CSE-376 (Spring 2020) Homework Assignment #2
(version 5)
Due Monday, March 30, 2020, 11:59pm
(This assignment is worth 15% of your grade.)

*** PURPOSE:

To become familiar with many common C bugs involving pointers and memory corruptions. To develop libraries that can capture some of these bugs, report, and possibly act on them. This program may be about 500 LoC or so.

*** Part 1 -- LKmalloc:

Write your own simple malloc-debugging "library" (a .c file you can compile and link with another program), called LKmalloc (LeaK detecting malloc). This library should export AT LEAST these three functions: lkmalloc(), lkfree(), and lkreport(). lkmalloc/lkfree should have the same functionality as the usual malloc/free functions. In turn, lkmalloc and lkfree should use malloc and free, respectively (not the brk or sbrk system calls). Users who use this library are expected then to call lkmalloc/lkfree directly (not malloc/free). lkreport() would produce a report of memory leaks, allocations, and more.

Prototypes:

```
1. int lkmalloc(u_int size, void **ptr, u_int flags)
```

This will ask to allocate "size" bytes, and if successful, assign the newly allocated address to *ptr. By passing an addr or a ptr to lkmalloc(), you can do more intelligent checking of bugs. lkmalloc() should return 0 on success -errno on failure (e.g., -ENOMEM). Example use

```
char *buf = NULL;
int ret;
ret = lkmalloc(10, &buf, flags);
```

The 'flags' variable should mean as follows (LKM_* is the #define name):

- LKM_REG 0x0: allocate memory without any of the special protections below.
- LKM_INIT 0x1: initialize the memory being allocated to 0s.
- LKM_OVER 0x2: allocate 8 more bytes of memory after the requested size, and write the pattern 0x5a in those upper bytes.
- LKM_UNDER 0x4: allocate 8 more bytes of memory before the requested size, and write the pattern 0x6b in those lower bytes.

These "over" and "under" are called 'redzones' -- special values put before and/or after a buffer in order to try and detect buffer over/underflows. You should consider whether some flags can be combined or not.

```
2. int lkfree(void **ptr, u_int flags)
```

This'll take the addr of the ptr that was presumably allocated by lkmalloc (but maybe not), and attempt to free it. Return 0 on success, -errno on failure (e.g., -EINVAL, etc.).

The 'flags' variable should mean as follows (LKF_* is the #define name):

- LKF_REG 0x0: free only if the ptr passed was exactly as was allocated.
- LKF_APPROX 0x1: free an allocation even if what is passed is in the middle of a valid allocation (normally free doesn't allow that).
- LKF_WARN 0x2: print a warning if you free a ptr as per LKF_APPROX.
- LKF_UNKNOWN 0x4: print a warning if asked to free a ptr that has never been allocated.
- LKF_ERROR 0x8: exit the program if any condition matches LKF_WARN or LKF_UNKNOWN.

You should consider whether some flags can be combined or not.

BTW, think about why the API I list above uses a double pointer and lkmalloc() doesn't just return the addr back.

Internally, lkmalloc and lkfree should track what memory "objects" (i.e., generic bufs allocated) are being allocated and freed. The functions should warn when at least these conditions occur: double free, double malloc (a memory leak), trying to free something in the middle of an allocated memory chunk, trying to free a NULL or addr that was never allocated, and memory leaks. By "warn" I mean that they should "fprintf(stderr)" a descriptive error message saying what went wrong.

The lkmalloc library should internally capture the following information in each "record" (e.g., each time lkmalloc/lkfree is called): the file name, line number, and function name where the allocation or free took place; the pointer allocated and returned; the pointer passed; and the time of the allocation (in UNIX seconds+microseconds since the epoch). You will need to design/use an efficient data structure to store this information (e.g., a hash table, tree, something else? Think how this'll scale if you have millions of alloc/free records).

```
3. int lkreport(int fd, u_int flags)
```

This will dump a report of memory issues to the file whose file descriptor is 'fd'. Return -errno on any error (e.g., invalid fd); return the number of records in your report (e.g., return 17 if the report written to fd had 17 "records" captured) on success. Note that 'fd' can be stdout or stderr, as well as any file you opened for writing.

The 'flags' field can be as follows (LKR_* is the #define name):

- LKR_NONE 0x0: do not produce a report
- LKR_SERIOUS 0x1: print only the most serious memory leaks (e.g., mallocs w/o a corresponding free of the same addr)
- LKR_MATCH 0x2: print perfectly matching alloc/free pairs
- LKR_BAD_FREE 0x4: print bad ‘free's (ones where the passed addr is in middle of alloc addr)
- LKR_ORPHAN_FREE 0x8: print orphan ‘free's (ones that had never been allocated)
- LKR_DOUBLE_FREE 0x10: print double free’d pointers
- (Other flags may be defined later)

You should consider whether some flags can be combined or not.

Study on_exit(3) and call lkreport on exit from your program. This has the benefit of reporting real memory leaks when the program is exiting (meaning that anything that wasn't free is potentially a leak).

