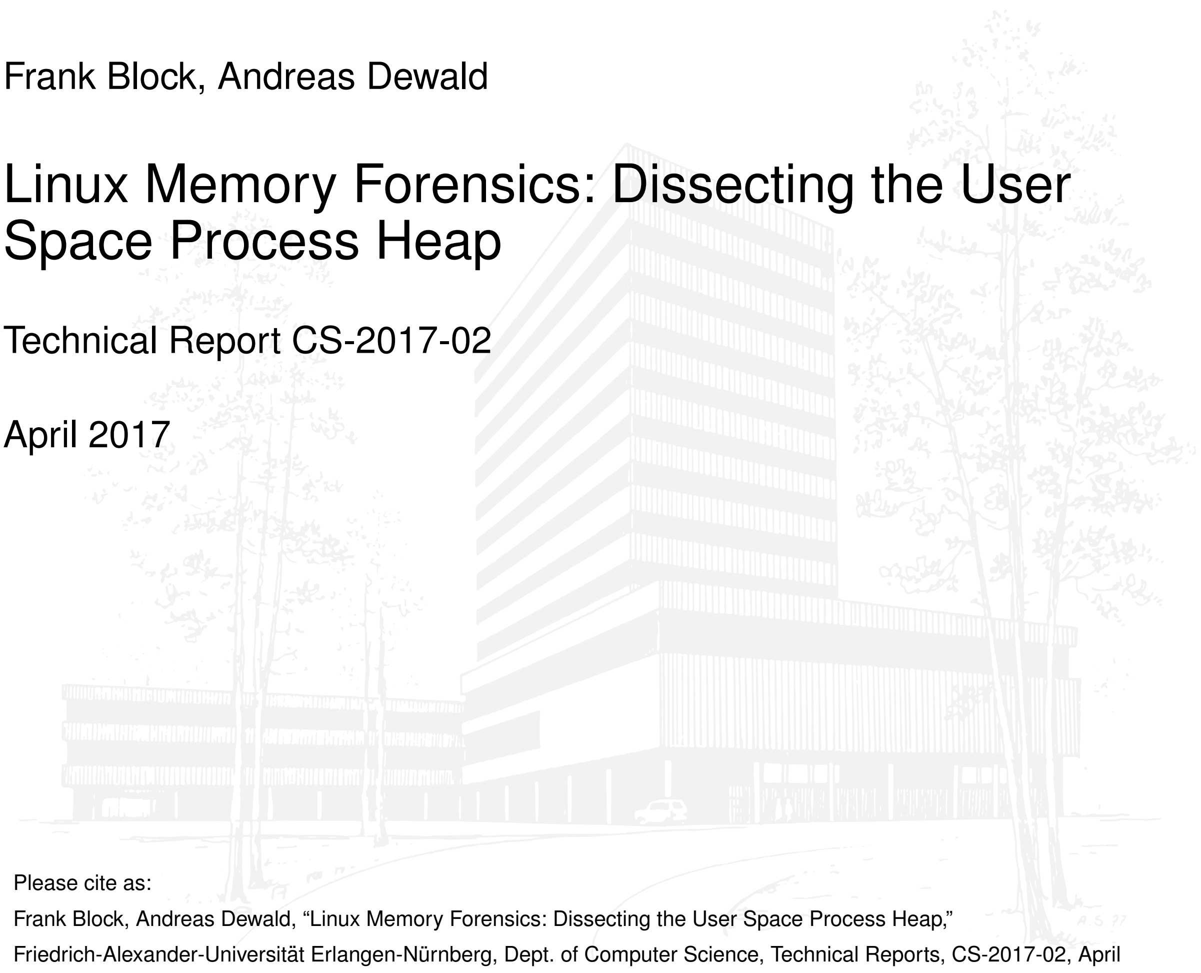
Department Informatik

Technical Reports / ISSN 2191-5008



Frank Block, Andreas Dewald

Linux Memory Forensics: Dissecting the User Space Process Heap

Technical Report CS-2017-02

April 2017

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.cs.fau.de

Linux Memory Forensics: Dissecting the User Space Process Heap

Frank Block, Andreas Dewald
IT Security Infrastructures
Dept. of Computer Science, University of Erlangen, Germany
ERNW Research GmbH, Heidelberg, Germany

Abstract—**The analysis of memory during a forensic investigation is often an important step to reconstruct events. While prior work in this field has mostly concentrated on information residing in the kernel space (process lists, network connections, and so on) and in particular on the Microsoft Windows operating system, this work focuses on Linux user space processes as they might also contain valuable information for an investigation. Because a lot of process data is located in the heap, this work in the first place concentrates on the analysis of Glibc's heap implementation and on how and where heap related information is stored in the virtual memory of Linux processes that use the Glibc heap implementation. Up to now, the heap was mostly considered a large cohesive memory region from a memory forensics perspective, making it rather hard manual work to identify relevant information inside. We introduce a Python class for the memory analysis framework Rekall that is based on our analysis results and allows access to all chunks contained in the heap and their meta information. Further, based on this class, six plugins have been developed that support an investigator in analyzing user space processes: Four of these plugins provide generic analysis capabilities such as finding information/references within chunks and dumping chunks into separate files for further investigation. These plugins have been used to reverse engineer data structures within the heap for user space processes, while illustrating how such plugins ease the whole analysis process. The remaining two plugins are a result of these user space process analyses and are extracting the command history for the *zsh* shell and password entry information for the password manager *KeePassX*.**

This report is an extended version of our paper published at DFRWS USA (Block and Dewald, 2017).

I. Introduction

As the memory represents the current state of a running system, it contains for example information about browser history, entered commands, active network connections, loaded drivers as well as active processes and hence allows detailed insights into previous activities.(Ligh et al., 2014) While much of this information is located in the kernel space and can be examined with existing solutions for various operating systems such as Rekall (Google Inc, 2016c) and Volatility (Foundation, 2016), there is also a lot of information located in the user space that might be of interest in a forensic investigation, too. The heap of a user space process for example is typically a rich source of various kinds of data and, depending on the concrete application, might contain credentials, IP addresses/DNS names or a command history. However, this information is, at least in the context of Linux, not yet easily extractable.

To efficiently and reliably identify and extract this information, the investigator requires a view of the heap that is the same or at least similar to the one the process has: Knowledge about where the data is located, what kind of data is stored at a specific position and which amount of memory a specific data portion occupies. Otherwise, the investigator can only work with the heap as one large memory region, with all pieces of information located inside without any known structure.

A. Motivation

In the context of Linux processes, the focus of user space analysis was in the past limited to searches for specific patterns within the complete process memory or the whole heap/stack. For example, in order to reconstruct the command history for the Linux *bash* shell, Rekall's *bash* Google Inc (2016a) plugin searches the heap for a hashtag followed by a Unix timestamp in string format (e.g. `#1471572423`) (Ligh et al., 2014, 630 ff.). If however the information of interest is not marked in an easily detectable way, a simple pattern matching will fail.

When taking a scenario in which the investigator identifies a certain string in memory and tries to identify references to it, this search might fail even though there are (indirect) references. This could be the case if the string is part of a struct or object and is not located at the beginning, while the pointer of interest references the struct instead of the string directly (see Section V-B for an example). To be able to find those references, the beginning of that struct must be known, which requires knowledge about the size of its fields and the location of the string within that struct. However, these details are normally not available in a black box analysis.

B. Contributions

In this work, we make the following contributions:

- We analyzed the Glibc heap implementation and summarize the information that enables an investigator to perform a manual heap analysis or implement his or her own tool for this purpose. In particular, we explain how heap structures are arranged and where they are typically located in memory (Section II-D).
- We demonstrate that some chunks might hide somewhere in the memory, for which we propose an algorithm to retrieve them (Section III-F).
- These insights have been used to develop a Python class called *HeapAnalysis*, which can be used to implement specific heap analysis plugins.
- Based on this class, we developed plugins that support the investigator in analyzing the heap and its chunks.
- We explain how data of user space processes can be analyzed by applying these plugins to gather relevant information (similar to the analysis done by Cohen (2015) for a Windows user space process).
- A result from this analysis are two further plugins: The first one gathers all executed commands from the heap of a *zsh* shell (Version 5.2) process and the second extracts the title, username, URL and comment field of all retrievable password entries from the heap of the password manager *KeePassX* (version 0.4.3).

The *HeapAnalysis* class and all mentioned plugins support x86 and x64 architectures.

C. Outline

This work is structured as follows: Section II covers details about the heap of user space processes that use Glibc's heap implementation. In Section III, we provide an overview of the developed heap analysis plugins, Section IV covers the evaluation of those plugins, Section V provides a detailed analysis of applications, and Section VI concludes this paper.

D. Related Work

Cohen (Cohen, 2015, p. 1138) states that "the analysis of user space applications has not received enough attention so far". This not only underlines the motivation of this work, but also the reason for which there is not much literature about that specific topic. Existing literature in the field of Linux memory forensics mainly covers kernel related topics such as the work of Urrea (2006), Case et al. (2010) and Ligh et al. (2014).

The few exceptions are the work by Leppert (2012) and Macht (2013), who both focused on the Android operating system of mobile devices, which is Linux based, and analyzed applications and their heap data. However, their analysis concentrated primarily on serialized Java objects contained in the heap and not on the way heap objects are managed. Other exceptions are the already existing plugins *cmdscan* Google Inc (2016b) and *bash* Google Inc (2016a), which extract the command history from Windows' *cmd* and Linux's *bash* shell, respectively. These plugins, however, leverage the fact that in those cases it is possible to identify the information by only looking at the heap as one large memory region. Another related work is the analysis of Notepad's heap (Ligh et al., 2014, p. 223). This is to the best of our knowledge the only example that uses any heap details and, as most of the prior work, is related to Windows too.

Outside the scope of forensics, there has been fundamental research on the heap and, in particular, on the heap of Linux processes and how it is managed. Especially the research by Ferguson (2007) serves a solid understanding about Glibc's heap implementation. This previous research, however, focused more on the ways the heap can be exploited and hence does not provide enough information to reliably gather all relevant information from the heap in a memory forensics scenario.

The work of Cohen (2015) is the first to approach this research gap with a set of analysis tools for the Windows Operating system. While there are some similarities between Windows' heap implementation and the one from Glibc (for example in both cases an allocated chunk is preceded by a struct containing at least the chunk's size), they differ in the details.

II. Glibc Analysis

In this section, we present the results of our analysis of the Glibc heap implementation from a memory forensics perspective.

A. Different Heap Implementations

There are various heap implementations available, most of them used in the context of a certain operating system or application. The reasons that there are multiple implementations are amongst others an increase in functionality requirements over the last decades (e.g. support for multi-threading (Gloger, 2006)) and the pursuit for performance improvements (Ghemawat and Menage, 2015). The following list shows some of those implementations.

dlmalloc An early implementation, which was also the basis for *ptmalloc2*.

ptmalloc2 Improved implementation of *dmalloc*, which was used as a basis for Glibc.

Glibc malloc The implementation covered in this work.

jemalloc Mostly used in FreeBSD and Mozilla products.

tcmalloc Mainly used for Google's browser (Chrome).

Low Fragmentation Heap Part of the heap implementation used for Windows Vista and later.

The reason why most of the heap implementations contain the word *malloc* in it, is because of the same named function. This function is the core of any of those implementations as it is responsible to allocate a given amount of bytes from the heap and to return it to the caller. The latter part is realized with a pointer to that memory space that the caller can use to access and store data in it. Those implementations are hence sometimes also referred to as memory allocators.

One of the first implementations was *dlmalloc* by Doug Lea (Lea, 2006). As it does not support multi-threading, it was improved by Wolfram Gloger with *ptmalloc2* (Gloger, 2006). *ptmalloc2* is used in many Linux/Unix distributions but may also be used in applications compiled under mingw or cygwin. (Cohen, 2015, page 1139)

As the usage of a certain heap implementation is not bound to a specific operating system, a user space process can easily choose to use a different one (Cohen, 2015, page 1139). This might either be realized by using one, offered by the operating system, or by already including such an implementation within the application (e.g. the case with *Mozilla* products such as *Firefox*). Such processes might then however not be analyzable using the information or tools introduced in this work.

Microsoft uses yet another implementation which has already been analyzed by Valasek (2010). Their design consists of a *back-end* and *front-end allocator*. The *back-end allocator* mainly initializes the heap and provides large memory regions while the *front-end allocator* is responsible for splitting those large regions in smaller ones and managing them. The *front-end allocator* implementation, used in Windows Vista and later, is called *Low Fragmentation Heap* and is only used if the implementation decides it is necessary. Otherwise, only the *back-end allocator* will serve the application with memory.

The implementation examined in this work is Glibc version 2.23 (Free Software Foundation Inc., 2016), which is based on Wolfram Gloger's *ptmalloc2* (Gloger, 2006). As written in a comment of the Glibc source code, their implementation of *malloc* and further functions have been substantially changed and do not share many similarities anymore (see line 22 in Listing 34). The following Section II-B will give a high level overview of the most important objects and structs used in Glibc's heap implementation.

B. Glibc Heap Overview

This section provides a high level overview of the most important objects and structs used in Glibc's heap implementation. Figure 1 shows a potential heap layout of a running process, with a focus on references between the various elements. Starting from the lowest and most important level, a chunk contains the actual user/process data, which has been allocated e.g. explicitly via a malloc call or implicitly via a new call in the context of class instantiation. Those chunks are located in a certain memory region.

Given for example the first part of a C program shown in Listing 1 which uses the Glibc, `chunk_pointer` represents a pointer to a chunk somewhere in one of these memory regions or more precisely in one of the *Allocated Chunks* areas and the `memset` call fills the chunk's complete available space with the character `A`.

This means that the `memset` call results in a long list of `A` characters residing somewhere in an area marked as *Allocated Chunks* and which belong to a certain chunk. (It should be noted that there are normally no fixed areas in which allocated or freed chunks are located, but this example has been chosen for the sake of an easy introduction.)

