System Programming
Linker Lab: Memtrace

Introduction

In this lab, we use the at-load-time library interpositioning to trace calls to the dynamic memory management system of a compiled binary program. Your memory tracer will be able to log all dynamic memory management calls of a program and identify memory blocks that a program does not free. You then augment your memory tracer to prevent incorrect memory deallocations of random binaries.

By solving this lab, you will learn a lot about library interpositioning and simple management of data within a shared library. A lot of the code is provided already, this lab will also force you (an teach you how) to read someone else’s code.

The lab may require a significant effort if you are not familiar with C programming. We provide enough hints and support in the lab sessions so that everyone can solve the lab with a good mark. Nevertheless, start early and ask early!

Logistics

Installation
Download the tarball from

```
eTL : linklab.tar
```

and unpack it into a directory of your choice. In the examples below, we assume the directory name is `linklab`:

```
devel@gentoo ~ $ mkdir linklab
devel@gentoo ~ $ cd linklab
devel@gentoo ~/linklab $ tar xvzf linklab.tgz
devel@gentoo ~/linklab $ ls -l
```

Compiling, Running and Testing
`Makefiles` in the various directories assist you with submitting your compiling, running, and testing your implementations. Do not modify the Makefiles unless explicitly instructed. Run 'make help' to find out which commands the Makefile support.
A set of test programs is provided with which you can test your solution. Note that we will use a different test set to evaluate your submission.

```
~/linklab/part1 $ make help
```

Submission
You should submit a report about your implementation. You should include description about how to implement for each part of the lab, what was difficult, and wat was surprising, and so on. The report is 10 points.

Upload a zip file of your submission by eTL. The file should include a tarball of your implementation and a report.

Dynamic Memory Management

Overview
In many languages, memory is explicitly allocated and sometimes even deallocated by the programmer. The POSIX standard defines the following four dynamic memory management functions that we want to intercept.

`void *malloc(size_t size)`
`malloc` allocates `size` bytes of memory on the process' heap and returns a pointer to it that can subsequently be used by the process to hold up to `size` bytes. The contents of the memory are undefined.

`void *calloc(size_t nmemb, size_t size)`
`calloc` allocates `nmemb*size` bytes of memory on the process' heap and returns a pointer to it that can subsequently be used by the process to hold up to `size` bytes. The contents of the memory are set to zero.

`void *realloc(void *ptr, size_t size)`
`realloc` changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents are copied up to `min(size, old size)`, the rest is undefined.

`void free(void *ptr)`
`free` explicitly frees a previously allocated block of memory.

Use the Unix manual to learn more about `malloc`, `calloc`, `realloc`, and `free`:

```
~/linklab $ man malloc
```

Dynamic memory management functions are defined in the standard library. Include `stdlib.h` in your program to have access to `malloc`, `calloc`, `realloc`, and `free`.

Example

The following example program demonstrates how dynamic memory management functions can be used.

```
#include <stdlib.h>

void main(void) {
  void *p;
  char *str;
  int  *A;

  // allocated 1024 bytes of memory
  p = malloc(1024);

  // allocated an integer array with 500 integer
  A = (int*)calloc(500, sizeof(int));

  // allocate a string with 16 characters…
  str = (char*)malloc(16*sizeof(char));

  // ...then resize that string to hold 512 characters
  str = (char*)realloc(str, 512*sizeof(char));

  // finally, free all allocated memory
  free(p);
  free(A);
  free(str);
}                                                    example1.c
```

Note that all allocators return an untyped pointer (`void*`) that needs to be converted to the correct type in order to prevent compiler warnings.

Part 1: Tracing Dynamic Memory Allocation (30 Points)

Subdirectory: part1/

In this part, your job is to trace all dynamic memory allocations/deallocations of a program. Print out the name, the arguments, and the return value of each call to `malloc`, `calloc`, `realloc`, and `free`. When the program ends, print statistics about memory allocation (number of bytes allocated over the course of the entire program, average size of allocation). You can ignore freed bytes when calling realloc (just add all allocated bytes).