When printing, use a CSV format such as this, which can be loaded into any spreadsheet program (also print the following headers as line 1 of the CSV file):

```
record_type,filename,fxname,line_num,timestamp,ptr_passed,retval,size_or_flags,
alloc_addr_returned
```

Your report will include both lkmalloc and lkfree records, distinguished by a different integer in the “record_type” field; but there’s some different information needed for each record type. All columns from “filename” to “retval” are common fields to both records. If "record_type" is 0 (for lkmalloc records), then “size_or_flags” should print the size requested; and “addr_returned” should be the address returned to the caller of lkmalloc. If "record_type" is 1 (for lkfree records), “size_or_flags” should print the flags passed; addr_returned should be left empty.

Your debugging library should be put into a file called lkmalloc.c.

Next, write several small test C programs to detect each one of the issues listed above for programs that link with your lkmalloc.o. Also write a driver program to demonstrate each of the flags and each of the "bad things" that your library catches through these tests.

*** Part 2 -- test scripts:

For each of the test conditions above, write a short test script to exercise the condition. Like the first assignment, name your tests test01.sh, test02.sh, etc.

*** Part 3 -- Makefile:

Your Makefile should have different targets for building the library, each test C program, for cleaning, for running test scripts, etc. Be sure to set proper CFLAGS, LDFLAGS, CC, etc. Use at least "-Wall -Werror" as your flags.

Add also a "make depend" target: to build a dependency set of files automatically, which will get included in the makefile if they exist. You can assume GNU Make (hint: -include syntax) and GNU gcc syntax (so check the gcc man page for -MD options).

* STYLE AND MORE:

Aside from testing the proper functionality of your code, we will also evaluate the quality of your code. Be sure to use a consistent style, well documented, and break your code into separate functions and/or source files as it makes sense.

To be sure your code is very clean, it must compile with "gcc -Wall -Werror" without any errors or warnings! If the various sources you use require common definitions, then do not duplicate the definitions. Make use of C's code-sharing facilities.

You must include a README file with this and any assignment. The README file should describe what you did, what approach you took, results of any measurements you made, which files are included in your submission and what they are for, etc. Feel free to include any other information you think is helpful to us in this README; it can only help your grade. The code you write should be your own, but if you want to use any online code, you must clear it with me (note: github sources NOT allowed), and cite it both in your code and your README. Make sure to reasonably adhere to the requirements of the README.

* SUBMISSION

You will need to submit all of your sources, headers, scripts, Makefiles, and README. Submission is accepted via GIT only! Do not submit regenerable files like binaries, *.o files, or any temp files -- only true "source" files.

PLEASE test your submission before submitting it, by checking it out in a separate directory, compiling it cleanly, and testing it again. DO NOT make the common mistake of writing code until the very last minute, and then trying to figure out how to use GIT and skipping the testing of what you submitted. You will lose valuable points if you do not get to submit on time or if your submission is incomplete!!!

(General GIT submission guidelines are available on the class website.)

*** EXTRA CREDIT (up to 19 points, optional for all students):

If you do any of the extra credit work, then your EC code must be wrapped in

```
#ifdef EXTRA_CREDIT
     // EC code here
#else
     // base assignment code here
#endif
```

This extra credit is worth a total of 20 extra points (the main assignment is worth 100 points).

[A] 15 points.

Study the mmap(2) API including mprotect(2) and more. Support the following two additional lkmalloc flags:

- LKM_PROT_AFTER 0x8: given requested allocation of size S, allocate one or more "primary" new pages, using mmap, to hold the newly requested buffer of size S. Align this new memory allocation so the last byte of the requested size S is also the last byte of the newly primary allocated pages. This means that the user will be given a starting address to access the buffer, that is calculated as the last addr of the primary set of pages minus S. Then allocate using mmap ANOTHER memory page whose

address is right after the that last byte of the primary pages, and set this new page's protection to PROT_NONE.
- LKM_PROT_BEFORE 0x10: Same as LKM_PROT_AFTER, but instead, align the first byte of S with the first byte of the primary pages; and then alloc a new page before the first byte, and set that new page's protection to PROT_NONE.

Note: it may be possible to alloc all these pages at once and just set the mmap protection bits for each page as needed.

By using PROT_NONE, any program that tries to read or write a byte from a PROT_NONE-protected page, will generate a SEGV. This is a clean way, using OS and MMU facilities, to catch buffer under/overflows. Note that if the memory allocated size is a multiple of 4KB, then it would be possible to turn on BOTH LKM_PROT_AFTER and LKM_PROT_BEFORE. Add more test scripts to showcase any combination of these two new LKM_PROT_* flags.

[B] 4 points (max).

To incentivize you to submit your code earlier than the deadline, we'll give you 2 points if you submit your assignment at least 24 hours before the official deadline; and 4 points if you submit at least 48 hours before.
Note that we count the LAST git-push to your repo as the last time you submitted the assignment, even if you made a very small change.

Good Luck.

* Copyright Statement

* ChangeLog

- V1: initial draft.
- V2: TA review
- V3: add LKR_DOUBLE_FREE flag
- V4: update CSV header titles for lkreport
- V5: clarify you should adhere to the README requirements closely. Extend the deadline to Monday March 30 (given that spring break was extended).