Besides allocated chunks that represent in essence chunks currently in use, there are also *Freed Chunks* that represent chunks which were in use but no longer are. The transition from allocated to freed is normally done via the *free* function call. When a chunk gets freed, the chunk itself, or at least its data, stays in most scenarios at the same location as before, and its data is (beside

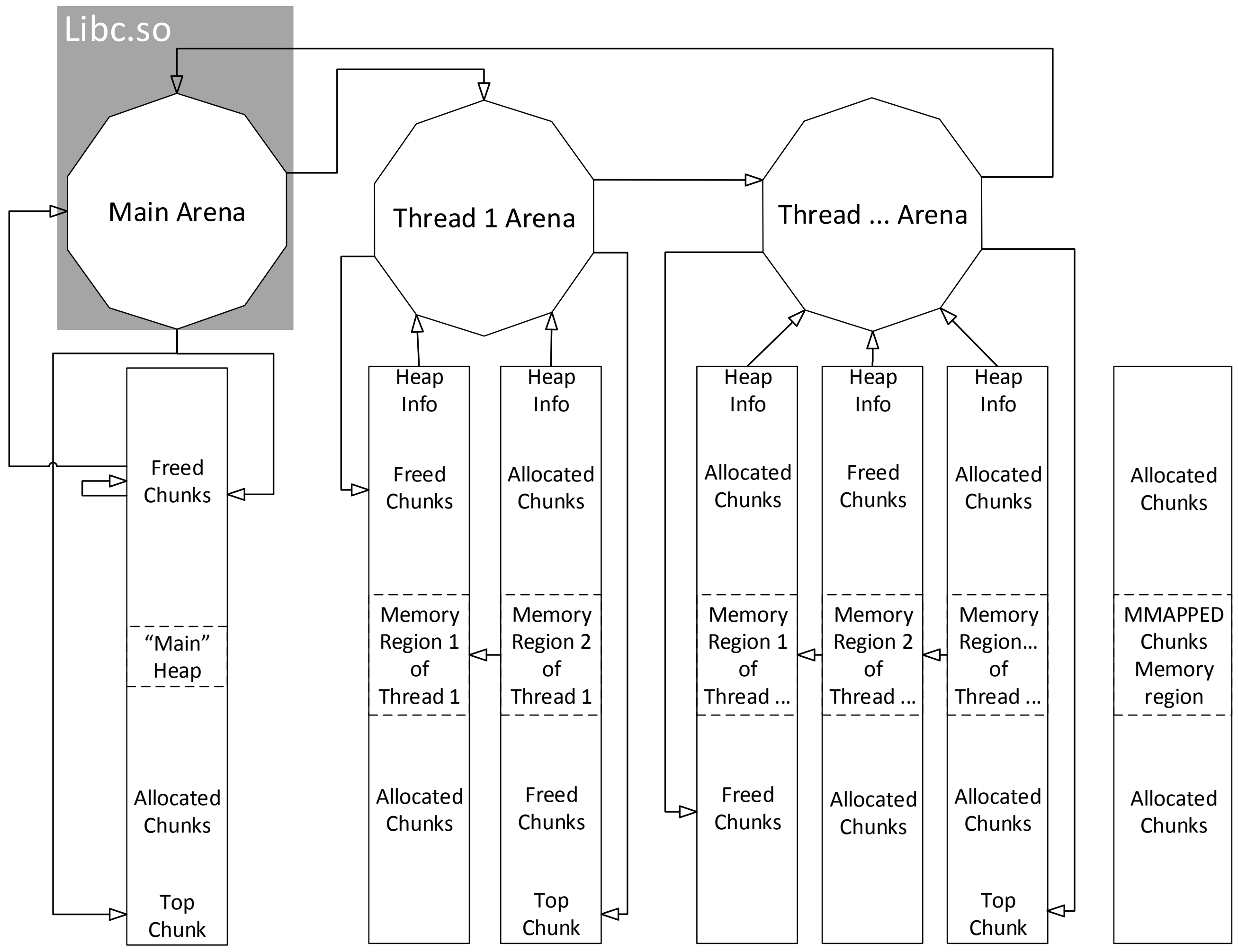

Libc.so
Main Arena
Thread 1 Arena
Thread ... Arena
Heap Info
Freed Chunks
"Main" Heap
Allocated Chunks
Top Chunk
Memory Region 1 of Thread 1
Memory Region 2 of Thread 1
Memory Region 1 of Thread ...
Memory Region 2 of Thread ...
Memory Region... of Thread ...
MMAPPED Chunks Memory region

Fig. 1: Glibc Heap Overview

```
char* chunk_pointer;
chunk_pointer = malloc(100);
memset(chunk_pointer, 65, malloc_usable_size(chunk_pointer));
```

Listing 1: Simple malloc example

some modifications which are explained later on) not deleted or overwritten. When however a new chunk is requested, whose size fits in the freed chunk, this freed chunk might get used for the new chunk and hence, the old information is overwritten.

At the highest level are arenas, which represent in essence heap space belonging to one or multiple threads while each arena has its own memory regions containing allocated and freed chunks from the associated thread(s). While an arena does not have a direct link to each memory region or to allocated chunks (see Figure 1), there are other connections like pointers to freed chunks (will be covered in Section II-C1 on the facing page), the next arena and the *top chunk*. This special chunk represents the remaining free space for a given arena. If a new chunk is requested and no freed chunk is available that could serve the request, the new chunk is created from the necessary amount of space taken from the top chunk (see Section II-C5 for more details or Section II-E for special scenarios). Besides a connection to some chunks, each arena contains also a pointer to the next arena while the last arena points to the first one located in the mapped *Glibc* library (see also Section II-D1).

One level beneath arenas are the *heap_info* structs. Despite their name they do not describe the whole heap of a process or the part of the heap associated with a thread but only a part of a mapped memory region (described by an *vm_area_struct* struct) they belong to. More specifically, each mapped memory region belonging to an arena (except for the main arena) contains, at least at the beginning of the memory region, one instance of the *heap_info* struct, which holds the size of the current heap part in that memory region. Besides the size, each *heap_info* struct holds a pointer to the associated arena (*malloc_state* struct) and a pointer to the previous *heap_info* struct within the same arena. In that way, all of them are linked together.

Excluded from arenas and *heap_info* regions are MMAPPED chunks. As can be seen in Figure 1, there are no links from MMAPPED chunks to any other structures or from heap structures to them. Those chunks are normally created when an allocation request exceeds a given threshold (typically $128 * 1024$ bytes on x86 architectures). In that case, the chunk is not included in the main heap or any memory region belonging to another arena, but the operating system is asked for an exclusive memory region just for that chunk (via the *mmap* API call), in which the chunk is placed.

While memory space for the main heap is acquired using the *brk* system call, MMAPPED chunks and also all thread arenas are using the *mmap* system call to acquire memory regions for their chunks. That means, the main heap is located in the area marked as *Heap* in Fig. 2 and MMAPPED chunks and the data from thread arenas, respectively, in the area marked as *Mmap Region* in Fig. 2.

Chunks and all other mentioned objects are described and realized via structs. For each connection which is observable in Figure 1, there is an corresponding pointer in the relevant struct to the linked struct. Those structs and connections will be examined in more detail in the next section.

C. Glibc Heap Details

The following Sections will give a low level explanations of the objects, structs and concepts described in Section II-B on page 3. The information provided in this section is based on the work of Ferguson (2007), but takes a deeper look in each topic. Further details, especially in the context of memory layout, are covered in Section II-D on page 14.

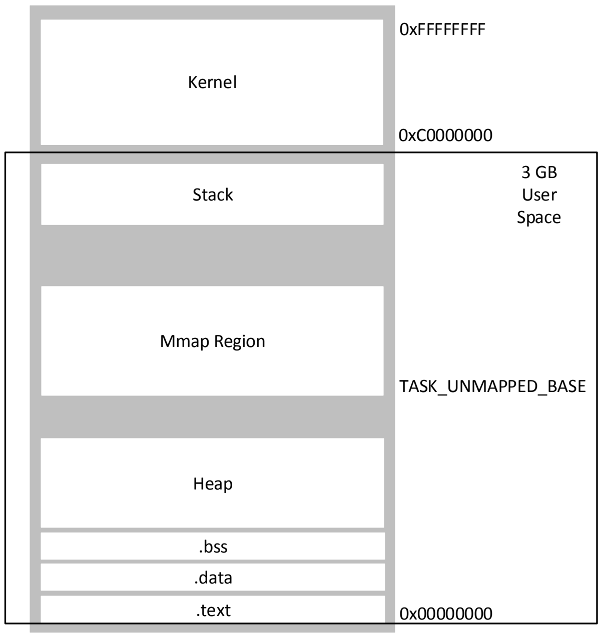


Fig. 2: Process memory layout

1) Arena and Heap Info structs: Arenas are realized via *malloc_state* structs. The struct itself and the already mentioned relations between the arena and chunks can be seen in more detail in the Fig. 3 on the next page. The *malloc_state* struct on the left side, contained in the mapped Glibc library, is called *main arena* as it is used by the first/main thread thread (see also *NON_MAIN_ARENA* flag mentioned in Section II-C2 on page 7). It holds, like all arenas, pointers to freed chunks but none to allocated chunks. The relevant members in this context are *fastbinsY* for fastbin chunks and *bins* for bin chunks. As can be seen however, the arena does not hold pointers to all freed chunks as they contain further links to subsequent freed chunks themselves. The bins and top chunk (pointed to by the *top* member) are explained in more detail in Section II-C4 on page 11.

Another relevant member of the *malloc_state* struct is *system_mem*. It holds the size of the whole arena, which means all memory regions containing any chunks belonging to that arena. In the case of the main arena, this does not include the bytes allocated by the *malloc_state* struct itself, in all other cases their size is included (see Section II-D1 on page 14 for further details). As the MMAPPED chunks are not part of any arena, their size and memory regions are hence not part of the *system_mem* value of any arena.

As described in Section II-B on page 3 and shown

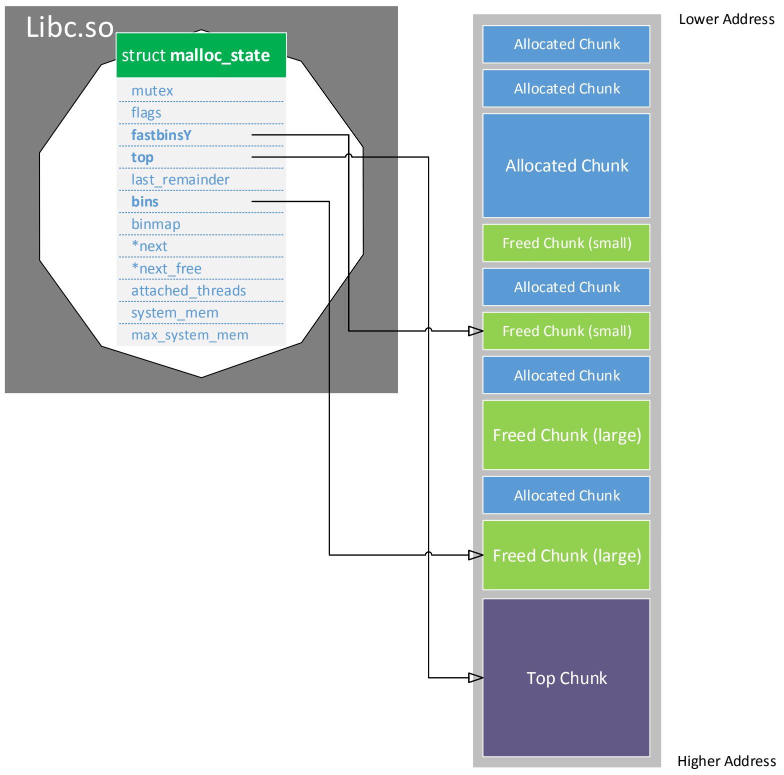


Fig. 3: malloc_state Pointers

in Fig. 1 on page 4, all arenas are linked together. This is on the one hand done via the *next* member, which realizes a circular linked list where each arena points to the next arena and the "last" arena points to the first one (main arena contained in the mapped Glibc library), and on the other hand with the *next_free* member, pointing to freed arenas (e.g. from dead threads).

The further fields of the *malloc_state* struct are not important in the context of this work and hence will be only shortly covered:

mutex This member is used during runtime to ensure exclusive access to the *malloc_state* struct during modification operations and hence prevents simultaneous access.

flags Holds flags e.g. indicating whether or not there are any fastbin chunks or any arena corruption has been detected during runtime.

last_remainder If a freed chunk is used for an allocation but is bigger than the requested size, the chunk is split and the remaining bytes form new chunk called the *last_remainder*. As this chunk is also pointed to by the unsorted bin (see Section II-C4 on page 11), this field adds no new information from a memory analysis perspective.

binmap This field holds the information which bins are empty and which do not. It is realized via 4 unsigned integers (= 128 bit), where each bit represents one bin (see also Section II-C4 on page 11). The purpose of this field are performance reasons, as traversing all bins one after the other in order to find freed chunks is more time consuming than just checking bits.

attached_threads This member was introduced in Glibc Version 2.23 and counts the number of threads that are using this arena. As there is normally a maximum number of arenas, new threads not always get their own arena but start to share them with other threads. In these cases, this counter is accordingly increased or decreased if a thread dies.

max_system_mem Is used in the context of the *system_mem* member, but serves no further valuable information for this work.

The number of arenas corresponds in principle with the number of threads. But their quantity is limited and the maximum possible amount of arenas depends on certain factors. Listing 2 illustrates the relevant code from the *malloc.c* source file, explaining those factors.

One important function in this case is `get_arena2`, which is responsible for deciding whether or not a new arena is created. The decision depends on the one hand on the existence of free arenas that can be used for the current request (line 7 and 8) and on the other hand on the amount of already existent arenas (lines beginning at 8). If there are free arenas, those are used to serve the current request for a new arena. If there are no free arenas and the current number of arenas does not exceed the value of `narenas_limit` (line 24), a new arena is created (line 26). The value of `narenas_limit` is calculated with the macro `NARENAS_FROM_NCORES` (line 28) and depends on the architecture and the number of cores (line 14). In the case of 32 bit systems the value of `narenas_limit` is the amount of cpu cores times two and for 64 bit architectures times eight.

There is however one exception, which results from the two if statements in lines 10 and 24. The value of `mp_.arena_test` is calculated with a static core number of 1. This means, the `if` statement in line 10 only returns true, if the current amount of arenas is already bigger than the result from `NARENAS_FROM_NCORES (1)`. Only in this case, `narenas_limit` is set to a value other than zero. As long as `narenas_limit` is zero, the `if` statement in line 24 evaluates to true for almost all numbers, as the operation $0 - 1$ results in the largest possible number for that data type (assuming an unsigned type). So for a system with one CPU core, `narenas_limit` is only set to a non-zero value if the number of arenas already exceeded the value of `NARENAS_FROM_NCORES (1)` and hence, the maximum number of arenas is 3 for 32 bit and 9 for 64 bit architectures. With more cores than one, `NARENAS_FROM_NCORES` dictates the maximum number.

There is however a third setting which could influence the maximum number of arenas. The function *mallopt* allows to change members of the *malloc_par* struct at runtime. This struct's members are used to control certain global settings, more specifically this is done via the only instance of that struct, called *mp_* and located in the mapped Glibc library. Amongst the members is also the field *arena_max*, which allows to set the maximum amount of arenas, independent of architecture or number of CPU cores.

While the arena describes the whole memory space related to a certain thread, the *heap_info* struct describes only a specific memory region, containing at least a subset of all chunks belonging to the associated arena. Fig. 4 on the next page shows the struct and its members.

