CMPSC 473 - Fall 2019 - Project 2: Slab Allocation and Defenses

1 Dates

- **Out:** *September 23, 2019*
- **Due:** *October 11, 2019*

2 Introduction

In this project, you will write dynamic storage allocator for C programs, i.e., your own version of the `malloc` and `free`. You will implement a particular type of allocator called a *slab allocator* described below and setup the ability to check pointers for memory you allocate to prevent some attacks: (1) buffer overflow; (2) use-after-free; and (3) type confusion.

The only file you need (or should) modify is *cmpsc473-mm.c*. For the allocator, you will implement the following functions (see functions for defenses in Section 6).

- int mm_init(void);
- void *my_malloc(unsigned int size);
- void my_free(void *buf);

You are encouraged to define other helper functions to modularize your code.

3 Slab Allocation

Slab allocation is briefly described in the "Three Easy Pieces" book in Section 17.4. The idea is that the heap for a slab allocator is designed to allocate objects of predetermined sizes, e.g., for commonly used data structures like tasks (processes) and files.

A slab allocator has a particular structure. Please use the data structures for `heap_t`, `slab_cache_t`, `slab_t`, and `bitmap_t` defined in *cmpsc473-mm.h*. Some bit operations are provided. See `WORD_OFFSET` and `BIT_OFFSET` in *cmpsc473-mm.h* and `set_bit`, `clear_bit`, and `get_bit` in *cmpsc473-mm.c*).

The relationship among these data structures is shown in Figure 1.

A single heap of type `heap_t` is the root of the slab allocator structure. A heap consists of a contiguous sequence of pages that may be used for allocating data for each structure from the field `void *start` for size `unsigned int size`. Your slab allocator manages heap memory in slabs, which are the same as pages (i.e., same size as a memory page).

See Figure 2 to see an example of how pages may appear in the heap memory. Use the heap's `bitmap` field to track which pages are free and which are allocated.

There are three slabs caches of type `slab_cache_t` (see *cmpsc473-mm.h*) in your slab allocator's heap: one for each of the structs A, B, and C. Please use your *cmpsc473-format-X.h* file from Project 1 (where X

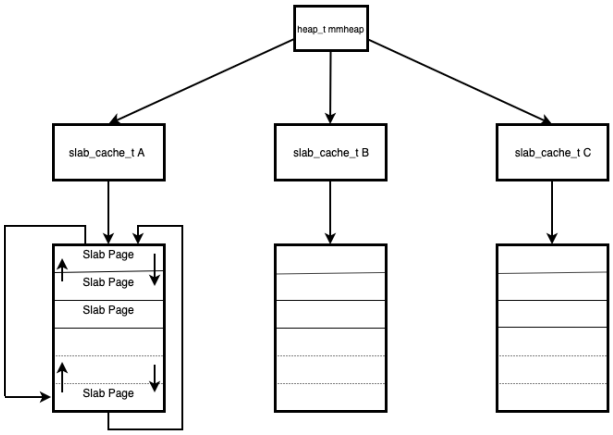

Figure 1: Slab allocator data structures and relationships

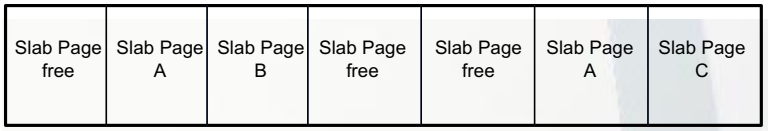

Figure 2: Layout of slab pages in the heap

is your format file number) to define the object format for the objects that you will be allocating memory for from your slab allocator.

The three slab caches each store a circular, doubly-linked list of slabs from a single reference to one slab at `slab_t *current`. The slab cache tracks slabs by the count of slabs allocated (unsigned int ct) and the size of the data structures (for struct A, B, anc C from your *cmpsc473-format-X.h* file). The slab cache also stores stwo function pointers to cache-specific functions for allocating metadata (for canary and free count, see Section 6) and for checking canaries (see Section 6).

The slabs of are type `slab_t`. The slabs store a state: `SLAB_EMPTY`, `SLAB_PARTIAL`, and `SLAB_FULL` defined in *cmpsc473-mm.h*. Each slab has a start address (*void *start*), which is always page-aligned since we are allocating slabs in pages. There are `next` and `prev` pointers to slabs in the circular, doubly-linked list of slabs (see Figure 1.

The format of an individual slab page is shown in Figure 3. Note that the slab data structure is always at the end of the slab page. From the start of the slab page, a sequence of objects (called `allocX_t` of type "struct X") of size `num_objs` are available for allocation. You must align objects on 16-byte boundaries, as we will use this restriction to enable implementing a defense below. A count (`unsigned int ct`) of objects have been allocated from the slab at any time. Please track this count.

