    by 0x40066E: main (in /./out_of_bounds)
==12190==  ...
```

You can see that *Valgrind* warns you about a write of one byte in the function `strcpy`, which is called by the function `main`. The ninth character was to be written to location `0x51fd048`, which is not allocated and resides in memory just next to the eight bytes that were allocated.

You may ask yourself why *Valgrind* doesn't warn you about the first bug in the code, which copies the string onto too little allocated memory on the stack. The catch is that ***Valgrind* does not perform bounds checking on memory on the stack**.