As can be seen in Fig. 1 on page 4 or in more detail in Fig. 5 on page 16, each *heap_info* instance has a pointer to the associated arena and to the previous *heap_info* instance that also belongs to that arena. The arena pointer is stored in the *ar_ptr* member and the reference to the previous *heap_info* in the *prev* member. In contrast to an arena's *next* field however, the *prev* pointers do not realize a circular linked list. Instead, the *prev* field of the first *heap_info* is simply null.

Similar to an arena's *system_mem* member, the *heap_info* struct contains a *size* field, which defines the amount of bytes that belong to this heap region. This size normally ranges from the beginning of the memory region described by the *vm_area_struct* struct until its end. The two left members are *mprotect_size*, which is only used in the context of shrinking or growing a heap, and *pad* which simply ensures, that data following the *heap_info* struct is aligned.

2) Allocated Chunks: As explained in Section II-B on page 3, allocated chunks contain the user/process data and are located somewhere in a memory region. There are no pointers from any other heap related struct, referencing allocated chunks.

A chunk in general is described by the *malloc_chunk* struct which marks the beginning of each chunk and contains the following fields:

prev_size If the previous chunk is freed, it contains the previous chunk's size.

size The distance from the beginning of the current chunk until the next chunk in bytes.

fd Forward link, pointing to the next freed chunk.

bk Backward link, pointing to the previous freed chunk.

fd_nextsize Points to the next chunk with a bigger size.

bk_nextsize Points to the next chunk with a smaller size.

This struct is located at the beginning of each chunk but not all fields are used for every chunk. In the case of allocated chunks, primarily the *size* field is used for its intended purpose. Only if the previous chunk is a freed chunk, is the *prev_size* field used. The relevant member for this sub section is the *size* field. All other members of the *malloc_chunk* struct are only relevant for freed chunks and explained in more detail in Section II-C4 on page 11. The *size* field contains the distance from the beginning of the current chunk until the next chunk in bytes (see the appropriate subsections in Section II-D2 on page 15 for calculations regarding chunk size and data portions). As already explained by Ferguson (2007, section 0.2.1), this field not only contains size information but the lower 3 bits of the size are used as special

```
static mstate
internal_function
arena_get2 (size_t size, mstate avoid_arena)
...
  static size_t narenas_limit;

  a = get_free_list ();
  if (a == NULL)
...
          else if (narenas > mp_.arena_test)
            {
              int n = __get_nprocs ();

              if (n >= 1)
                narenas_limit = NARENAS_FROM_NCORES (n);
              else
                /* We have no information about the system. Assume two
                   cores. */
                narenas_limit = NARENAS_FROM_NCORES (2);
            }
...
      size_t n = narenas;
...
      if (__glibc_unlikely (n <= narenas_limit - 1))
...
          a = _int_new_arena (size);
...
#define NARENAS_FROM_NCORES(n) ((n) * (sizeof (long) == 4 ? 2 : 8))
...
```

Listing 2: Calculation of Arena threshold

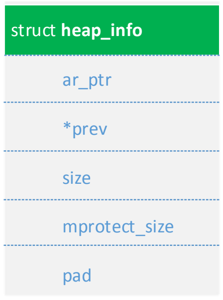


Fig. 4: heap_info struct

flags, indicating whether or not

- the previous chunk is an allocated or fastbin chunk or a freed chunk belonging to a bin. If the previous chunk is freed (but not a fastbin chunk), the least significant bit (called *PREV_INUSE*) is set.
- this chunk is an MMAPPED chunk. If it is, the second least significant bit (called *IS_MMAPPED*) is set.
- this chunk belongs to the main arena. If not, the third least significant bit (called *NON_MAIN_ARENA*) is set.

The reason that these three lower bits can be used as flags and are not required to specify the chunk's size is a size alignment. This means, that the size cannot have arbitrary values but is typically a multiple of 8 for 32 bit and a multiple of 16 for 64 bit architectures. The relevant macros which are used to get an aligned size are `request2size` and `checked_request2size` (see Listing 3). As can be seen in the macro, it not only aligns the size in the sense of ensuring it is a multiple of 8 or 16, but also always returns a higher value than given (even if the given value is already aligned). The reason for this is that it already takes into account, that the chunk size includes also struct information and hence increases the user request by that size, in order to serve the user the requested space for its data (see Section II-D2 on page 15 for more details). For example, when requesting 16 byte of data on a 32 bit architecture, this value is already perfectly aligned, but `request2size` returns 24.

Regarding those bits, the following additional statements can be made:

- An MMAPPED chunk has always the *IS_MMAPPED* bit but never the *NON_MAIN_ARENA* or *PREV_INUSE* bit set (see also Section II-C3 on the facing page).
- A freed bin (not fastbin) chunk has neither the *IS_MMAPPED* nor *NON_MAIN_ARENA* bit set. It furthermore has normally always the *PREV_INUSE*

bit set, as neighbored freed bin chunks are consolidated (see also Section II-C4 on page 11).

- A freed fastbin chunk residing in a thread arena keeps its *NON_MAIN_ARENA* bit, in contrast to a freed bin chunk (see Section II-C4 on page 11).
- A chunk following a freed bin chunk has no *PREV_INUSE* bit set, a chunk following a freed fastbin chunk however has this bit still set (see Section II-C4 on page 11).
- As stated in the comments of *malloc.c* regarding the *PREV_INUSE* bit, the "very first chunk allocated always has this bit set, preventing access to non-existent (or non-owned) memory" (see lines 1187 and 1188 in Listing 35).
- The top chunk always has the *PREV_INUSE* bit set (see Section II-C5 on page 14).

The smallest size for a chunk is defined in *malloc.c* and mainly depends on the architecture. Listing 3 shows an excerpt of the relevant lines of code from that source file.

The macro `MIN_CHUNK_SIZE` holds in fact the minimal size a chunk could have. It is defined by taking the number of bytes from the beginning of the `malloc_chunk` struct until the `fd_nextsize` member (see Sectionsec:allocatedChunks). As *prev_size* and *size* are normally unsigned integers (4 byte on 32 bit and 8 byte on 64 bit architecture) and *fd* and *bk* are pointers, the minimal chunk size on a 32 bit architecture is typically 16 and for a 64 bit architecture 32 byte. However, to ensure alignment, it is recalculated and assigned to `MINSIZE` , which is the internal relevant variable and keeps the smallest size a chunk is allowed to have. On this scenario this recalculation does not change the calculated values, so it stays with 16 and 32 byte, respectively. It should be noted that these values represent the default behavior and may differ e.g. when `INTERNAL_SIZE_T` is changed at compile time.

Line 27 of Listing 3 shows the relevant macro which enforces the minimum chunk size. On a chunk allocation, function *malloc* uses the macro `checked_request2size` and hence implicitly macro `request2size` to get the final size used for the allocation. As can be seen in lines 28 and 29 of Listing 3, if the requested size is too small, the size to allocate is set to `MINSIZE` . This is e.g. the case when *malloc* is called with a size of zero.

Regarding the maximum chunk size, it would be theoretically possible to allocate a chunk with the size 2^{32} on a 32 bit architecture as the *size* member is here normally a four byte integer. It is however restricted by the macro from line 21 in Listing 3, which is used by `checked_request2size` . The theoretically maximum amount of bytes for a chunk allocation is thus subtracted by $2*$`MINSIZE` which would typically result in $2^{32}-32$ byte on a 32 bit architecture. An allocation request must hence typically be smaller than $2^{32} - 32$ on a 32 bit architecture. When now using the maximum value that can be requested without generating an error ($2^{32}-33$) it results in the maximum possible chunk size of $2^{32} - 24$ (see line 30 of Listing 3).

Despite the maximum size defined in Glibc, a chunk with that size will most probably never be allocated. The reason for that is the limitation of the virtual address space and the fact that a large percentage of it is typically reserved for the kernel (typically about 1 GB out of 4 GB on a 32 bit architecture). But even if the kernel would not take a large percentage of the memory space, such an allocation request would ask the kernel for nearly the whole virtual address space. Even if the operating system would theoretically be able to serve that space in the sense of physical space, the virtual address space is still limited and as the process itself needs a certain amount of memory space for its code and libraries (if nothing else, at least the Glibc library) it will not be able to fulfill this request.

3) MMAPPED Chunks: MMAPPED chunks are in essence allocated chunks, that exceed a given threshold in size. Their name originates from the fact, that a dedicated memory region is requested for each chunk from the operating system using the API call *mmap*. Most probably for that exact reason, they are called *MMAPPED regions* in the *mallinfo* output (see Section IV-A on page 35). The threshold, which defines if the new allocation request should be served with a *mmap* call, is controlled with the *malloc_par* field *mmap_threshold*, which is in the beginning normally $128 * 1024$ and can be set manually by changing the *mmap_threshold* value during runtime (*mp_*; see also Section II-C1 on page 5). This is however not the only way this value might change. As long as the *malloc_par* member *no_dyn_threshold* is 0, the threshold might change dynamically during runtime. (It is for example set to 1, if the *mmap_threshold* is set manually using the *mallopt* function.) If not, the threshold is adjusted each time an MMAPPED chunk is freed, as can be seen in Listing 4. The threshold does however only increase (line 2956), and not exceed a given threshold (line 2957). That means, that there might be MMAPPED chunks with a size smaller than the current value of `mp_.mmap_threshold` . The maxi-

```
...
/* The corresponding word size */
#define SIZE_SZ          (sizeof(INTERNAL_SIZE_T))
...

#define MALLOC_ALIGNMENT (2 *SIZE_SZ)
...

/* The corresponding bit mask value */
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
...

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE (offsetof(struct malloc_chunk, fd_nextsize))

/* The smallest size we can malloc is an aligned minimal chunk */

#define MINSIZE \
  (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK))
...
#define REQUEST_OUT_OF_RANGE(req)                  \
  ((unsigned long) (req) >=                                  \
   (unsigned long) (INTERNAL_SIZE_T) (-2 * MINSIZE))

/* pad request bytes into a usable size -- internal version */

#define request2size(req)                          \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE) ? \
   MINSIZE :                                       \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)

/* Same, except also perform argument check */

#define checked_request2size(req, sz)              \
  if (REQUEST_OUT_OF_RANGE (req)) {                          \
      __set_errno (ENOMEM);                                  \
      return 0;                                              \
    }                                                        \
  (sz) = request2size (req);
```

Listing 3: Chunk size calculation

mum value for the dynamic threshold is defined by the `DEFAULT_MMAP_THRESHOLD_MAX` variable, which is typically 524288 for 32 bit and 33554432 for 64 bit architectures.

The `mp_.mmap_threshold` value can also be set manually via the *mallopt* function. The maximum value it can be set to is typically the same as for the dynamic threshold (see Section VII-B on page 49), but it can be set to an arbitrary low value like 1. When doing that, this means that all allocations are served with MMAPPED chunks, but does not lead automatically to chunks with a size of 1 or `MINSIZE` (see Listing 3). The reason for that is the *mmap* API call, which returns memory space in terms of pages. As it returns only a multiple of page size, the minimum amount of memory served by this function is one page (which is at least 4096 byte). Because the whole page is reserved for that one chunk (a follow up *malloc* call would again end in a *mmap* call, requesting more pages for the new chunk), there is no reason to not assign the whole space to this one chunk, which is why the size of an MMAPPED chunk is always a multiple of page size and at least as large as one page of the underlying operating system (see Section II-D3 on page 17 for further details). This can also be seen in the code excerpts in Listings 5 and 6. Lines 2316 and 2318 in Listing 5 are responsible for setting the new MMAPPED chunks's size (`nb` is the size requested by the *malloc* call) and do that by the usage of the `ALIGN_UP` macro defined in line 67 in Listing 6.

Another characteristic of MMAPPED chunks is their usage of flags. While all MMAPPED chunks have the *IS_MMAPPED* flag set, none of them use the *PREV_INUSE* or *NON_MAIN_ARENA* flags. Furthermore, there are no freed MMAPPED chunks, which is why there is no need for the *PREV_INUSE* flag and hence leaves the *prev_size* member and all freed chunk pointers unused. When an MMAPPED chunk is freed, the whole memory space it is allocating is removed from

```c
    if (!mp_.no_dyn_threshold
        && p->size > mp_.mmap_threshold
        && p->size <= DEFAULT_MMAP_THRESHOLD_MAX)
    {
        mp_.mmap_threshold = chunksize (p);
```

Listing 4: Glibc 2.23(malloc/malloc.c): Dynamic MMAP Threshold adjustment

```c
    if (MALLOC_ALIGNMENT == 2 * SIZE_SZ)
        size = ALIGN_UP (nb + SIZE_SZ, pagesize);
    else
        size = ALIGN_UP (nb + SIZE_SZ + MALLOC_ALIGN_MASK, pagesize);
```

Listing 5: Glibc 2.23(malloc/malloc.c): Calculation of minimum MMAPPED Chunks size

```c
/* Align a value by rounding down to closest size.
e.g. Using size of 4096, we get this behavior:
{4095, 4096, 4097} = {0, 4096, 4096}. */
#define ALIGN_DOWN(base, size) ((base) & -((__typeof__ (base)) (size)))

/* Align a value by rounding up to closest size.
e.g. Using size of 4096, we get this behavior:
{4095, 4096, 4097} = {4096, 4096, 8192}.

Note: The size argument has side effects (expanded multiple times). */
#define ALIGN_UP(base, size) ALIGN_DOWN ((base) + (size) - 1, (size))
```

Listing 6: Glibc 2.23(include/libc-internal.h): Definition of ALIGN_DOWN and ALIGN_UP

the process space and returned to the operating system. Because there are no pointers from the bins or any structs to MMAPPED chunks and also no connection between themselves, each MMAPPED chunk stands alone and is only referenced by the returned pointer for the allocation call.

4) Freed Chunks and Bins: When talking about freed chunks in more detail, first the concept of bins needs to be understood. A bin can be seen as a container for freed chunks and bins always belong to a specific arena. To put it a bit more technically, they are in essence an array of pointers residing in the *malloc_state* struct (arena). If a chunk is freed within a certain arena, it gets added to a bin of that arena by setting the bins pointer to that chunk. Furthermore, it gets added only to a bin it fits in, as almost all bins allow only a defined size or size range. There are two types of bins: fastbins and emphnormal bins (which are referenced in this document simply as bin). The second type is again split up in two types: small bins and large bins.

Based on that information and going a step further, there are basically four different kinds of free chunks to distinguish:

Small Bin chunks Are freed chunks in a size range from 16 to 508 bytes on a 32 bit and 32 to 1008 byte on a 64 bit architecture belonging to a bin.

Large Bin chunks Are freed chunks in a size range from 512 byte on 32 bit and from 1024 byte on 64 bit architectures until the maximum size a chunk can have (belong also to a bin).

Fastbin chunks Are freed chunks in a size range from 16 to a maximum of 80 byte on 32 bit and from 32 to a maximum of 160 byte on 64 bit architectures (they belong to a fastbin).

