Heap exploitation

G. Lettieri

15 November 2023

1 Introduction

The heap is the area of a process' memory from which the C library's `malloc()` function and the C++ **`new`** operator allocate space. These functions/operators return a pointer to the allocated memory, which the program can then use freely. However, no bound checks are usually performed and the program may inadvertently write beyond the allocated memory. Such overflows in heap-allocated data can overwrite other data and also *heap metadata*. An attacker can exploit these bugs to execute arbitrary code, similar to stack overflows.

In addition, languages such as C and C++, where the programmer must also remember to `free()`/**`delete`** the memory that she has allocated, also create new opportunities for bugs. We will examine the following two categories of bugs:

- *double-free*, where the programmer accidentally frees the same memory twice;
- *use-after-free*, where the programmer frees a data object, but then accidentally continues to access it.

2 Heap implementation

The kernel has only a very coarse view of a process' heap: it only remembers its boundaries. The heap typically grows from lower to higher addresses. The lowest address is chosen by the kernel (if ASLR is not enabled, the kernel simply chooses the first properly aligned address after the end of the `.bss`). The highest address is chosen by the process itself, using the `brk()` system call or, more likely, the `sbrk()` function, which internally calls `brk()`. The kernel will allow memory accesses within the heap boundaries without any interference, while accesses outside the boundaries (at addresses that are not mapped to anything else) will cause a segmentation fault.

This heap region can be used by the process as it wishes. Typically, however, programmers use an heap management library, such as the one included in the C library, to allocate/dealloc memory from the heap region. The C malloc libraries implement at least these two functions:

- **void** `*malloc(size_t sz)`: allocate a data object of `size_t` bytes;
- **void** `free(`**void** `*p)`: free a previously allocated item.

There are many libraries that implement these functions, but the one implemented in the GNU C library (glibc) is probably the most used on Linux systems, as it is the one available by default. This library was originally based on Doug Lea's malloc, also known as *dlmalloc*, a freely available, state-of-the-art malloc implementation. Today, glibc's malloc is still derived from Doug Lea's malloc, but with many enhancements. In the following we will only consider dlmalloc, but most of what we say applies to glibc as well.

2.1 Doug Lea's malloc

The purpose of any malloc library is to remember which parts of the heap have been allocated and which parts are still unused. These parts are called *chunks* in dlmalloc. When `malloc()` is called, the library should find a sufficiently large free chunk and mark it as occupied. If the free chunk is larger than requested, the library can also split the chunk, allocating the requested part and creating a new free chunk to hold the rest. When `free()` is called, the library should mark the chunk as free and possibly merge it with any adjacent free chunk to create a larger free chunk. This process, called *coalescing*, is necessary to reduce the risk of *fragmentation*, i.e., the creation of many small free chunks that cannot be used to serve larger requests.

Doug Lea's malloc implementation is based on the following ideas, as suggested by Donald Knuth in *The Art of Computer Programming*:

- embedded metadata;
- boundary tags.

By "embedded metadata" we mean that the descriptors of the chunks are stored in the same heap region as the chunks themselves. In particular, each chunk, whether free or used, is preceded by a *chunk header* which stores information about the chunk and, in particular, remembers its size. This design is almost forced by the fact that the `free()` function only gets the pointer to the chunk to free, but does not say what the size of the chunk is. The library must then recover the size itself, and an easy way to do this is to store it in memory just before the pointer. The pointer returned by `malloc()`, which the user should then pass to `free()` is called the *user pointer*. The pointer to the header is instead called the *chunk pointer*, and can be obtained from the user pointer by subtracting the header size (8 bytes on 32 bitxs systems and 16 bytes in 64 bit systems). The part of the chunk that the user should be able to access (i.e., the chunk memory minus the header) is called the *user memory*. The header is not the only metadata stored in the heap. Free chunks are kept in a list (actually one of several disjoint lists) and the chunk's list pointers are stored in the otherwise unused chunk's user memory itself, right after the header. Note that this means that dlmalloc will always allocate chunks of a size that can