There are two fields that describe the object's size in each slab. One `real_size` stores the size of the structure being stored (i.e., `sizeof` for struct A, B, or C). The other `obj_size` stores the size of the slab object. Each slab object includes the structure data and two metadata fields for the free count (`ct`) and canary (`canary`) discussed in Section 6 and must be 16-byte aligned as described above.

4 Requirements for Allocator Functions

Implement the following functions. Note that you can get these functions below running without adding or checking the defenses below. Thus, the project can be programmed incrementally once you have basic memory allocation working.

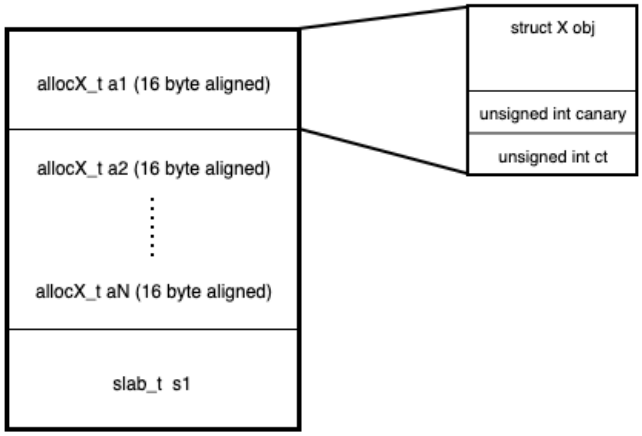

Figure 3: Layout of an individual slab page in memory

- `int mm_init( void )`: Before calling `my_malloc` or `my_free` you need to initialize the heap to handle these requests. Please allocate 1M for slab pages (using the standard memory allocation functions). Please make sure that the heap's slab pages are all page-aligned. You should also initialize the canary value for the process at this time as described in Section 6.

- `void *my_malloc( unsigned int size )`: The `my_malloc` routine returns a pointer to an allocated block payload of at least size bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk. You should allocate an object from a slab page to fulfill this request. If you have no free objects in all your slab pages for a cache, you must allocate a new slab page for the cache.

- `void my_free( void *buf )`: The `my_free` routine frees the block pointed to by `buf`. It returns nothing. This routine is only guaranteed to work when the passed pointer (buf) was returned by an earlier call to `my_malloc`. You should make the object whose start address corresponds to this `buf` pointer free. Other addresses should fail to deallocate the object. If you free all the objects from a slab page, you must free the slab page for use by another cache, unless that page is the only slab page in the cache.

5 Attacks on Heap Data

In recent years, heaps have become the main target for attacks in processes. This is partly because defenses have been added to prevent attacks on stack memory, and partly because standard heaps typically store both the application data and the heap metadata on the heap. Attackers can use memory errors (e.g., overflows) to overwrite the heap metadata enabling attacks. In our case, only the `slab_t` metadata is stored in the slab pages of a slab allocator.

We will examine three types of attacks:

- **Overflows**: We want to prevent a write to an object field from writing another object's data. Such an attack may compromise the integrity of another object or the `slab_t` metadata.

- **Type Confusion**: We want to prevent an adversary from being able to use a pointer to memory of one type (say, `struct B`) to reference memory of another type (say, `struct A`). A program vulnerability may allow an adversary to control a pointer value, and the adversary could leverage this vulnerability to reference an object of another type to change its field values using field operations on the first type.

- **Use-After-Free**: In this attack, an adversary obtains a pointer to memory that can be used in a statement after the memory is freed. If the memory is reallocated, the adversary could reference this newly allocated memory - perhaps of a different data type if the entire slab has been assigned to another cache.

6 Requirements for Preventing Attacks

You will develop modest defenses for each of these three attack types.

Each of these "check" functions returns an integer value. You must return "0" on success and any other number on failure (typically "-1" is used though).

6.1 Canaries

A *canary* is an unpredictable (i.e., pseudorandom) value that is used to detect overflows. In this project, each `allocX_t` includes a canary field after the object data. This is to detect an overflow on a write to an object field that writes beyond the end of the object.

You need to implement the following two functions to apply the canary as a defense.

- `void canary_init( void )`: This function should be run before any `my_malloc` or `my_free` call to initialize the pseudorandom canary value. The canary value must differ on each run of the process annd be pseudorandom. The canary value must be set for each allocated object.

- `int check_canary( void *addr )`: You need to define a function to check the canary value of a object reference (`addr`) and assign that function to the appropriate `slab_cache`. You may define one function for all types or one function each. We will invoke this function after any `write` operation.

6.2 Check Pointer to Object Size

To prevent type confusion attacks, you must implement the function `int check_type( void *addr, char type )` to check that the pointer `addr` is an appropriate reference for an object of type `type`, which is a one character value of either A, B, or C for the corresponding structs. We will test by asking for pointers of the wrong data type for particular object locations. You must determine whether it is acceptable for an object reference to be cast to the requested type. Since there may be many pointers and references, you want to do this efficiently.