Top chunks This chunk is not part of any bin, present exactly once in each arena and represents the left free space of its arena. See Section II-C5 on page 14 for more details.

A differentiation on those chunk types is especially important when trying to gather user data out of them (see subsections in Section II-D on page 14).

The first bin is neither part of the small nor large bins but contains freed chunks of arbitrary size without any size order. It is used for performance reasons as integrating a chunk in the according bin in the right position takes more operations than adding it to the unordered bin. Bins containing chunks with varying sizes are normally ordered by size, so for a new chunk the right position must be found and furthermore not only the new chunks pointers set, but also the pointers of the chunk before and after it. So if a new allocation is following a *free* call and a fitting chunk can be found in the arbitrary bin, some bin modifications and comparisons have been saved. Every chunk that gets freed (except for chunks being placed in fastbins) is at first placed in the unordered bin. The *free* function itself does not place any chunks into small or large bins. This is done by the *malloc* function on its next call. While testing each chunk in the unordered list for fitting the current allocation request, all other chunks are placed in the corresponding bin.

As mentioned earlier, bins are split up into small and large bins. More precisely there are 62 small and 63 large bins, where the small bins contain only chunks of the same size and the large bins contain chunks with a size in a specific range. The exact distribution can be seen in the comment in Listing 7 which is taken from the Glibc source code. This output is however not entirely correct. On the one hand, there is a misleading value regarding number of small bins. The comment speaks of 64 bins, each 8 bytes apart from its neighbor, but there are actually only 62 as bin 0 does not exist (this is also stated in the comment on line 1466, but only a mathematical issue as the first bin in the *malloc_state*'s *bins* array is definitively used) and the first bin is used for the arbitrary sized chunks. The second deviation concerns each first bin of a given size range starting with bin 112 and will be explained in more detail in the following paragraphs.

Looking at the small bins, they begin with bin 2 (when counting from one forward) and a size of 16 and go up to bin 63 with a size of 504, while each bin is 8 bytes apart from its neighbors and contains only chunks of the same size. It should be noted that the information from Listing 7 and in this and following paragraphs only apply for 32 bit architectures. The number of bins does not change however for other architectures, only the size of chunks placed in there are increased (for example are the small bins 16 bytes apart on 64 bit architectures).

Starting with bin 64, chunks can have sizes in a given range, which is noted in the last column in lines 1453 to 1457 in Listing 7. So the first large bin can contain freed chunks with sizes from 512 - 568 byes. All following bins until bin 96 each contain chunks of the same size range and are 64 bytes apart from their neighbors. This pattern continues according to the remarks until bin 112. While bins 112 till 119 should normally all contain chunks in a size range of 4096 byte, the first bin (bin 112) does only include chunks with a size range of 1536 byte (bins 113 till 119 behave as expected). This deviating

```
1449    Bins for sizes < 512 bytes contain chunks of
            all the same size, spaced
1450    8 bytes apart. Larger bins are approximately
            logarithmically spaced:
1451
1452    64 bins of size  8
1453    32 bins of size 64
1454    16 bins of size 512
1455     8 bins of size 4096
1456     4 bins of size 32768
1457     2 bins of size 262144
1458     1 bin of size what's left
1459
1460    There is actually a little bit of slop in the
            numbers in bin_index
1461    for the sake of speed. This makes no
            difference elsewhere.
1462
1463    The bins top out around 1MB because we expect
            to service large
1464    requests via mmap.
1465
1466    Bin 0 does not exist. Bin 1 is the unordered
            list; if that would be
1467    a valid chunk size the small bins are bumped
            up one.
```

Listing 7: Glibc 2.23(malloc/malloc.c): Comment explaining bin sizes

pattern continues for each first bin of each new size range:

Size range 32768 First bin (bin 120) only includes chunks with a size range of 24576 byte.

Size range 262144 First bin (bin 125) only includes chunks with a size range of 98304 byte.

Besides those deviations, all other bins behave as expected and the last bin (bin 126) contains an ordered list of sizes not fitting in any of the other bins. A complete list with all bins and their size ranges can be seen in Section VII-G on page 53. This Section also shows the bin distribution when *MALLOC_ALIGNMENT* is set to an architecture atypical value (see also Section II-D2 on page 15) and for a 64 bit architecture, respectively. In both cases, the distribution differs from the one explained in this section and displayed in Listing 7, respectively. The program in Section VII-G on page 53 uses the *bin_index* macros, that are also used internally by the Glibc to decide which chunk is placed in which bin.

Fastbins are similar to small bins in the sense that they contain only chunks of the same size and are all 8 bytes (16 bytes on 64 bit architectures) apart. And like with the unordered list, the reason for the fastbins itself is their performance improvement while using them as fewer operations regarding e.g. pointer operations have to be made. However, there are only ten fastbins which is defined by the `NFASTBINS` (see Listing 8). Furthermore, only nine out of those ten fastbins are actually used for chunks. This can be seen for example in line 1608 of Listing 8 where `NFASTBINS` is calculated (`SIZE_SZ` has typically a value of 4 for 32 bit and 8 for 64 bit architectures). The macro `fastbin_index` returns the index into the fastbins array for a given size. In this case, it is given the value of `MAX_FAST_SIZE`, which is Glibc's internal maximum size for a fastbin chunk. So the index for the fastbin containing the biggest possible fastbin chunk is returned and assigned to `NFASTBINS` while adding one up to it. The last fastbin is hence normally never used and maybe only created for alignment reasons.

The maximum size for a fastbin chunk is returned by the macro `get_max_fast` shown in Listing 9 and for example used in the *_int_malloc* function when deciding whether or not an already existing fastbin chunk should be used. The macro returns the value of `global_max_fast`, which is calculated with the macro `set_max_fast` (line 1676 of Listing 9).

`set_max_fast` is typically called with the `DEFAULT_MXFAST`, which is 64 byte for 32 bit and 128 byte for 64 bit architectures. So normally, only seven out of 10 fastbins are used while fastbin chunks have a size from 16 up to 64 byte. It can however also be called manually via the function *mallopt* and when supplying the maximum possible value, nine out of ten fastbins are used. The relevant part of function *mallopt* is shown in Listing 10 (`value` is the given size for the new maximum fastbin chunk size).

When freeing chunks, there is also a difference between fastbin and bin chunks. In this scenario comes the *PREV_INUSE* bit and the *prev_size* field of the *malloc_chunk* struct from Section II-C2 on page 7 into play. If the current chunk that should be freed has a direct neighbor which is already a freed bin chunk or the top chunk, and the current chunk would not end up as a fastbin chunk, both are getting consolidated to one chunk. If the current chunk or the neighbors are fastbin chunks, no consolidation happens. The distinction is made upon the *PREV_INUSE* bit. If this bit is not set, the previous chunk is considered free and ready for consolidation. A chunk following a fastbin chunks keeps its *PREV_INUSE* bit set and hence they are not consolidated on free and the next chunk's *prev_size* field does not contain the fastbin chunk's size.

If the current chunk is too big for any fastbin and the previous chunk is either a small or large bin chunk, they will be consolidated on a call to *free*. As there are no pointers to allocated chunks (which the current chunk that should be freed at this moment still is) and allocated chunks themselves keep no pointers to previous or next chunks, the task now is to figure out the beginning of the previous chunk (as this chunk gets the new freed chunk with an increased size). This is accomplished by the *prev_size* field which keeps the size of the previous chunk if it is a freed bin chunk (this is not the case for fastbin chunks). So by using the size of the previous chunk and the offset of the current chunk, the offset of the previous one can simply be calculated via subtraction. The last part now is to adjust the previous chunk's size to reach until the end of the current chunk (the current chunk is assimilated) and to integrate the current chunk in the appropriate bin (for details see the following part of this section and Section II-D4 on page 18).

The second scenario is that the next chunk might be available for consolidation. In this case, there are two scenarios to distinguish:

1) The next chunk is the top chunk. The current chunk gets consolidated with it.
2) If not, the *PREV_INUSE* bit of the chunk after next (it is retrieved by simply adding up the chunks sizes) is examined, whether or not the next chunk (from the current chunk's point of view) is in use. If not, the last step is similar the scenario with the previous freed chunk, except that the next chunk is first released from its bin and afterwards assimilated (the current chunk's size is adjusted and added to the appropriate bin).

The last parts left from the *malloc_chunk* struct from Section II-C2 on page 7 are the members behind the *size* field. They all are pointers and used in conjunction with freed chunks. The first field *fd* is a forward pointer to the next free chunk of the same bin, which must not necessarily be located on a following address in the sense of memory space but can also be located behind that chunk. The counterpart is the field *bk*, which is a backward pointer and hence points the previous freed chunk of the same bin. In the case of small and large bin chunks, both pointers are used, realizing a doubly linked list. Those lists are also circular, which means that following e.g. only the *fd* pointer will iterate over all freed chunks of that bin over and over again.

There are two differences to fastbin chunks at this point. Fastbin chunks use only the forward link *fd* but not the backward link (or any other pointer) hence realizing a simple linked list (it does not loop). The *fd* field of the

```
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)


/* The maximum fastbin request size we support */
#define MAX_FAST_SIZE (80 * SIZE_SZ / 4)

#define NFASTBINS (fastbin_index (request2size (MAX_FAST_SIZE)) + 1)
```

Listing 8: Glibc 2.23(malloc/malloc.c): Number of fastbins

```
#define set_max_fast(s) \
  global_max_fast = (((s) == 0)                                 \
                     ? SMALLBIN_WIDTH : ((s + SIZE_SZ) & ~MALLOC_ALIGN_MASK))
#define get_max_fast() global_max_fast
```

Listing 9: Glibc 2.23(malloc/malloc.c): Maximum fastbin chunk size

```
      if (value >= 0 && value <= MAX_FAST_SIZE)
        {
          LIBC_PROBE (memory_mallopt_mxfast, 2, value, get_max_fast ());
          set_max_fast (value);
```

Listing 10: Glibc 2.23(malloc/malloc.c): Setting maximum fastbin chunk size via mallopt

last freed chunk of a fastbin is hence set to null.

In the case of large bins, also the members *fd_nextsize* and *bk_nextsize* come into play. As already mentioned, large bins contain chunks of a given size range. So, those fields are used to point to the next/previous chunk with a different size. They are mainly used as a performance improvement as it speeds up traversing long lists of freed chunks while searching the right position for a new freed chunk for that bin (large bins are ordered by size in descending order: the largest chunks for that bin are the first chunks). While not all large bin chunks have those pointers set but only the first chunk of each size, those fields are still overwritten with null for all other large bin chunks. See also Section II-D4 on page 18 for further details.

5) Top Chunk: Each arena has one top chunk and holds a pointer to it. It represents the free space left in that arena and is used for allocation requests where the bins are not able to offer an appropriate free chunk. On such an allocation, the portion necessary to fulfill the request is used as the new allocated chunk and the rest becomes the new top chunk. If the top chunk does not offer enough space for the current allocation request, additional space is gathered either via a *brk* or *mmap* API call. For the top chunk, neither the *NON_MAIN_ARENA* nor the *IS_MMAPPED* flag are set but always the *PREV_INUSE*. While at first this might not be obvious for the *NON_MAIN_ARENA* flag, it definitely makes sense for the other two. In the case of *IS_MMAPPED* flag (for more details see Section II-C3 on page 9), the reason is obvious: there are no top chunks inside MMAPPED regions. Regarding the *PREV_INUSE* flag, there are two scenarios to consider:

1) The top chunk is the first chunk in the current memory region, which results in the same situation explained earlier in Section II-C5 regarding the *very first chunk*.
2) Freeing chunks residing directly before the top chunk are either getting fastbin chunks (in this case the *PREV_INUSE* is not unset; see also Section II-C4 on page 11) or consolidated with the top chunk, leaving the beginning of the memory region or a fastbin/allocated chunk before it, in which cases the *PREV_INUSE* again stays set.

D. The Memory View

The following sections describe how and where the structs, described in Section II-C on page 5, are stored in memory for a running Linux user space process that uses Glibc for heap allocations.

1) Arena and Heap Info structs in Memory: The main heap is a continuous region of memory containing all chunks of the main arena. While it is continuous, it can however get split up in multiple contiguous memory regions described by *vm_area_struct* structs. Its describing *malloc_state* struct is stored in the *bss* section of the mapped Glibc library and as already stated, no *heap_info* structs are used for the main arena.

The *malloc_state* struct for thread arenas however is stored together with chunks in the same memory region. More precisely, it is located right after the first *heap_info* struct and before the first chunk of that arena. Normally, a *heap_info* stuct can be found at the beginning of each mapped memory region belonging to a thread arena. Besides that, there are also instances where further *heap_info* stucts end up in the same mapped memory region. These can be related to the same arena but also to another one.

It can however be stated that at the beginning of a memory region belonging to a thread arena, normally always a *heap_info* struct can be found.

Fig. 5 on the following page illustrates the previously described scenarios and the information from Section II-C1 on page 5. Grey areas are memory regions described by *vm_area_struct* structs, structs and chunks marked in blue belong to the same arena and the ones in green to another. The pointers starting on the *size* member mark the end of the memory area described by the corresponding *heap_info* struct, *ar_ptr* points to its arena (*malloc_state* struct) and *prev* to the previous *heap_info* struct in the same arena.

Heap and stack are normally at opposite sides and grow towards each other. This is, in fact, true as long as the heap does only consist of the main arena without any thread arenas or MMAPPED chunks. But as soon as either one of them is introduced, this strict separation is broken. The MMAPPED chunks scenario is explained in Section II-C3, Section II-D3 and Section III-F. Similar to MMAPPED chunks, the memory regions belonging to thread arenas most of the time get mixed up with memory regions containing stack frames for certain threads.

Regarding the value of an arena's *system_mem* and a *heap_info*'s *size* member, respectively, those sizes must not necessarily correspond with the size of the associated memory regions. There have been instances observed, where slack space was at the end of an arena. More precisely, it was slack space right after the top chunk. In these cases, the top chunk did not consume the whole space until the end from the containing memory region but left some slack space.

2) Allocated Chunks in Memory: As already described in Section II-C2 on page 7, the size of chunks is aligned. Alignment in this context does however not only apply to the size but also to the location of chunks. The address where each chunk starts is not arbitrary but controlled with the variables *MALLOC_ALIGNMENT* and *MALLOC_ALIGN_MASK*, while *MALLOC_ALIGNMENT* has typically a value of 8 for 32 bit and 16 for 64 bit architectures and *MALLOC_ALIGN_MASK* is for both architectures `MALLOC_ALIGNMENT - 1` (see Listing 3 for their definitions). Glibc macros that test for a chunk's alignment and use these variables are `aligned_OK` and `misaligned_chunk`, which are shown in Listing 11.