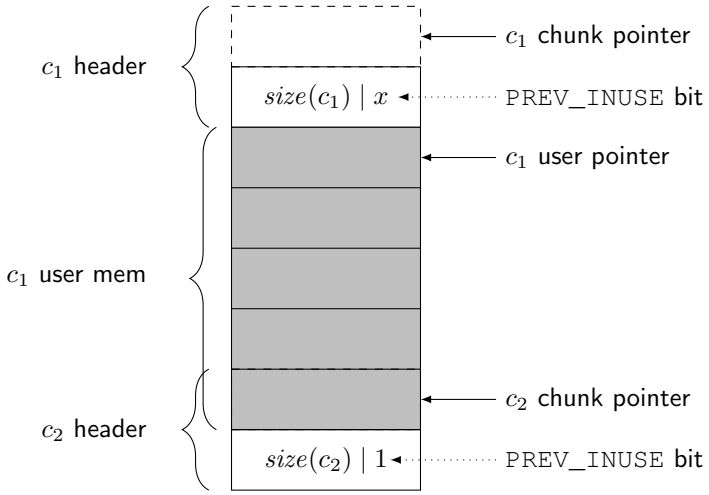


Figure 1: Memory layout for an in-use chunk c_1, followed by a chunk c_2. Each line is 4 bytes on 32b systems and 8 bytes on 64b systems. The header of c_1 contains the size of c_1 and the `PREV_INUSE` bit with either $x = 1$ or $x = 0$ depending on whether the chunk before c_1 (not shown) is in use or free. The `PREV_INUSE` bit in the header of c_2 is set to 1 because c_1 is in use. Note that the user memory of c_1 extends to the first line of the header of c_2. Note also that the chunk pointers always point to the beginnig of the header, even if the first line of the header is part of the user memory of the previous chunk.

contain these pointers, even if you request a smaller size. The library needs two pointers for each chunk, called `fd` for forward and `bk` for backward, to implement a double-linked list, so the minimum size of a chunk is 16 bytes on 32 bit systems ($4 + 4$ for the two pointers, plus the 8 byte header) and 32 bytes on 64 bit systems. For alignment reasons, dlmalloc also only creates chunks that are multiples of 8 (32b) or 16 (64b) bytes.

By "boundary tags" we mean that each free chunk is bounded by two tags that store its size: one before the user pointer (head) and one at the end of the user memory (foot). In dlmalloc, these tags always store the total size of the chunk (header plus user memory). The head tag is just the one stored in the chunk header. The foot tag is useful when the chunk is free and the next adjacent chunk is also free. Let us call the two adjacent chunks c_1 and c_2, where c_1 already free. The free operation on c_2 should coalesce c_2 with c_1, but it must first be able to find the header of c_1. The foot tag of c_1 will then give the offset that must be subtracted from the chunk pointer of c_2 to obtain the chunk pointer of c_1. In dlmalloc, the foot tag of c_1 is actually part of the header of c_2. However, since the foot tag is only needed when c_1 is free, the tag memory becomes part of the user memory of c_1 while c_1 is in use. The header of c_2 contains a bit, the `PREV_INUSE` bit, which is set when c_1 is in use and reset when it is free. Because of the alignment constraints, the lower bits of the chunk