For the program `test1.c` given below

```
#include <stdlib.h>

void main(void) {
  void *a;

  a = malloc(1024);
  a = malloc(32);
  free(malloc(1));
  free(a);
}                                                     test1.c
```

the following output should be generated (the pointer values may differ from system to system):

```
~/linklab/part1 $ make run test1
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x55d13a4622d0
[0003]          malloc( 32 ) = 0x55d13a462710
[0004]          malloc( 1 ) = 0x55d13a462770
[0005]          free( 0x55d13a462770 )
[0006]          free( 0x55d13a462710 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total          0
[0012]
[0013] Memory tracer stopped.
~/linklab/part1 $
```

Leave the statistics on the total number of freed memory bytes (`freed_total`) at 0 for now.

You can start from scratch or implement your solution by extending the skeleton provided in `~/linklab/part1/memtrace.c`.

Use the logging facilities provided in `~/linklab/utils/memlog.c or .h`. This will ensure that your output looks exactly as above and we can automatically test your submission for correctness.

Part 2: Tracing Unfreed Memory (30 Points)

Subdirectory: part2/

The tracer from part 1 is quite useful, but it has a serious shortcoming: it cannot check whether all allocated memory has been freed. In this part, we add this functionality. While the program is running, keep track of all allocated blocks and check which ones get deallocated. When the program ends, print a list of those blocks that were not deallocated. In addition, also compute and output statistics about the total number of freed memory bytes.

For the program `test1.c` shown on the previous page, the output should look as follows:

```
~/linklab/part2 $ make run test1
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x55edb72f02d0
[0003]          malloc( 32 ) = 0x55edb72f0710
[0004]          malloc( 1 ) = 0x55edb72f0770
[0005]          free( 0x55edb72f0770 )
[0006]          free( 0x55edb72f0710 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total          33
[0012]
[0013] Non-deallocated memory blocks
[0014]   block              size       ref cnt
[0015]   0x55edb72f02d0     1024       1
[0016]
[0017] Memory tracer stopped.
~/linklab/part2 $
```

You need to keep track of blocks to implement this functionality. We provide a memory block list in `~/linklab/utils/memlist.c/h` so that you don't have to write a linked list yourself (of course, you can write you own if you like). The documentation can be found in the header file.

Extend your tracer from part 1 by copying the source file `memtrace.c` into the directory of this part.

```
~/linklab/part2 $ make run test2
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x55880e9692d0
[0003]          free( 0x55880e9692d0 )
[0004]
[0005] Statistics
[0006]   allocated_total      1024
[0007]   allocated_avg        1024
[0008]   freed_total          1024
[0009]
[0010] Memory tracer stopped.
~/linklab/part2 $
```

Note that:
1) If all memory blocks are deallocated correctly, the program should not print logs about non-deallocated memory block like the output above.
2) You have to include reallocated old blocks in statistics about freed memory.

Part3: Detect and Ignore Illegal Deallocations (30 Points)

Subdirectory: part3/

Some programs call free more than once on a memory block.

```
#include <stdlib.h>

void main(void) {
  void *a;

  a = malloc(1024);
  free(a);
  free(a);
  free((void*)0x1706e90);
}                                                test4.c
```

This programming error results in the process being aborted as shown below:

```
~/linklab/test $ ./test4
  *** Error in `./test4': double free or corruption (top): 0x000000025da010 ***
  ======= Backtrace: =========
  /lib64/libc.so.6(+0x72603)[0x7f09c84f4603]
  /lib64/libc.so.6(+0x77ee6)[0x7f09c84f9ee6]
  ...
  Aborted
~/linklab/test $
```

Extend your tracer from part 2 by detecting and ignoring deallocations of not allocated memory blocks. There are two cases to deallocate not allocated memory: double free and illegal free. Double free means you try to deallocate already freed memory and illegal free means you try to deallocate never allocated memory. For the program `test4.c` shown above, the output should look as follows:

```
~/linklab/part3 $ make run test4
[0001] Memory tracer started.
[0002]          malloc( 1024 ) = 0x55ee279b62d0
[0003]          free( 0x55ee279b62d0 )
[0004]          free( 0x55ee279b62d0 )
[0005]      *** DOUBLE_FREE  *** (ignoring)
[0006]          free( 0x1706e90 )
[0007]      *** ILLEGAL_FREE *** (ignoring)
[0008]
[0009] Statistics
[0010]   allocated_total      1024
[0011]   allocated_avg        1024
[0012]   freed_total          1024
[0013]
[0014] Memory tracer stopped.
~/linklab/part3 $
```

Note that you have to consider cases of double/illegal free in reallocation.