This means that chunks are typically located on an address that is evenly divisible by 8 on a 32 bit and by 16 on a 64 bit architecture, respectively. It should however be noted that it is possible to set *MALLOC_ALIGNMENT* to an architecture untypical value via compile-time options resulting in different address and size alignments.

Another fact to consider regarding the first chunk of an arena respectively a memory region described by a *heap_info* struct: This chunk might, in some cases, not be located right after a *malloc_state* or *heap_info* struct, but a few bytes after them. The reason for that is on the one hand the alignment requirement for the chunk's starting address, as explained previously, and on the other hand the *malloc_state*'s and *heap_info*'s struct size, respectively. Because their size must not be evenly divisible by 8 and 16, respectively, there might be cases in which the next free space after those structs is at an address which does not satisfy the alignment requirement. This is for example the case for Glibc version 2.23 on a 32 bit architecture, as the *malloc_state* struct has a total size of 1108, which is not evenly divisible by 8. So for the first memory region of the thread arena, which contains a *heap_info* struct (total size of 16) followed by a *malloc_state* struct, the next free address would be 1124, but the next aligned address is 1128 which is where the first chunk will end up.

As previous Glibc versions like 2.22 and 2.21 did not have the *attached_threads* member for the *malloc_state* struct, which has a size of 4 byte on 32 bit architectures, the first chunk is stored in these cases right after *malloc_state* struct. This is for example conversely true for 64 bit architectures. Here did the *attached_threads* member make the *malloc_state* struct evenly divisible by 16, which was not the case on versions before Glibc 2.23.

The additional offset scenario is however not relevant for the main arena, because neither a *malloc_state* nor a *heap_info* struct are placed in the main heap memory region and the start address of the main heap is typically evenly divisible by 8 and 16.

Fig. 6 on page 17 illustrates an allocated chunk in

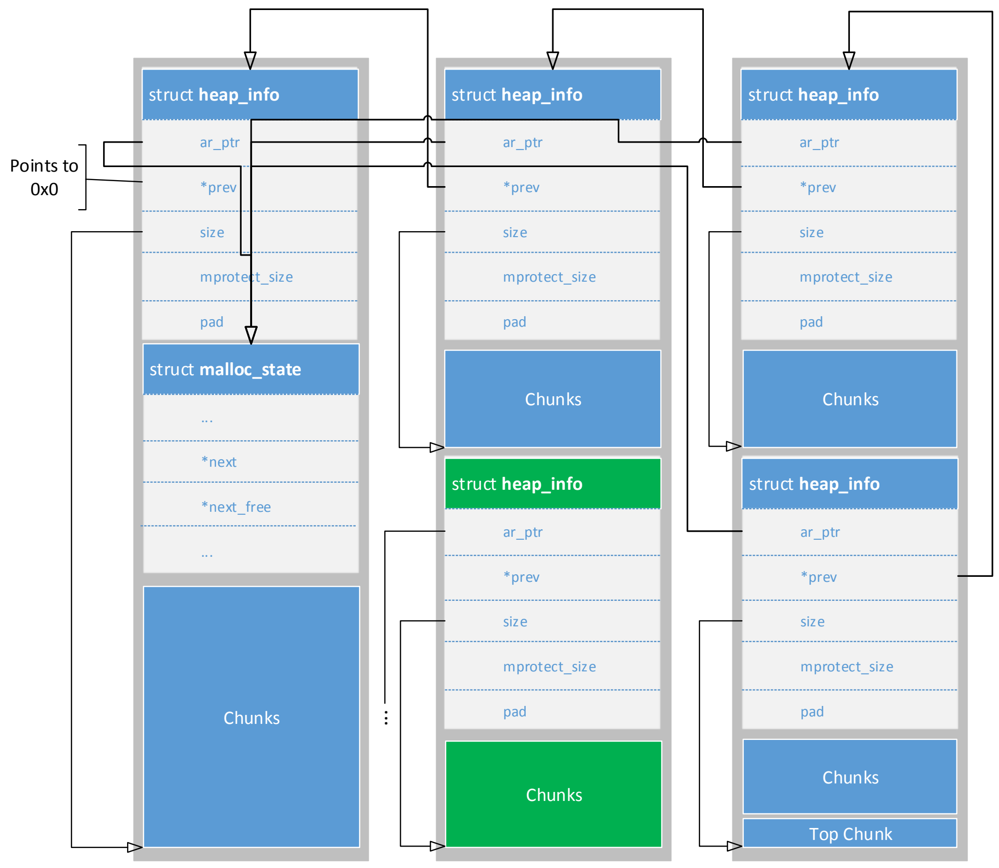


Fig. 5: heap_info and malloc_state structs in memory

```
/* Check if m has acceptable alignment */

#define aligned_OK(m)  (((unsigned long)(m) & MALLOC_ALIGN_MASK) == 0)

#define misaligned_chunk(p) \
  ((uintptr_t)(MALLOC_ALIGNMENT == 2 * SIZE_SZ ? (p) : chunk2mem (p)) \
   & MALLOC_ALIGN_MASK)
```

Listing 11: Glibc 2.23(malloc/malloc.c): Macros for alignment test

memory. As can be seen, the user data starts right after the *size* member and reaches until the size member of the next chunk. So all members of the *malloc_chunk* struct following the *size* field are overwritten with user data. This goes a step further and should be explained in an example.

When allocating a chunk e.g. on a 32 bit architecture with *malloc(500)*, a chunk with the value 504 (disregarding any flags) in its *size* member is created. But when calling the Glibc function *malloc_usable_size*, which returns the amount of bytes the given chunk can store, it will return 500. There is a second fact to take into account. As stated before, the data part begins right after the *size* member, while the *size* member itself defines the size of the whole chunk, starting with the current chunk's *prev_size* member until the *prev_size* member of the next chunk. On a 32 bit architecture, this would normally leave only 496 byte for user data. The reason that the usable size is still 500 byte, shall be explained using the Fig. 6 on the facing page.

As can be seen, user data of an allocated chunk (in this case the chunk at the top) reaches from the *fd* member until the beginning of the *size* field of the next chunk while overwriting all blue fields with user data. As ex-

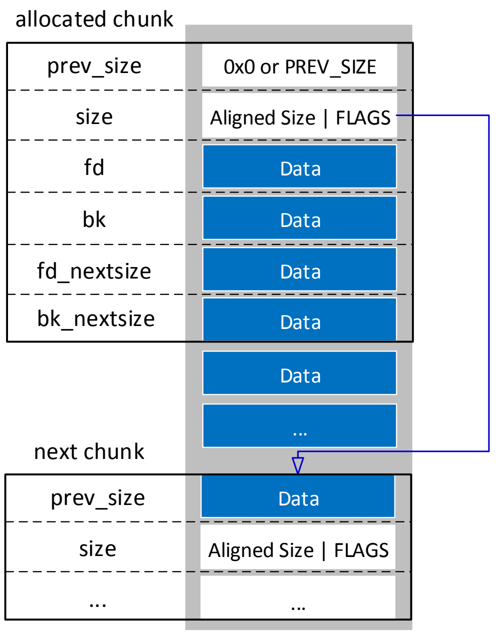


Fig. 6: Allocated chunk in memory

plained in Section II-C4 on page 11, the least significant bit of the *size* field is the *PREV_INUSE* bit, which determines whether or not the previous chunk is allocated or free and hence the *prev_size* field contains the previous chunk's size. If the next chunk's *PREV_INUSE* bit is set, which is the case for chunks following an allocated chunk, its *prev_size* member does not contain the size of the previous chunk but are simply part of the previous chunk's user data. So for all allocated chunks within the main or any thread arena, the *prev_size* field of the next chunk is used to store data. This technique of using fields of a struct only in cases they are required and otherwise leaving them for other purposes is called *boundary tags* (see e.g. the survey of such techniques by Wilson et al. (1995, p. 28)).

From an external point of view, the amount of usable bytes can be calculated with one of the following formulas, where `sizeof(FIELD)` represents the size of the given field and not the value of that field (e.g. on a 32 bit architecture, the `size` field is typically an four byte unsigned integer: `sizeof(size)=4`) and `chunksize` is the value of the `size` member without any flags:

$$\begin{aligned}\texttt{usable\_bytes} &= \texttt{chunksize} - (\texttt{sizeof(prev\_size)} + \\ &\quad\ \texttt{sizeof(size)} + \texttt{sizeof(prev\_size)}\\ \texttt{usable\_bytes} &= \texttt{chunksize} - \texttt{sizeof(prev\_size)}\end{aligned}$$

3) MMAPPED chunks in Memory: MMAPPED chunks are normally located in a dedicated memory region described by a *vm_area_struct* struct (see Section III-F on page 30 for exceptions), which contains one or more of such chunks. As described in Section II-B on page 3, there are no pointers from meta structures like *malloc_state* or *heap_info* that reference the memory regions or the chunks themselves. While those regions are often located near other heap related memory regions, they can also be stored somewhere else in the process address space (e.g. between mapped files).

Similar to their size, which is always a multiple of page size (see Section II-C3 on page 9), they are always located at an address that is evenly divisible by page size. This is due to the *mmap* API function that returns the memory space for the requested chunk, which not only returns a size but also an address on a page size boundary. Moreover, despite the fact that each MMAPPED chunk results from a separate *mmap* call, multiple MMAPPED chunks can end up in the same memory region described by one *vm_area_struct* struct, as the kernel can simply enlarge a region. On the other hand, a continuous memory region can get split up in two separate regions if an MMAPPED chunk, located between two or more MMAPPED chunks from the same region, is freed (its related pages are returned to the operating system).

As described in Section II-C3 on page 9, the requested size for a chunk (when exceeding the mmap threshold) is increased to the next highest value evenly divisible by page size with the macro *ALIGN_UP*, as *mmap* returns anyways a memory region that is a multiple of page size. Despite that fact, there are still scenarios in which an MMAPPED chunk does not use the whole memory region returned by the *mmap* call. The memory region returned by *mmap* in such cases was at least one page size larger than the size requested by the Glibc implementation (after aligning it up). This leaves some slack space (similar to the scenario described in Section II-D1 on page 14) right after the last MMAPPED chunk for a given memory region that typically consists only of null bytes.

The calculations for the usable size of an allocated chunk done in Section II-D2 on page 15 are not valid for MMAPPED chunks as they do not use the *prev_size* member of the next chunk. This is also illustrated in Fig. 7 on the following page. There are two reasons for that:

- It is not guaranteed that an MMAPPED chunk is followed by another chunk, whose *prev_size* could

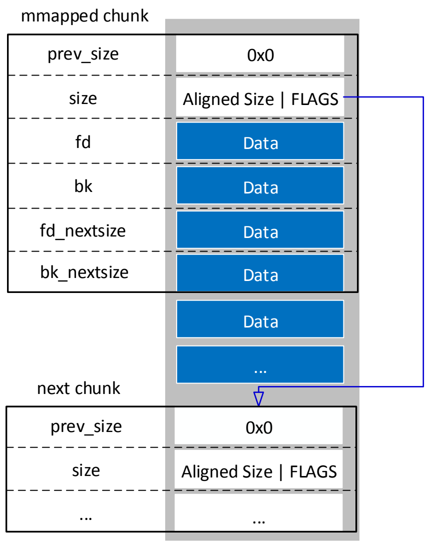


Fig. 7: Allocated MMAPPED chunk in memory

be used for data (e.g. in the case of only one MMAPPED chunk, or for the MMAPPED chunk at the end of a memory region).

- Even if the current MMAPPED chunk has a following chunk whose *prev_size* could be used, as soon as the following chunk is freed, its memory space is removed from the current process and would hence lead to missing data.

The usable size for an MMAPPED chunk can be calculated with the following formula:

$$\texttt{usable\_size} = \texttt{chunksize} - 2 * \texttt{sizeof}(\texttt{prev\_size})$$

4) Freed Chunks and Bins in Memory: At the beginning of an arena, all bins and fastbins are empty. This is however realized in different ways. In the case of fastbins, their pointers are all initialized with zero, while bins have a pointer to themselves. How this referencing works shall be explained using the following figures. The first Fig. 8 shows three bins and the initial situation for bin 11, whose pointers are still referencing itself. However, when looking solely at those pointers, it seems like they reference the previous bin. The reason for this is that they are pointers to a *malloc_chunk* struct. As each bin contains a forward and a backward link, the initial pointers do not really point to a previous bin or to themselves, but to the position where a chunk would be stored, if bin 11's *fd* and *bk* pointers are those of a *malloc_chunk*. The template of an imaginary chunk (drawn in black) in Fig. 8 tries to visualize this scenario.

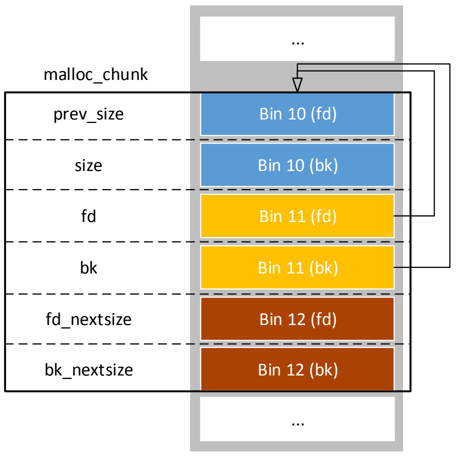


Fig. 8: Initial bin situation

Taking this a step further, Fig. 9 on the next page shows the situation in which bin 11 contains exactly one chunk. As can be seen, the real freed chunk (on the right side) references the imaginary chunk in the arena, so that this chunk's *fd* and *bk* pointers match with bin 11's pointers.

The last general modification is done when multiple chunks are part of a bin. This is illustrated in Fig. 10 on the facing page. As can be seen, the bin's backward pointer points to the last chunk of that bin (which is the chunk with the smallest size) and the forward pointer references the first chunk (the biggest one). On the other hand, the first chunk's *bk* and the last chunk's *fd* pointer both reference the imaginary chunk in the arena and hence realize a circular doubly linked list.

The scenario with fastbins is a bit different. As already noted, initially all fastbin pointer are set to zero and as described in Section II-C4 on page 11, fastbin chunks only use the *fd* pointer and are not linked circularly. If a chunk is included in a fastbin, a simple *malloc_chunk* pointer is created and beginning with the first fastbin chunk, the *fd* field is used to point to the next chunk but none of them points back. Hence, there is no scenario in which a reference points back to another fastbin in order to realize an imaginary chunk whose *fd* field would end up at the correct offset for the current fastbin. If all chunks of a fastbin are reallocated, the arena's fastbin pointer is again set to zero.

