Malloc recitation 1 (Fall 2019)

The tar file is located at https://www.cs.cmu.edu/~213/activities/f19-rec-malloc.tar. Download and extract the tar on a Shark machine, then compile the source:

```
$ wget https://www.cs.cmu.edu/~213/activities/f19-rec-malloc.tar
$ tar xvf f19-rec-malloc.tar
$ cd f19-rec-malloc
$ make
```

Example 1 (mm.c)

Run the following command:

```
$ gdb --args ./mdriver -c traces/syn-mix-short.rep
(gdb) run
```

Oh no, a segmentation fault! Let's figure out where we are…

```
(gdb) backtrace
(gdb) list
```

Hmm, let's print the block header.

```
(gdb) x/gx block                    (show 8 bytes of memory)
(gdb) print *block                  (show struct contents)
(gdb) print *(block_t *) <address>  (same thing)
```

That doesn't look too right. We should only call write_footer on a block if it has a valid header, right? Let's take a look at the calling function.

```
(gdb) frame 1
(gdb) list
```

Aha, that doesn't look right…

Incidentally, if you run mdriver-dbg, it actually assert-fails in the write_footer function instead, which would've made the bug even easier to track down. Please make liberal use of assertions in your own code, and make sure you have a robust checkheap function!

Example 2 (mm-2.c)

Now, let's try mdriver-2. This time we'll run with "-d 2", which performs stricter error checking.

```
$ gdb --args ./mdriver-2 -d 2 -c traces/syn-mix-short.rep
(gdb) run
```

The first time you run, you should get a garbled bytes error. Set a watchpoint on the address of the first garbled payload you see, which will stop whenever that address is modified. Then start running again.

```
(gdb) watch *0x8000026d0

(gdb) run
(gdb) continue
(gdb) continue
(gdb) continue
(gdb) continue
```

Most of the time you'll stop in the __memcpy_ssse3_back function (this is just the driver putting data into the payload). Continue until you stop in the mm_free function. Where are we?

```
(gdb) backtrace
(gdb) list
```

Hmm, nothing seems suspicious there. Maybe it's because of our optimization level! The -O3 optimization level causes GDB to emit incorrect stack traces.

Try running again, but with mdriver-2-dbg this time. A neat trick is that you can change the file without changing the args/watchpoints as follows:

```
(gdb) file mdriver-2-dbg
(gdb) run
(gdb) continue
...
```

Ah, that line of code looks pretty suspicious! Note that the backtrace now shows the correct function name:

```
(gdb) list
```

That's because -Og preserves important debugging information like the stack trace, which you will probably find useful.

One last thing: try printing out the values of the prev_alloc and next_alloc variables.

```
(gdb) print prev_alloc
$1 = <optimized out>
```

Unfortunately, the variable is still optimized out! While -Og ensures that the information that GDB shows is *correct* (compared to -O3), there will still be some variables that are optimized out. This makes sense if you think about how assembly works: the value of many variables are often discarded, unless there is a reason to save them.

To fix this, go into the Makefile and change the debug optimization level to "COPT_DBG = -O0". The resulting code is very inefficient, but it should preserve *all* local variables.

```
(gdb) quit
$ vim Makefile
```

Now force-recompile and run again. (Make sure the filename is mdriver-2-dbg this time.) We'll set a breakpoint directly on line 450 now, to save us time. Printing out the value of next_alloc should now work!

```
$ make clean
$ make
$ gdb --args ./mdriver-2-dbg -d 2 -c traces/syn-mix-short.rep
(gdb) b mm-2.c:450
(gdb) run
(gdb) continue
...

(gdb) print next_alloc
$1 = true
```