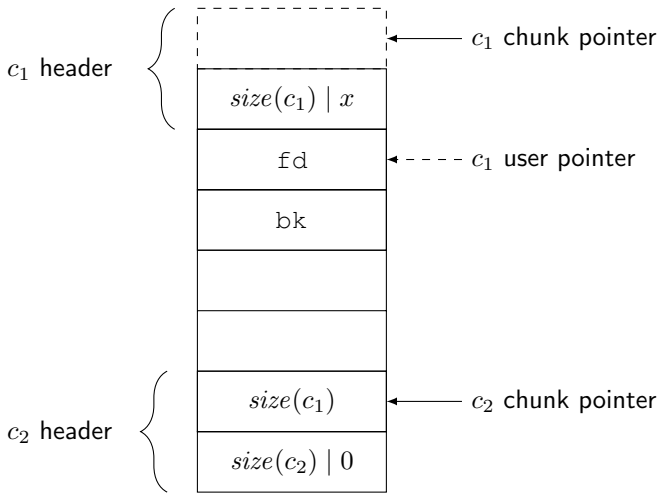


Figure 2: The same chunks as in Figure 1 after c_1 has been freed. The first two lines of the user memory of c_1 have been reused to store the `fd` and `bk` pointers that put c_1 into a doubly-linked list of free chunks. The last line of c_1 user memory now contains c_1's foot boundary tag, which is actually part of c_2's header.

size are always zero and can be reused to store some flags. In particular, the `PREV_INUSE` bit is stored in the least significant bit of the size field. Figure 1 shows the state of the chunks when c_1 is in use, while Figure 2 shows the same chunks when c_1 is free.

2.2 Fastbins

The basic scheme described above is enhanced in a number of ways. We will focus on just one of these: the fastbins. The fastbins are a cache of small chunks that have been freed and can be recycled in their current form. When the `free()` function receives a chunk whose size qualifies it for recycling, it doesn't free it (and therefore doesn't merge it with any neighbouring free chunk), it just puts it, as-is, into the fastbins. The fastbins are an array of singly linked lists of reusable chunks. There is a separate list for each possible size, starting with the minimum size and going up to a maximum size. For example, on 64b systems there may be a list for 32 byte chunks, another for $32 + 16 = 48$ byte chunks, then 64 byte chunks and so on up to, say, 144 byte chunks. The head pointers of these lists are stored in an array that is part of the main heap data structure, called the *arena*. The arena itself is stored at a know address within the library. Chunks with eligible sizes are pushed to the front of the appropriate fastbin list. When `malloc()` needs to allocate a chunk of one of the fastbin sizes, it first looks in the corresponding fastbin list. If the list is not empty it pops the first element and returns it, otherwise it continues with the normal

search. Fastbins are important for performance because they skip all the chunk splitting and merging operations, reducing `malloc()` and `free()` operations to just a few instructions.

3 Metadata Exploitation

An overflow in the user memory of a chunk can overwrite the header of the next chunk in the heap. Probably the first public exploit of this type of bug was published by Solar Designer in 2000[1]. The idea was to abuse the `unlink()` macro in the dlmalloc `free()` function. This macro is called when `free(a)` tries to merge the `a` chunk with the neighbouring free chunks. The macro removes the free chunk from its doubly-linked list by writing to the `fd` and `bk` pointers. In particular, if `p` is the pointer to the chunk to be extracted, the macro performs the following two assignments:

```
p->fd->bk = p->bk;
p->bk->fd = p->fd;
```

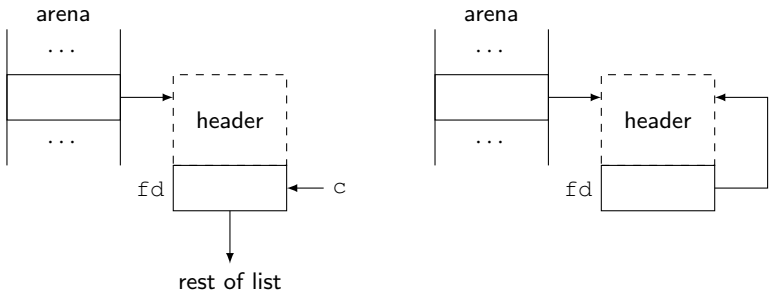
Now, by overwriting the header of the chunk, we can create a fake previous free chunk pointing to some memory controlled by the attacker. So the two assignments above will use the `p->fd` and `p->bk` pointers provided by the attacker, let us call them x and y. The first assignment writes y to address x plus the offset of `bk` in a chunk, call it o_b (three lines, see Figure 2), which gives the attacker an arbitrary write primitive. The possible uses of this primitive are actually limited by the second assignment, which also writes x to address y plus the offset `fd`, calling it o_f (two lines). So both $x + o_b$ and $y + o_f$ must point to writable memory. Solar Designer's exploit makes $x + o_b$ point to one of the GNU malloc hooks, such as `__free_hook` and `__malloc_hook`. These hooks are called by glibc malloc whenever `free()` or `malloc()` is called, and can be used by programmers to trace or extend malloc functionality. The exploit overwrites one of these hook with a pointer y in memory controlled by the attacker, in the heap itself, where some shellcode had previously been injected. The hook would then be overwritten when the program called `free()` on the overwritten chunk, and the shellcode would then be executed the next time the program tried to call the hook (see the exercises for more details).