Continuing the usable amount of space calculations for allocated chunks described in Section II-D2 on page 15 and Section II-D3 on the previous page, the four different kinds of free chunks shall be examined

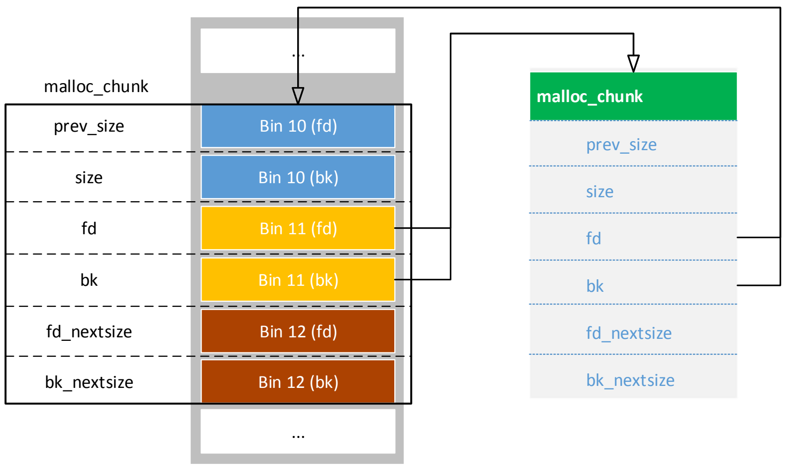


Fig. 9: Bin with one chunk

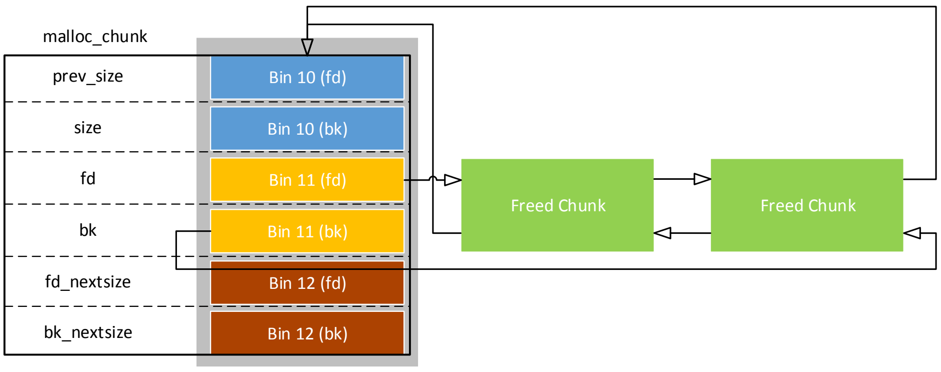


Fig. 10: Bin with multiple chunks

for their amount of potentially not overwritten amount of data. In the case of a fastbin chunk, the only part that differs is the increased offset for the beginning of remaining user data. As can be seen in Fig. 11 on the following page, the fastbin chunk uses the *fd* member and hence overwrites any potential data at that position. Parts marked blue are data from that fastbin chunk, parts marked in purple are data from other chunks. The area for non-overwritten data starts with the *bk* field and reaches until the *size* member of the next chunk, like with an allocated chunk, because the *PREV_INUSE* flag is not unset for fastbin chunks and hence the *prev_size* field of the next chunk is not overwritten with size information (see also Section II-C4 on page 11).

Using the last calculation example from Section II-C2 on page 7, the following formula can be derived for the amount of not overwritten user data:

$$\begin{aligned}\texttt{not\_overwritten\_bytes} &= \texttt{chunksize} - \texttt{sizeof(prev\_size)}\\ &\quad - \texttt{sizeof(fd)}\end{aligned}$$

Regarding the case of a small bin chunk, there are slightly more differences. As can be seen in Fig. 12 on the next page and explained in Section II-C4 on page 11, all freed bin chunks use the *fd* and *bk* members for the doubly linked list. This means that any potential data at that position has been overwritten and that the earliest offset for extractable user data starts with the *fd_nextsize* member. On the other hand, as the *PREV_INUSE* flag of the next chunk is set for this freed chunk, the *prev_size* field of the next chunk now contains a size value, hence not containing any user data anymore. The formula for the amount of not overwritten user data can now be calculated as follows:

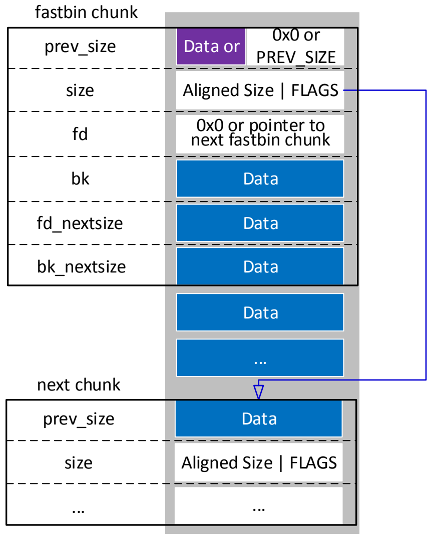


Fig. 11: Fastbin chunk in memory

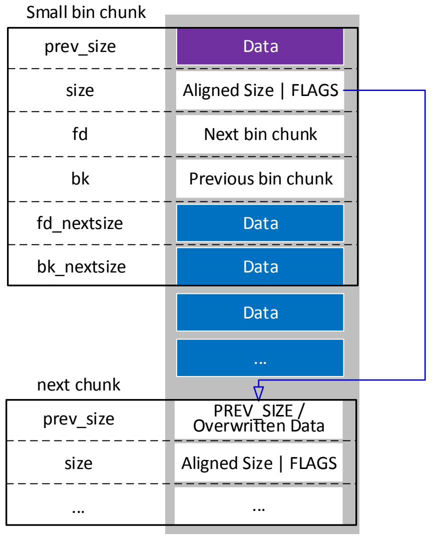


Fig. 12: Small bin chunk in memory

$$\texttt{not\_overwritten\_bytes} = \texttt{chunksize} - 2 * \texttt{sizeof(prev\_size)} - \texttt{sizeof(fd)} - \texttt{sizeof(bk)}$$

An interesting example are large bin chunks. As they also use the *fd_nextsize* and *bk_nextsize* members, which are the last members of the *malloc_chunk* struct, and like with small bin chunks the *prev_size* of the next chunks does not contain user data anymore, their remaining data

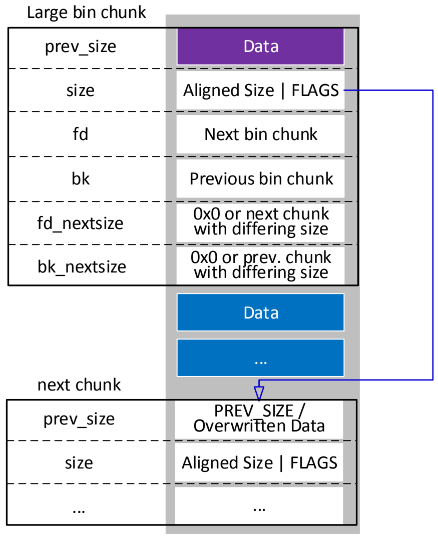


Fig. 13: Large bin chunk in memory

part reaches exactly from the end of the current chunks struct until the beginning of the next chunks struct (see Fig. 13). The amount of not overwritten user data can be calculated as follows:

$$\begin{aligned}\texttt{not\_overwritten\_bytes} = {} & \texttt{chunksize} - 2 * \texttt{sizeof(prev\_size)} \\ & - \texttt{sizeof(fd)} - \texttt{sizeof(bk)} \\ & - \texttt{sizeof(fd\_nextsize)} \\ & - \texttt{sizeof(bk\_nextsize)}\end{aligned}$$

The last case is the top chunk. As this chunk does not use any pointers, the data part starts like with allocated chunks right after the *fd* member and as it is the last chunk in an arena (and most of the time ends right at the memory region boundary), there is no following chunk whose *prev_size* field could be used (see also Fig. 14 on the facing page). The amount of not overwritten user data can hence be calculated using the following formula:

$$\texttt{not\_overwritten\_bytes} = \texttt{chunksize} - 2 * \texttt{sizeof(prev\_size)}$$

E. The Bottom Chunks Scenario

This section explains two chunks that are located at the bottom of a heap region (one described by a *heap_info* struct), that does not contain top chunk. Those chunks do not fulfill the requirements of normal chunks and might contain user data.

When the top chunk for a certain arena does not serve enough space for a new *malloc* request, the *sysmalloc*

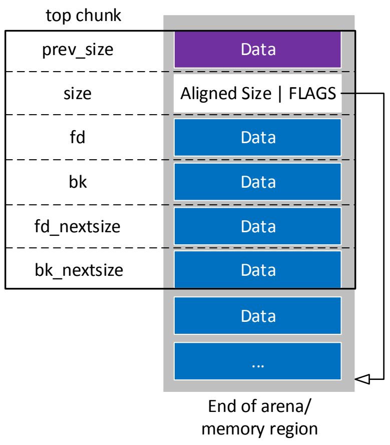


Fig. 14: Top chunk in memory

routine is used to decide how this request is handled. There are in essence four different scenarios regarding how this request ends up:

1) If certain conditions are met (like the request size must be at least as large as the *mmap_threshold*), a new memory region is created that serves space for an MMAPPED chunk (see Section II-C3 on page 9).
2) If the given arena is the main arena and scenario 1 has not been succeeded/is not be used, the main arena is simply enlarged using *brk*.
3) If the given arena is not the main arena, the heap is enlarged if the new value does not exceed *HEAP_MAX_SIZE* (see Section VII-B on page 49) and otherwise a new heap is created (including a *heap_info* struct at the beginning of the new memory region).
4) If scenario 2 failed and no arena has been specified, the *malloc* request fails.

If a new heap is created (second part of scenario 3), it results in some modifications of the old heap. These include at least the creation of a new top chunk within a new a heap region and the decrease of the old top chunk by some bytes.

Which further modifications are taken depends on the available space of the old top chunk. The following Listing 12 represents the relevant code excerpts for all possible modifications.

Independent from a certain initial situation, at least the following modifications are done:

- The macro `set_head` from line 2 sets the given chunk's size.
- The macro `set_foot` from line 4 sets the *prev_size* field of the next chunk.
- The macro `chunk_at_offset` from line 6 returns a chunk at offset $p + s$.
- Line 8 decreases the top chunk's size by MINSIZE bytes
- Line 9 sets the size field of the last chunk (in essence the last 4 bytes of the memory segment) to 0x1, which means a chunk size of 0 with the *PREV_INUSE* bit set.

The `if` and `else` statement will be explained in detail in the following sections. However in both cases, exactly two bottom chunks are created that do not conform to the properties of a chunk in general.

1) Following the if Statement: If the newly calculated `old_size` is at least as big as `MINSIZE`, the `if` branch is executed. Describing that scenario and the commands executed in natural language:

- The size of the old top chunk has been decreased by `MINSIZE`.
- In the case of the `if` statement, the bottom chunks both take exactly a size of `MINSIZE` which is why this value is subtracted.
- The `if` statement hence tests if the rest of the old top chunk leaves enough space to create a chunk that fulfills the minimum size requirement.
- As in this case two chunks use the space that normally is at least required for one chunk, those chunks do not fulfill the minimum size requirement for chunks.
- The `if` statement essentially creates the first bottom chunk (see also Fig. 15 on page 23 and Section II-E3 on the following page), sets the second bottom chunk's *prev_size* field and frees the rest of the top chunk.

Figure 14 and 15 illustrate those modifications. The initial situation is the same as illustrated in Fig. 14, with the top chunk as the last chunk. The location of the top chunk plus its size points to the end of the heaps memory segment (indicated by the pointer at the right corner of the Figure).

Fig. 15 on page 23 shows the resulting heap layout after following the `if` statement. As can be seen, the old top chunk has been changed to a normal freed chunk, which is part of a bin or fastbin and whose size now points to the beginning of another chunk. The bytes after the *new freed chunk* and before the end of the memory segment are now used for two new chunks (the *bottom chunks*) created by lines 9 and 12, each of them having an

```c
1  ...
2  #define set_head(p, s) ((p)->size = (s))
3  ...
4  #define set_foot(p, s) (((mchunkptr) ((char *) (p) + (s)))->prev_size = (s))
5  ...
6  #define chunk_at_offset(p, s) ((mchunkptr) (((char *) (p)) + (s)))
7  ...
8          old_size = (old_size - MINSIZE) & ~MALLOC_ALIGN_MASK;
9          set_head (chunk_at_offset (old_top, old_size + 2 * SIZE_SZ), 0 | PREV_INUSE);
10         if (old_size >= MINSIZE)
11           {
12             set_head (chunk_at_offset (old_top, old_size), (2 * SIZE_SZ) | PREV_INUSE);
13             set_foot (chunk_at_offset (old_top, old_size), (2 * SIZE_SZ));
14             set_head (old_top, old_size | PREV_INUSE | NON_MAIN_ARENA);
15             _int_free (av, old_top, 1);
16           }
17         else
18           {
19             set_head (old_top, (old_size + 2 * SIZE_SZ) | PREV_INUSE);
20             set_foot (old_top, (old_size + 2 * SIZE_SZ));
21           }
```

Listing 12: Glibc 2.23(malloc/malloc.c): Relevant Glibc code for Top Chunk Modifications

absolute size of `SIZE_SZ`∗2. These chunks contain only size information but no user data anymore. Annotations like *line 14* in Fig. 15 on the next page reference the corresponding line from Listing 12 and relate to the according field on the same altitude.

2) Following the else Statement: Regarding the `else` statement, there are two scenarios in which this branch is executed.

1) The old top chunk leaves exactly enough space for the two bottom chunks (`MINSIZE` bytes) but not more.
2) The old top chunk is larger than `MINSIZE` but smaller than `MINSIZE` ∗ 2.

In both scenarios, no additional freed chunk is created from the old top chunk. In the first scenario, the old top chunk is simply transformed in the two top chunks, again each with a size of `SIZE_SZ`∗2. In the second scenario, the old top chunk is again transformed in the two bottom chunks, but the first one (see Figure 15) has a bigger size and might contain user data. The amount of not overwritten user data depends mainly on the architecture. In can be calculated using the following formula:

$$\texttt{not\_overwritten\_bytes} = \texttt{old\_top\_size} - \texttt{MINSIZE}$$

This scenario only occurs if the old top chunk's size is smaller than `MINSIZE` ∗ 2 and hence leaves at most 8 byte of not overwritten data for 32 bit and 16 byte for 64 bit architectures. As however the second half of that space consists of the old top chunk's *size* field, only 4 byte for 32 bit and 8 byte for 64 bit architectures, respectively, of actual remaining user data can be found in that chunk at most. Regarding a 32 bit architecture, the following section illustrates the different scenarios using a live example.

3) Live Example: The following listings illustrate the different bottom chunk scenarios and use a chunk annotation that is taken from the Rekall plugin described in Section III-G on page 32, as this plugin has been used to analyze the bottom chunk scenario. The basis for this example is the program *Bottom chunks* from Section VII-D on page 51. One relevant function that is used by this program is `fillHeapWithChunks` (see Section VII-G on page 53), which fills a heap region (described by a *heap_info* struct) with chunks until its maximum size, but leaving at least `SIZE_SZ`∗2 bytes of free space. It can however be called with a value greater than zero for the second argument, which leads to more free space between the last chunk and the end of the heap: `SIZE_SZ` ∗ 2 + `argument`. The first argument to that function is only used to compensate the additional size of the *malloc_state* struct in the first heap.

