From pointer to GDB my H.3.R.0

1 Introduction

Good Luck.

1.1 Malloc

Malloc is an allocator of memory. You will know more about it later. For the moment you just have to know that if you want to store some informations in the memory, you must use malloc. For example as you may probably noticed, ”String” type does not exist in C language. A string is simply an array of characters no? So what if we just put a lot of characters one after another in a memory space?

```c
#include <stdlib.h>
int main(void)
{
    //we want to write the alphabet (26 chars + 1 for the '\0')
    char *text = malloc(27 * sizeof(char))
    if (!text)
        return 1;
    for (int i = 0; i < 26; i++)
        text[i] = 97 + i;
    text[26] = '\0';
    printf("%s\n", text);
    return 0;
}
```

```
42sh> abcdefghijklmnopqrstuvwxyz
```

For char* array, the end of the array is declared by using ”. There is no equivalent for the other types.

1.2 GDB

GDB is a debugger. Learn to use it well. Here is some commands :

1.2.1 start

start will start the program from the beginning of the main. It can also take arguments.

1.2.2 run

run will start the program running under gdb. The program that starts will be the one that you have previously selected with the file command, or on the unix command line when you started gdb. You can give command line arguments to your program on the gdb command line the same way you would on the unix command line, except that you are saying run instead of the program name: run 2048 24 4

1.2.3 break

a “breakpoint” is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the break command. break 'function' sets the breakpoint at the beginning of function. If your code is in multiple files, you might need to specify filename:function.
break 'linenumber' or break filename:linenumber sets the breakpoint to the given line number in the source file. Execution will stop before that line has been executed.

1.2.4 delete

delete will delete all breakpoints that you have set. delete 'number' will delete breakpoint numbered number. You can find out what number each breakpoint is by doing info breakpoints. (The command info can also be used to find out a lot of other stuff. Do help info for more information.)

1.2.5 clear

clear 'function' or 'line' will delete the breakpoint set at that function or line.

1.2.6 continue

continue will set the program running again, after you have stopped it at a breakpoint.

1.2.7 step

step will go ahead and execute the current source line, and then stop execution again before the next source line.

1.2.8 next

next will continue until the next source line in the current function. This is similar to step, except that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again.

1.2.9 print

print expression will print out the value of the expression, which could be just a variable name. print var will print the content of var.

1.2.10 N.B

p -¿ print n -¿ next s -¿ step disp -¿ display ...

1.3 Valgrind

Valgrind is a tool which can find memory leaks in your programs, such as buffer overflows and bad memory management. Function "allocate_memory" creates 1kB of space on the heap by a call to the 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:

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

uint8_t * allocate_memory ()
{
    uint8_t * memory = ( uint8_t *) malloc ( sizeof ( uint8_t ) * 1000) ;
    return memory ;
}

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

```
42sh > 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.
Valgrind can also be very helpful when you are trying to write on a memory that is not allocated, or when you do not initialize a variable.

1.4 Fsanitize

```c
#include <stdlib.h>

int main()
{
    int *arr = malloc(5 * sizeof(int));
    for (int i = 0; i < 6; i++)
        arr[i] = i;
}
```

Now compile using : gcc -o main main.c -fsanitize=address. If you run ./main :

```
==10668==ERROR: AddressSanitizer: heap-buffer-overflow on address
 0x603000000024 at pc 0x55d2a02268a5 bp 0x7fff090cd460 sp 0x7fff090cd450
WRITE of size 4 at 0x603000000024 thread T0
    #0 0x55d2a02268a4 in main (/tmp/a.out+0x8a4)
    #1 0x7fdaa9ddcb96 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
    #2 0x55d2a0226759 in _start (/tmp/a.out+0x759)

0x603000000024 is located 0 bytes to the right of 20-byte region
[0x603000000010,0x603000000024)
allocated by thread T0 here:
    #0 0x7fdaaa28ab50 in __interceptor_malloc
    (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xdeb50)
    #1 0x55d2a022684b in main (/tmp/a.out+0x84b)
    #2 0x7fdaa9ddcb96 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/tmp/a.out+0x8a4) in main
Shadow bytes around the buggy address:
  0x0c067fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c067fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c067fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c067fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c067fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
=>0x0c067fff8000: fa fa 00 00[04]fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==10668==ABORTING
```

All of this simply means that you are trying to read something over your memory allocated data.

1.5 Pointers

Be CAREFUL while using pointers. Here are some tips :

1.5.1 Difference between pointer and address

A pointer is a link to an address which is a memory cell that can contain a variable.

```c
int *p = 5;
```

P is an address. If we want to get the value inside we have to dereference it '*p'. Using malloc allows us to get a memory space which represents a list of contiguous addresses.

1.5.2 Copy your pointer before any manipulation

If you have a linked list for example :

```c
struct linked_list
{
    int data;
    struct linked_list *next;
};

int main(void)
{
    struct linked_list *list = malloc(sizeof(struct linked_list));
    //imagine if we fill the list before the next of the code
    while (list)
    {
        list = list->next;
    }
    //Here we lost the pointer of the head.
    //You should use a copy of list instead of list
    struct linked_list *cpy = list;
    while (cpy)
    {
        cpy = cpy->next;
    }
    //Here cpy = NULL and list is still the head of the linked_list
}
```

1.5.3 Pointer Arithmetic

For this part, here is a tutorial that explains pretty well the concept :
url http://web.cse.ohio-state.edu/ reeves.92/CSE2421au12/SlidesDay12$_1$3.*pdf*