6.3 Free Count Tracking

A free count tracks the version of the allocation to prevent use-after-free attacks. Each object of type `allocX_t` is associated with a field for the free count (`ct`). In addition, `my_malloc` must encode the free count of the pointer returned in the 4 low-order bits of the pointer returned - remember all allocated objects are 16-byte aligned, so we don't need those bits for addressing..

You must implement the function `int check_count( void *addr )` to check that the free count of the object referenced (at `addr`) must comply with the free count of the pointer. Note that the free count of the pointer can be no larger than 15 (since you only have 4 bits), so you must figure out how to use the reference count stored in the object to do the check effectively.

7 Doing the Project

Perform the following tasks to complete the project.

7.1 Download Tarball

The tarball for Project 2 is available from Canvas from `http://www.cse.psu.edu/~trj1/cmpsc473-19f/p2.tgz`. This tarball includes my format file *cmpsc473-format-1.h*. You will use your format file from project 1 instead.

7.2 Project Tasks

You will then perform the following steps.

- **Task 0**: As in Project 1, please change the SNUM value in the Makefile to your format file number from Project 1.
- **Task 1**: As in Project 1, please change the header file references to *cmpsc473-format.1.h* to reference your format file.
- **Task 2**: Develop the three slab allocator functions: `mm_init`, `my_malloc`, and `my_free`. You may develop subroutines for these functions as you see appropriate.
- **Task 3**: Develop the functions for the three types of defenses for the slab allocator: canaries, type checks, and free counts.

  You can also using the `malloc_fn` and `free_fn` function pointers to define cache-specific (object-specific) functionality for malloc and free to enable defenses, and use the `canary_fn` for cache-specific (object-specific) checking of the canary values. If you don't need them, you are not required to use them.

8 Testing

We will test your submission on machines in the Linux lab in W204 Westgate. The machines are named cse-p204instXX.cse.psu.edu, where XX is a number from 01 to at least 40.

We will test your program by supplying input files containing command sequences consisting of commands for memory allocation/deallocation and pointer use, which may or may not indicate an attack. All the commands that we will execute are provided by the `process_cmds` function in the *cmpsc473-p2.c* file. DO NOT EDIT THIS FILE.

The memory allocation deallocations are:

- **malloc** *A/B/C id-start id-end*: Allocate objects of either type (struct) A, B, or C from index *id-start* to *id-end*. We will refer to these objects subsequently by ID to use the associated pointer values in other commands. We will pass `my_malloc` requests for allocating memory for the associated type for each ID.

- **free** *id-start id-end*: Free all objects of IDs in the range between *id-start* and *id-end*. We will pass `my_free` requests to free the objects associated with those IDs allocated using `my_malloc`.

If you can support execution of these two commands as described above, you will complete the memory allocation portion of the project.

For performing operations using the allocated references, the following commands will be provided.

- **write** *id bytes*: The *write* command writes *bytes* number of bytes to the pointer associated with ID *id*. Write commands may or may not write beyond the end of the available object memory, so we run `check_canary` to detect whether the canary's integrity is preserved.
- **save** *A/B/C id*: The *save* command stores a pointer of type either (struct) A, B, or C for later use for the pointer associated with ID *id*. Such an operation may try to save a pointer allocated for one type to a pointer allocated to another type, so we run `check_type` to prevent this kind of type confusion.
- **use** *(saved reference)*: The *use* operation will perform a one-byte read from the pointer stored in the save operation. If the memory pointed to by the pointer has been freed since the save, we want to prevent such an operation, so we run `check_count`.

You should SSH into those machines to verify that your code works. We developed and tested the project code on those machines, so should work fine, but it is up to you to make sure.

You will need to speak to the CSE IT folks if you do not have access to those machines.

9 Questions

Please answer the following questions regarding slab memory allocation and defenses.

- Does slab allocation create either external or internal fragmentation?
- How much memory is not available for allocation given the slab allocator implementation for your objects of struct A per page?
- Describe an overflow attack that would work even given our implementation of canaries.
- How did you implement `check_type` to prevent type confusion attacks?

10 Deliverables

Please submit the following:

1. A tarball created using "make tar" from the directory *p2-assign*, creating a file *p2-assign-X.tgz*, where X is your format number as in Project 1.
2. A file in PDF format providing answers to the questions above.

11 Grading

The assignment is worth 60 points total broken down as follows.

1. We can build and run what you have submitted without incident (9 points).
2. Slab allocator tests complete correctly (24 points)
3. Slab allocator performance and memory utilization are within bounds (12 points). Performance must be within 90% of course standard and memory utilization must not incur more than 25% excess memory in slab pages used by caches.
4. Slab allocator defenses complete correctly (15 points)