The output in the following Listings represents the relevant parts of the *malloc_chunk* struct for the respective chunk and are the result from the Vtype language used by the Rekall framework, but with slight modifications for a better understanding (like renaming and stripping). The hexadecimal value after the `@` is the chunk's location and the value in the first column before each struct member is that field's offset within that struct. The value at the end of all other lines and sometimes in square brackets is that field's value.

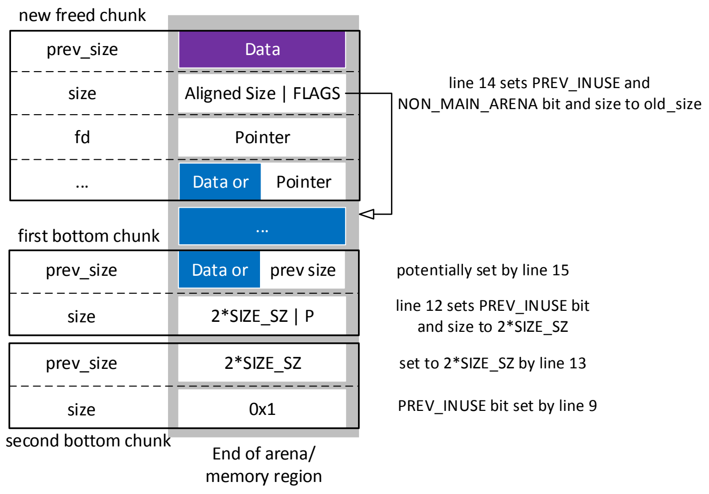

new freed chunk
prev_size
Data
size
Aligned Size | FLAGS
fd
Pointer
...
Data or
Pointer
line 14 sets PREV_INUSE and NON_MAIN_ARENA bit and size to old_size
...
first bottom chunk
prev_size
Data or
prev size
potentially set by line 15
size
2*SIZE_SZ | P
line 12 sets PREV_INUSE bit and size to 2*SIZE_SZ
prev_size
2*SIZE_SZ
set to 2*SIZE_SZ by line 13
size
0x1
PREV_INUSE bit set by line 9
second bottom chunk
End of arena/ memory region

Fig. 15: Top chunk modifications on old heap - Heap Layout after modification

The first Listing 13 illustrates the result from the first `fillHeapWithChunks` call in line 10 of Listing 39 and an additional *malloc* call afterwards (realized by going until the next `fillHeapWithChunks` call) and represents the first scenario of the `else` statement. As can be seen, the `last_chunk` fills up the entire space until the last two bottom chunks, leaving them each with a size of 8 bytes. The fields `fd` , `bk` and so on of the `last_chunk` and the `prev_size` field of the first bottom chunk do not contain pointers or an actual size, but only the user data from chunk `last_chunk` (in this case the string `zzzzzzzzzzzzzzzz...zzzz`).

The next Listing 14 illustrates the result from the second `fillHeapWithChunks` call in line 14 of Listing 39 and an additional *malloc* call afterwards and represents the second scenario of the `else` statement. The relevant changes to the previous Listing are the decreased size of the `last_chunk` and the increased size of the `first bottom_chunk` . The additional space in the `first bottom_chunk` is also indicated by showing the fields `fd` and `bk` , which do not contain any pointers, but not overwritten data from this part of the heap. So if an allocated chunk had user data on that position, it would now be probably still there. A proof of concept program can be found in Section VII-E on page 52 while Section VII-F on page 53 is showing an example output when dumping such a chunk.

The following Listing 15 illustrates the result from the third `fillHeapWithChunks` call in line 18 of Listing 39 and an additional *malloc* call afterwards and represents the first scenario in which the `if` statement gets executed. As can be seen, in addition to the `last_chunk` there is a `free_chunk` which is also part of a bin (the `fd` and `bk` pointers are set). This results from the 16 byte between the `last_chunk` and the space necessary for the bottom chunks, that are now fulfilling the requirement from Listing 12, line 10.

Listing 16 just illustrates what happens if the free space increases but stays beyond the required size of the current *malloc* request. The size of the `free_chunk` simply increases, the `prev_size` of the `first bottom_chunk` is set accordingly and everything else stays pretty much the same.

When running all scenarios on a 64 bit architecture, the only thing that differs is the amount of bytes needed to change between the different scenarios and the size of the `first bottom_chunk` in the scenario described by Listing 14 is 32 byte, leaving 16 byte of not overwritten data.

```
[malloc_chunk last_chunk] @ 0xB6CF04E0
 0x00 prev_size [unsigned int:prev_size]: 0x00000000
 0x04 size      [unsigned int:size]: 0x0000FB15
 0x08 fd       <malloc_chunk Pointer to [0x5A5A5A5A] (fd)>
 0x0C bk       <malloc_chunk Pointer to [0x5A5A5A5A] (bk)>
 0x10 fd_nextsize <malloc_chunk Pointer to [0x5A5A5A5A] (fd_nextsize)>
 0x14 bk_nextsize <malloc_chunk Pointer to [0x5A5A5A5A] (bk_nextsize)>

[malloc_chunk first bottom_chunk] @ 0xB6CFFFF0
 0x00 prev_size [unsigned int:prev_size]: 0x5A5A5A5A
 0x04 size      [unsigned int:size]: 0x00000009

[malloc_chunk second bottom_chunk] @ 0xB6CFFFF8
 0x00 prev_size [unsigned int:prev_size]: 0x00000008
 0x04 size      [unsigned int:size]: 0x00000001
```

Listing 13: No additional space between last chunk and bottom chunks

```
[malloc_chunk last_chunk] @ 0xB6AF0088
 0x00 prev_size [unsigned int:prev_size]: 0x00000000
 0x04 size      [unsigned int:size]: 0x0000FF65
 0x08 fd       <malloc_chunk Pointer to [0x5A5A5A5A] (fd)>
 0x0C bk       <malloc_chunk Pointer to [0x5A5A5A5A] (bk)>
 0x10 fd_nextsize <malloc_chunk Pointer to [0x5A5A5A5A] (fd_nextsize)>
 0x14 bk_nextsize <malloc_chunk Pointer to [0x5A5A5A5A] (bk_nextsize)>

[malloc_chunk first bottom_chunk] @ 0xB6AFFFE8
 0x00 prev_size [unsigned int:prev_size]: 0x5A5A5A5A
 0x04 size      [unsigned int:size]: 0x00000011
 0x08 fd       <malloc_chunk Pointer to [0x00000000] (fd)>
 0x0C bk       <malloc_chunk Pointer to [0x00000000] (bk)>

[malloc_chunk second bottom_chunk] @ 0xB6AFFFF8
 0x00 prev_size [unsigned int:prev_size]: 0x00000010
 0x04 size      [unsigned int:size]: 0x00000001
```

Listing 14: Eight bytes space between last chunk and bottom chunks

```
[malloc_chunk malloc_chunk] @ 0xB68F0088
 0x00 prev_size [unsigned int:prev_size]: 0x00000000
 0x04 size      [unsigned int:size]: 0x0000FF55
 0x08 fd       <malloc_chunk Pointer to [0x5A5A5A5A] (fd)>
 0x0C bk       <malloc_chunk Pointer to [0x5A5A5A5A] (bk)>
 0x10 fd_nextsize <malloc_chunk Pointer to [0x5A5A5A5A] (fd_nextsize)>
 0x14 bk_nextsize <malloc_chunk Pointer to [0x5A5A5A5A] (bk_nextsize)>

[malloc_chunk free_chunk] @ 0xB6BFFFE0
 0x00 prev_size [unsigned int:prev_size]: 0x5A5A5A5A
 0x04 size      [unsigned int:size]: 0x00000011
 0x08 fd       <malloc_chunk Pointer to [0xB6C00048] (fd)>
 0x0C bk       <malloc_chunk Pointer to [0xB6C00048] (bk)>

[malloc_chunk first bottom_chunk] @ 0xB6BFFFF0
 0x00 prev_size [unsigned int:prev_size]: 0x00000010
 0x04 size      [unsigned int:size]: 0x00000008

[malloc_chunk second bottom_chunk] @ 0xB6BFFFF8
 0x00 prev_size [unsigned int:prev_size]: 0x00000008
 0x04 size      [unsigned int:size]: 0x00000001
```

Listing 15: Sixteen byte space between last chunk and bottom chunks

```
[malloc_chunk last_chunk] @ 0xB69F0088
 0x00 prev_size [unsigned int:prev_size]: 0x00000000
 0x04 size     [unsigned int:size]: 0x0000FF4D
 0x08 fd      <malloc_chunk Pointer to [0x5A5A5A5A] (fd)>
 0x0C bk      <malloc_chunk Pointer to [0x5A5A5A5A] (bk)>
 0x10 fd_nextsize <malloc_chunk Pointer to [0x5A5A5A5A] (fd_nextsize)>
 0x14 bk_nextsize <malloc_chunk Pointer to [0x5A5A5A5A] (bk_nextsize)>

[malloc_chunk free_chunk] @ 0xB68FFFD8
 0x00 prev_size [unsigned int:prev_size]: 0x5A5A5A5A
 0x04 size     [unsigned int:size]: 0x00000019
 0x08 fd      <malloc_chunk Pointer to [0xB6C00050] (fd)>
 0x0C bk      <malloc_chunk Pointer to [0xB6C00050] (bk)>
 0x10 fd_nextsize <malloc_chunk Pointer to [0x00000000] (fd_nextsize)>
 0x14 bk_nextsize <malloc_chunk Pointer to [0x00000000] (bk_nextsize)>

[malloc_chunk first bottom_chunk] @ 0xB68FFFF0
 0x00 prev_size [unsigned int:prev_size]: 0x00000018
 0x04 size     [unsigned int:size]: 0x00000008

[malloc_chunk second bottom_chunk] @ 0xB68FFFF8
 0x00 prev_size [unsigned int:prev_size]: 0x00000008
 0x04 size     [unsigned int:size]: 0x00000001
```

Listing 16: Twenty four byte space between last chunk and bottom chunks

III. Plugin Implementation

The following sub sections describe the Python class *HeapAnalysis*, which represents the implementation of all analysis results described beforehand and the last sub section III-G the heap analysis plugins, which are based on that class. Our implementations are an extension to the Rekall Memory Forensic Framework (Google Inc, 2016c) and have been tested with Rekall versions 1.5.1 and 1.5.2.post1. At the point of writing, they support at least the Glibc versions 2.20, 2.21, 2.22, 2.23 and 2.24 on x86 and x64 architectures.Glibc version 2.24 was released after this research started and hence was not the basis for this work, but been tested against.

A. The HeapAnalysis Class

All plugins mentioned in this work are essentially Python classes that simply inherit from *HeapAnalysis* and use its methods to gather all relevant information, such as allocated chunks, to perform their analysis. The following list describes the most relevant public functions offered by the *HeapAnalysis* class that are intended e.g. for manual testing or plugin development.

init_for_task The function covered in Section III-B on the next page and normally the first function to use. It is responsible for initializing the *HeapAnalysis* class for a given task/process.

get_all_chunks This function returns all chunks that can be found (no matter if freed, allocated or MMAPPED). There are also functions for each chunk type and also the most relevant subtypes. They include *get_all_freed_chunks*, *get_all_freed_bin_chunks*, *get_all_freed_fastbin_chunks*, *get_all_allocated_chunks*, *get_all_allocated_thread_chunks*, *get_all_allocated_main_chunks* and *get_all_allocated_mmapped_chunks*.

get_main_arena Returns the internal main arena (either the real or the dummy arena).

search_chunks_for_pointers Searches all chunks for the given pointers and returns the ones containing at least one pointer (is e.g. used for the analysis of the zsh command history; see Section V-A on page 36).

get_aligned_address This function is mostly relevant for developing plugins. It takes a given address and returns an address, that is aligned regarding internal settings (see Section II-D2 on page 15)

get_aligned_size Works similar to the *request2size* macro explained in Section II-C2 on page 7. It is also mostly relevant for developing plugins. A scenario in that it for example gets relevant: If an investigator knows that a given struct is used by a process and knows its size, he can use this function to get the size of the resulting chunk and explicitly search for those chunks (see for example Section V-A on page 36).

chunk_in_use This function determines, if the given chunk is in use (not freed). It does that by taking a given chunk, gathering its next chunk, looking at that *PREV_INUSE* bit and afterwards returns *true* or *false* accordingly.

get_mallinfo_string It returns a string containing an output that is comparable with the *mallinfo* struct and the default print output, respectively (see also Section IV-A on page 35).

activate_chunk_preservation By calling this function it forces all allocated chunks to be stored in lists, which highly increases the speed of a second walk over those chunks. So plugins that need to iterate the chunks at least two times should use it (e.g. *heapdump* uses it).

to_string This function is not offered directly by the *HeapAnalysis* but by the *malloc_chunk* class. Hence, it can be called on each chunk instance returned from the *HeapAnalysis* class and returns the relevant data part as calculated in the subsections of Section II-D on page 14 (e.g. it omits the *fd* and *bk* field for a small bin chunk).

The *HeapAnalysis* is typically initialized for a given task on which it operates. From a high level perspective, the *HeapAnalysis* class works as follows:

1) The class instance is initialized for a given task (see Section III-B).
2) If the current task is no kernel thread, it tries to load the appropriate Glibc profile (see Section III-C on the next page).
3) One of the main tasks now is to gather the main arena (see Section III-D on page 28). This arena is important as it holds the bin information, a pointer to the other arenas and is used, due its location in the Glibc library, as verification during the initialization process (see Section III-D on page 28).
4) If the main arena has been found, its first chunk and, if existent, all other arenas are initialized.
5) If more than one arena is existent, all *heap_info* structs are enumerated and for each, their first chunk gathered.
6) The last step is to find any memory area that contains MMAPPED chunks and to get their first chunk.
7) The instance is now ready and all chunks can be walked. Walking in this context means, the current chunk's position plus its size is used to find the next chunk within the same memory region.

B. Initialization for a given Task

The relevant function at this point is *init_for_task*, which expects a *task* object as argument. This function controls all relevant functions used for the initialization process. If no major problem occurs and it returns *true*, the *HeapAnalysis* instance should be successfully set up and ready to walk chunks (see Section III-G on page 32). If something went wrong, it returns *false* and resets itself, which means process specific initializations, made before the error occurred, are revoked. This reset procedure is also called at the beginning of the *init_for_task* function, so no process specific information is kept anymore (relevant if a plugin is used for multiple processes).