This attack doesn't work on modern systems for a couple of reasons:

- writable memory is no longer executable, thanks to the NX bit, so there is nothing useful to assign to either $x + o_b$ or $y + o_f$;
- the current `unlink()` macro (now a function) does some integrity checks on the pointers before using them.

This doesn't mean that there are no other possible heap exploits. In fact, many more have been found since then, and many other integrity checks have been

[1] https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability



(a) After first `free(c)`. (b) After second `free(c)`.

Figure 3: Loop in a fastbin list created by a double-free bug.

added to malloc implementations, often in response to a new type of attack[2]. We examine just one example below.

3.1 Double-free and fastbins

Suppose a victim program contains a double-free bug on chunks whose size in within the fastbin range. Suppose that `c` is such a chunk. The first `free(c)` will push the chunk in front of its fastbin list. This is done by copying the fastbin head into the `fd` field in the chunk, and copying the chunk pointer into the fastbin head (Figure 3a). Now consider what happens when a second `free(c)` is called: the fastbin head, now pointing to (the header of) `c`, is copied into the `fd` field of `c`, and the chunk pointer of `c` is copied (again) into the fastbin head (Figure 3b). This creates a loop in the fastbin list, whichk can be exploited as follows

- let the victim program to allocate a new object from the fastbin; this returns `c`;
- cause the victim program to write a pointer of the attacker's choice into the user memory of `c`; this overwrites `fd`;
- cause the victim program to allocate another object from the fastbin; this returns `c` again, and *copies the attacker's pointer into the fastbin head*;
- causing the victim program to allocate another object from the fastbin; this returns *the attacker's pointer* (plus the size of the header);

Now, any attacker-controlled write to the user memory of the last chunk will write to the location chosen by the attacker. See the exercises for more details.

[2]Also, counter-attacks to many of the integrity checks have been found, including the one in `unlink()`.

4 Non-metadata exploitation

Let us now consider an example of how dynamic memory bugs can be exploited even without touching or overwriting any heap metadata.

Probably the most common bug found in programs using dynamic memory is the use-after-free bug: one part of the program frees an object, call it o_1, while some other part of the program still holds a pointer to o_1. This "dangling" pointer may be used later assuming that it still points to o_1, but o_1's memory may actually have been recycled and now stores a completely unrelated object, call it o_2.

This bug can be exploited in many ways, but a particularly favourable scenario for attackers is the following:

- the o_1 object contains a function pointer;
- the attacker controls a field of o_2 that overlaps a function pointer in o_1.

Now, when the victim program calls the function pointer in o_1, it will transfer control to the location chosen by the attacker (e.g., a one-gadget).

A scenario similar to the above can be easily observed in C++ programs. C++ objects that define virtual functions (including a virtual destructor) contain a *vtable* pointer in their first locations. The vtable pointer points to a table of virtual function pointers, one for each virtual function defined in the object's class. When the C++ program needs to call a virtual function member of the object, it actually makes an indirect call through one of the pointers in this table. It should be clear that if the conditions of a use-after-free attack are met, the attacker can overwrite the vtable of an object and make it point to an injected virtual table with pointers to locations of the attacker's choice. Examples can be found in the exercises.