Operating Systems: Reversing ptMalloc2

1 Introduction

In this lab we will try to reverse engineer ptMalloc2 internals by writing a number of programs. We will use the MallocInternals blog post as reference materials for cross-referencing ptMalloc2 data structures and algorithms. The relevant code is available at in the reversing malloc directory.

2 Background:

Before we delve into the internals of the ptMalloc2. Let us briefly cover the APIs required for this lab. In C programming, dynamic memory allocation allows programs to request memory at runtime. The functions `malloc`, `realloc`, `mallopt`, and `mallinfo` help manage memory effectively.

2.1 malloc

The `malloc` (memory allocation) function is used to allocate a specified amount of memory and returns a pointer to the allocated memory.

```
void *malloc(size_t size);
```

Arguments:

- `size`: The number of bytes to allocate.

Return Value:

- If the allocation is successful, `malloc` returns a pointer to the allocated memory. If it fails, it returns `NULL`.

2.2 realloc

The `realloc` function is used to resize previously allocated memory. It may move the memory block to a new location if necessary.

```
void *realloc(void *ptr, size_t new_size);
```

Arguments:

- `ptr`: Pointer to the previously allocated memory block. If `NULL`, it behaves like `malloc`.
- `new_size`: The new size in bytes for the memory block.

Return Value:

- If the allocation is successful, `realloc` returns a pointer to the allocated memory. If there isn't enough available memory to expand the block to the given size, the original block is left unchanged, and NULL is returned.

2.3 mallopt

The `mallopt` function is used to tune the behavior of the dynamic memory allocator by setting various parameters.

```
int mallopt(int param, int value);
```

Arguments:

- `param`: The parameter to configure (e.g., memory allocation behavior settings).
- `value`: The value to assign to the specified parameter.

This function is useful for optimizing memory allocation strategies, though it is less commonly used by beginner programmers.

Return Value:

- `0`: If the `mallopt` function fails to set the option.
- `non-zero`: If the option is successfully set.

2.4 mallinfo

The `mallinfo` function provides information about memory usage in the dynamic memory allocator.

```
struct mallinfo mallinfo(void);
```

Return Value:

- A `struct mallinfo` containing various statistics about memory allocation, such as total allocated space, free space, and memory used by overhead.

2.4.1 struct mallinfo

The `struct mallinfo` contains the following fields:

- `arena`: Total non-mapped space allocated from the system.
- `ordblks`: Number of free chunks.
- `smblks`: Number of fastbin blocks.
- `hblks`: Number of chunks allocated using mmap.
- `hblkhd`: Total space in mmapped regions.
- `usmblks`: Maximum total allocated space.
- `fsmblks`: Total space in free fastbin blocks.
- `uordblks`: Total allocated space.
- `fordblks`: Total free space.
- `keepcost`: Top-most, releasable space.

This function is useful for analyzing memory usage and optimizing performance.

2.5 malloc_trim

The `malloc_trim` function is used to release unused memory back to the system from the heap. It helps reduce fragmentation by attempting to return memory to the operating system.

```
int malloc_trim(size_t pad);
```

Arguments:

- `pad`: The minimum number of bytes to retain in the heap. The function will try to free memory while ensuring that at least `pad` bytes remain allocated. If `pad` is zero, the function attempts to release all unused memory without leaving any minimum allocated memory in the heap.

Return Value:

- `0`: If some memory is successfully released.
- `-1`: If no memory could be freed.

3 Programs

3.1 malloc.c

This program demonstrates dynamic memory allocation using 'malloc'. It prompts the user to enter the number of elements in an array, then allocates memory for those elements dynamically. After checking if the memory allocation was successful, it allows the user to input values, displays them, and finally frees the allocated memory to prevent memory leaks.

3.1.1 Conclusion:

This lab shows that how you can use malloc and free API.

3.2 mallinfo.c

The program demonstrates memory allocation and deallocation using `malloc` and `free`. It uses the `mallinfo2` structure to track memory details before and after allocation.

The `print_mallinfo` function prints memory statistics, such as total heap space, number of allocated chunks, free space, and releasable space. In the `main` function, memory is allocated (1MB) and freed, with memory details displayed before and after each operation.

3.2.1 Conclusion:

This lab shows that how you can use mallopt API. Based on our call, it looks like ptMalloc2 initializes the heap eagerly, which is opposite to what we learned in class, i.e., ptMalloc2 uses lazy initialization.

3.3 mallinfo2.c

Same as mallinfo.c, however, we change the size of allocation to only 10 bytes.

3.3.1 Conclusion:

Changing the size to a lower size shows that instead of increasing hblks (i.e. mmaped memory), ptMalloc2 uses the the heap area instead. In conclusion, ptMalloc2 only uses mmap for larger memory allocations. Furthermore, on free the memory allocated still stays the same.

3.4 mallinfo3.c

Same as mallinfo2.c, however, we change the order of calling mallinfo2. In this program we call mallinfo before calling printf.

3.4.1 Conclusion:

Changing the order shows that indeed ptMalloc2 uses lazy initialization. ptMalloc2 initialized the heap in the earlier programs while calling printf, which internally utilized dynamic memory, hence, initializing the heap.

3.5 mallinfo4.c

So far we have noticed that releasing small memory blocks is kept allocated. Seems like ptMalloc2 caches them for later use. In this program, we use the malloc_trim API to ask malloc to return the memory to the OS if possible.

3.5.1 Conclusion:

Calling malloc_trim forces ptMalloc2 to return as much memory as possible back to the OS, resulting in much smaller free space, however, the size of allocated space still says the same due to caching. In class, we were able to get this memory back from allocated to free. Looking at the malloc algorithm, can you figure out how can avoid this optimization? Hint: Vary the size of the allocation.

3.6 mallinfo5.c

Same as mallinfo4.c, however, we increase the allocated memory back to 1MB.

3.6.1 Conclusion:

Since large blocks are mmaped, this doesn't cause any effect on the results.

3.7 size_re.c

This program allocates memory using `malloc` based on user input, then prints the requested size, the actual allocated size using `malloc_usable_size`, and the internal size field of the memory block. It continuously prompts the user for a memory size, allocates the memory, retrieves the actual size (adjusted for alignment), and frees the memory after printing the details. The program runs in an infinite loop, allowing multiple allocations.

3.7.1 Conclusion:

This code reveals a number of facts about the ptMalloc2 implementation:

- ptMalloc2 caches the freed memory and returns the same pointer if the next allocation can be served by the same block.
- The minimum allocation for ptMalloc2 is 32 bytes, out of which, 24 bytes are usable.
- ptMalloc2 is 16-bytes aligned.

4 Takeaways:

Please try to play with allocation size, number of allocations, and order of allocations to infer more details about the internals of ptMalloc2. There is extra code in the bonus directory for you to explore.