This document is to describe the idea behind each function that i am using. More details is to find in the comment of the tracer.c file.

```
static pid_t get_pid(const char* proc_name)
```

This function will just execute the command *pgrep* on the process specify by *proc_name* and return the result of the command. It also initialize the global variable *tracee_pid_len*.

```
static unsigned long get_function_addr(const pid_t pid, const char* prog_name,
                                       const char* func_name)
```

The goal of the function is to return the address of the existing function (*func_name*) of the tracee (*prog_name*). It will do it by adding the starting address of the main of the tracee, that we get by executing the command *cat /proc/pid/maps,* and the offset the wanted function using the command *nm prog_name | grep func_name.*

```
static unsigned long get_libc_function_address(const pid_t tracee_pid,
                                               const unsigned long function_addr)
```

Return the address of a libc function (specify by *function_addr*) of the process *tracee_pid.* Assuming that the function of the libc are always at the place, the function get the starting address of the libc for the tracer and the tracee, compute the offset of the function inside the libc of the tracer and add it to the address of the libc of the tracee.

```
static int get_function_size(const char* function_name, const char* prog_name)
```

The function execute the command *nm -S -t d prog_name | grep function_name*
The nm command display the symbol table of a file, the *-S* option is to show the size of each symbol and the option *-t d* is to display the size in decimal format. Finally it grep the *prog_name* and return the size.

```
static void getdata(const pid_t pid, const long address_to_read, const int size,
                    unsigned char * data)
```

This function open the memory of the process specify by *pid,* read *size* amount of bytes starting at the address *address_to_read* and store it in the array *data*.

```
static void putdata(const pid_t pid, const unsigned long address_to_write,
                    const int size,
                    const unsigned char * data)
```

This function write the content of the array *data* at the address *address_to_write* inside the memory of the process specify by *pid*

```
static int isAddrInHeap(pid_t pid, unsigned long address)
```

The function execute the command *cat proc/pid/maps* and verify that the address *address* is inside an executable block of the heap

```
static int call_func(const pid_t pid, const unsigned long running_function,
                     const unsigned long function_to_call_address,
                     unsigned char * code,
                     const int parameter)
```

For this function use ptrace to make the process specify by *pid* call the function at the address *function_to_call_address.* The function is called by an indirect call (array *code* here), take an integer as *parameter* and return an integer. Also we put a break point after the indirect call to regain the execution of the process.
Here is the procedure :
We open the memory of the process to get a backup of the byte code that we will modify. We put a break point at the address *running_function* (that is the beginning of the running function of the tracee) to get the values of the register of the tracee.
Write at the *running_function* address the instruction for the indirect call.
Then we can now set the new value for the register, the rax takes the *function_to_call_address* because an indirect call will check the rax and execute the address in it, the rip takes the *running_function* address because we want the indirect call to be the next instruction executed and rdi is the register for the *parameter*.
We restart the process, wich means the *function_to_call_address* is executed and we hit a break point. So we can get the return value with the rax, restore the value of the registers and write the backup.

For the call_getpagesize, call_memalign, call_mprotect and call_free functions the procedure is the same. The difference is the return type or/and the number/type of the parameter each functions take. Also to ensure that the tracer is running well the return value of mprotect() and posix_memalign() are tested inside their respective function.

```
static void trampoline(const pid_t pid, const unsigned long running_function,
                       const unsigned long function_to_call_address,
                       const int parameter)
```

In this function we create a trampoline so no need for backup. We need to get the value of the registers. Write at the *running_function* address a jump instruction followed by the *function_to_call_address* and finally an ending instruction. Set the new value for the registers, the rip takes the *running_function* address, because this is where we wrote the jump and so on, and *parameter* for rdi and that is it.