The *task* object argument originates from the *LinProcessFilter* instance and provides all details about a given process that can be gathered at the moment by the Rekall framework. The first attribute of the *task* object that is examined is *mm*, which correlates to the *mm* member of the process' *task_struct* struct. As kernel threads normally have no associated *mm_struct*, this attribute can be used to easily exclude any kernel threads from the analysis, as the current heap analysis does not apply to the kernel.

If the current task is not a kernel thread, the linked list of *vm_area_struct* structs is used to initialize the Glibc profile. This step is repeated for every task, as processes can use different Glibc versions (takes effect if no debug information is provided by the investigator; see Section III-C on the next page). Depending on whether or not debug information for the current Glibc version are available, the global variable *mp_* (see Section II-C1 on page 5) is gathered from the mapped Glibc library, which will later serve as a verification regarding hidden MMAPPED chunks (see Section III-F on page 30 and Section IV-A on page 35).

If a Glibc profile is successfully loaded, *init_for_task* now tries to gather the main arena (see Section III-D on page 28). Depending on whether the main arena has been found in the loaded Glibc library or if the dummy arena has been created (see also Section III-D on page 28), the following steps differ. In the case the real main arena has been found, the first step is to verify that the circular linked list of arenas does loop and not contain more members than expected (see also Section II-C1 on page 5). The outcome of this test does however not stop the further initialization, but only print a warning if something unexpected has been detected. Now, all arenas contained in the linked list are initialized:

- The arena objects are kept in a list that is an attribute of the *HeapAnalysis* class instance.
- The bin and fastbin chunks for each arena are put in an arena specific list (each arena object has one list for bin and one for fastbin chunks). The reason for this step is efficiency. The freed chunks are used for tests on each chunk while walking allocated chunks in the memory (see also Section IV-A on page 35). As gathering the chunks each time from the memory dump would be very time consuming, they are kept

in a list in memory where iterating over them is way more faster.

- The top chunk is set as a separate attribute (see Section III-E on page 30).
- For all thread arenas, their corresponding *heap_info* structs are gathered and kept in an attribute for that arena. The process of getting them can be explained with Fig. 5 on page 16.
  - As shown in the most right memory region, the top chunk is located at the bottom in the second part of that region.
  - First the memory region described by a *vm_area_struct* struct, containing this top chunk is gathered.
  - The beginning of that memory region is now interpreted as a *heap_info* instance.
  - If the top chunk is located within its range (is determined with *heap_info*'s *size* field), this instance is returned.
  - If not, all following *heap_info* structs are gathered, until the correct one is found or the end of the surrounding memory region is encountered.
  - As soon as the correct *heap_info* struct instance is gathered, it is tested if its *ar_ptr* field points to the current arena.
  - If that is the case, the *prev* field is used to get all other *heap_info* structs (the one residing at the top chunk is the last one, so no need to look forward).
- After gathering the *heap_info* structs via the top chunk, a verification process starts that tries to find *heap_info* structs in all memory regions and compares that result with the already gathered *heap_info* structs. This process is described in more detail in Section IV-A on page 35.
- In the case of the main arena, the first chunk from the main heap memory region is gathered, which eases the task of walking the chunks later on.
- For all thread arenas, the first chunk is not held in the arena but in its *heap_info* objects, as they are the relevant descriptors for the corresponding memory regions.

The initialization process for the dummy arena scenario differs. On the one hand, there are no further arenas to be examined and on the other hand there are no usable bin and fastbin pointers from the dummy arena (see Section III-D on the following page and Section III-E on page 30). While it would be possible to gather at least freed bin chunks from the memory region, there are obviously none to gather (which is why the dummy arena scenario is used). And as freed fastbin chunks cannot be retrieved solely from memory, as they are not easily differentiable from allocated chunks (the next chunk keeps its *PREV_INUSE* bit set; see Section II-C2 on page 7), all bin lists stay empty and the only elements left to initialize are the first chunk and the top chunk. Both can be gathered by simply walking chunks in the main heap. The first chunk is located at the beginning of the memory region and the top chunk is the chunk, whose size reaches until the end of the memory region. Another difference is the fact, that the dummy arena's *system_mem* value is empty (because it cannot access the real field). It is simply set by calculating the difference between the end of the main heap (calculated by the offset of the top chunk plus its size) and the beginning of the main heap's memory region (gathered from the *vm_area_struct* struct).

The last step for both scenarios is to initialize the MMAPPED chunks, which are kept in a list attribute with the main arena object (no matter if dummy arena or not). Basically, this is done by looking at the beginning of all memory regions that are not known to be already part of an arena or contain a mapped file and then checking, if the data at this position complies with the existing knowledge about MMAPPED chunks (see Sections II-C3, II-D3 and III-F for details). As there are however also scenarios in which MMAPPED chunks hide somewhere behind other data, this approach is not sufficient and gets extended under specific circumstances. See Section III-F on page 30 for details.

After successful initialization, the functions listed in Section III-A on page 25 can be used to walk all chunks and gather further information.

C. Glibc profile

Because the debug information from Glibc are not included in the operating system specific profile (as it is gathered from the kernel that contains no details about Glibc's heap implementation), they must be provided in a different way. In the context of Glibc, the most relevant debug information to have are the struct details for *malloc_chunk*, *malloc_state* and *heap_info*. Also relevant, but not absolutely necessary, are the constant offsets for the two global variables *main_arena* and *mp_*. The offset to *main_arena* is only important if there are no further arenas and no freed bin chunks in the main heap to get the main arena (see also Section III-D), and primarily relevant to differentiate allocated chunks from fastbin chunks. The offset to *mp_* is relevant for result verification (see Section IV-A) and to detect the existence of hidden MMAPPED chunks (see Section III-F). Besides that, these constant offsets are not important to explore the heap.

The ideal scenario is that the investigator has some sort of access to the debug information of the Glibc library in use, and can provide it to the plugin. While debug information is typically in certain sections in the ELF file, they normally cannot be extracted from the mapped file in memory, as those sections are normally not loaded in memory. They hence must be gathered e.g. from the ELF file on disk, which requires in most scenarios local access to the target system. This information can then be provided to the plugins (see Section III-G) and the *HeapAnalysis* class will use them internally.

But in the context of a Glibc profile, there might be a further option, depending on the current scenario. The problem with custom kernels and the need for a custom profile is not that similar to the generation of a Glibc profile. While most distributions build their own Glibc version with custom compile flags and/or patches to the source code, which potentially results at least in varying constant offsets, the retrieval of this debug information might be much easier. The reason is that many Linux distributions come with binary packages. That means, software like Glibc is not compiled on the target system, but the software is pre-compiled. If the investigator knows which Glibc version is in use, he simply can download the binary package (most of the times also the debug package) from a repository server, and extract the relevant information from it.

All the previous options might however not always be available.

- The target system could be locked and the credentials are unknown
- Local access might be available but the operating system is outdated and the repositories do not serve the necessary packages anymore.
- The Glibc version could be customized and compiled without debug/symbol information, while no details are available on how it has been compiled (e.g. with which compile flags).

If the investigator still manages to get a working profile for the target operating system (e.g. by building a similar machine and creating it), but fails to generate an appropriate Glibc profile (e.g. the one used on the target system is customized), he is in most cases still able to investigate the heap of user space processes with plugins using the *HeapAnalysis* class. The reason is, the Python module containing the *HeapAnalysis* class comes with two additional classes called *GlibcProfile32* and *GlibcProfile64* (the first for x86 and the second for x64 architectures). They implement a basic Glibc profile that serves the relevant structs for the currently supported Glibc versions (see Section III-G). If the user does not provide debug information, a profile is automatically loaded using one of these classes.

The only task at this point, besides choosing between a 32 bit or 64 bit architecture profile, is to determine the Glibc version used by the current process. This is important, because with version 2.23, the *malloc_state* struct got a new field (*attached_threads*; see also Section II-D2 on page 15). The question to answer is hence: Is the Glibc version smaller or greater/equal to 2.23? As no debug information is available, this detail must be gathered elsewhere. At this point, the linked list of *vm_area_struct* structs comes again into play. By examining the *file* object of the *vm_area_struct* structs, holding the mapped Glibc module, this version information can typically be extracted from memory. The library's filename on disk contains normally the version string (e.g. `libc-2.23.so`) and thus does the file object. This version information decides now, which version of the *malloc_state* struct definition is used and the Glibc profile is loaded.

The only debug information missing in this last scenario are the constant offsets for *mp_* and *main_arena*. But as mentioned in the beginning, this fact does not always prevent a successful heap investigation.

D. Getting the Main Arena

One of the first tasks the *HeapAnalysis* class is performing during its initialization is to locate the main arena. Besides holding important information such as the arena's size and the bin pointers, its location within the mapped Glibc library is a good indicator that the information examined during the search process are correct (the details will be explained in this section).

The class *HeapAnalysis* implements different techniques to get the main arena. The most reliable method to get the main arena can be used if debug information, or more precisely, the constant offset for the *main_arena* symbol is available. This is typically only the case, if the investigator gathered the relevant debug information upfront (e.g. from the target system) and provides them to the plugin (gathering it from the memory dump is not easily possible; see Section III-C on the preceding page). If that is the case, the location of the main arena can be easily calculated. As the *main_arena* offset is relative to the start of the mapped Glibc file and not an absolute address in the virtual address space, the only information besides the *main_arena* offset needed is the virtual address where that mapped file begins. This is one of the reasons, why for loading the Glibc profile, the linked list of *vm_area_struct* structs is needed (mentioned in Section III-B on page 26; the other reason is covered in Section III-C on page 27). The relevant task is now to go through the *vm_area_struct* structs, examine their *file* member and find the *vm_area_struct* struct for the beginning of the mapped Glibc library (typically the one with the lowest *vm_start* value). The last step is now to add the *main_arena* offset to the virtual address pointing to the beginning of the mapped Glibc library, and dereference the data at that address as a *malloc_state* instance.

If however the constant offset for *main_arena* is not available, the main arena cannot be directly determined and hence is searched via two different techniques. The first assumes that there is more than one thread and hence also more than one arena. As only the main arena struct is stored in the mapped Glibc library, all others are typically located at the beginning of a memory region, right after a *heap_info* struct (see also Section II-D1 on page 14). So, for each memory region described by a *vm_area_struct* struct (with some exceptions like mapped file regions or stack frames), the beginning of it is treated as a *heap_info* struct. If its *ar_ptr* field points right after itself and its *prev* field is null (the first *heap_info* struct points to no other *heap_info* instance and if it sits right before the arena, it is the first instance), the arena pointed to by *ar_ptr* is temporarily saved. If at the end at least one arena could be identified, it is tested whether or not following their *next* member ends up somewhere in the memory region of the mapped Glibc library. If that is the case, the address pointing in the Glibc is treated as the main arena.

If no further arenas could be identified (that means, there is only the main heap and potentially some MMAPPED chunks), the second technique is used. It leverages the fact that each bin chunk holds a circular doubly linked list. The idea is to follow the *bk* pointer of a bin chunk, until it leads to the main arena. To get such a bin chunk (there is yet no arena, whose bins could be used), every chunk on the main heap is examined for its *PREV_INUSE* bit (see also Section II-C2 on page 7). If this flag is unset in any chunk, the previous chunk is most probably a freed bin chunk. The previous chunk's *bk* field is now followed, chunk after chunk, in order to eventually end up in the main arena. If at some point, the *bk* field points into the mapped Glibc library, the main arena is most probably found.

In this scenario, there is however still a problem. The *bk* field points to the middle of the arena (to a bin) and not to the beginning. While it would be possible to exactly determine the position within that main arena by examining the chunk's size (each bin typically contains only chunks of a given size (range)) this approach is not always perfectly reliable because of varying bin sizes (see Section II-C4 on page 11). The method used instead to get to the beginning of the main arena is a search for the top chunk pointer. There are two reasons to use this approach:

- The offset of the top chunk pointer in the *malloc_state* is fixed for a specific Glibc instance and hence reliable. So, if the virtual address of that field is found, the distance to the beginning of the arena can be easily calculated.
- There needs to be a reliable process to correlate an expected field with the current value. The *top* field serves such a correlation, as the chunk that it points to, should reach until the end of the memory region (the top chunk's offset plus its size).

The process to get to the *top* field is now to walk backwards from the current bin, treat each pointer sized value as a pointer and check if it points inside the main heap and meets the requirements of the top chunk. The step size for walking back is equal to the pointer size for the current architecture, as between the top chunk and the bins are no other data types than pointers. As soon as the *top* field is found, the main arena's address is calculated by subtracting the *top* field's offset within the *malloc_state* struct from the found *top* field's memory address.

As described in Section II-C4 on page 11, fastbin chunks do not use the *bk* pointer and are not linked in a circular list. Hence, even if they could be easily identified on the heap (which is not the case; see the explanation in Section III-B on page 26 regarding the dummy arena scenario), only with freed fastbin chunks there is no trivial way back to the corresponding *malloc_state* struct. This is why at this point (all other techniques failed), a dummy arena is used (see also Section III-E on the following page).

The dummy arena is a last resort kind of way to cope with the fact that there are no options left to get the main arena. The reason it is created anyways, is to be able to use all internal and public methods (see also Section III-A on page 25) normally, without having to change the internal logic. The dummy arena is in fact only an instance of the *malloc_state* class, whose fields have however no connections with any data of the process' virtual address space. The connection to the heap is created with separate class attributes that are set during the initialization (see Section III-B on page 26). For implementation specific information see also Section III-E.

No matter how the main arena is gathered, the sanity check regarding the arena count (described in Section III-B on page 26) is done afterwards (except for the dummy arena scenario).

E. The Dummy Arena

The dummy arena is necessary if the real main arena cannot be found, which has already been discussed in Section III-D on page 28. As also described in that section, the dummy arena is an instance of the *malloc_state* class that has no connection to the process' virtual address space. That means, its members do not read any information from actual memory and hence must be set manually. The separation from the process' address space is also a protection mechanism, to prevent arbitrary data from being accidentally used for this instance or that setting some values might get written in the memory dump (if e.g. write support within Rekall is activated).

From an implementation point of view, the problem whith this dummy object arises, when trying to access members that are pointers. To understand the problem, it is necessary to understand how using attributes of such objects works within Rekall. As those objects have typically an associated address space, the framework knows where to look for any data if a specific virtual address is given. This means, when using e.g. the *top* field of a *malloc_state* instance, which is a pointer to a *malloc_chunk* struct, the framework knows that this field points inside the process' address space and hence tries to automatically dereference this struct right there. The result of accessing the *malloc_state*'s *top* member is normally an instance of the *malloc_chunk* class, which represents the *malloc_chunk* struct with the values from the actual top chunk.

With the dummy arena, this context is however not existent. While it would be theoretically possible to set the *top* field to the address of the top chunk within the process' address space, using the actual top chunk via this field would not work, as the framework tries to access this address without the correct address space (leading normally to a *malloc_chunk* instance with zero values for all fields). To efficiently circumvent this problem, the attribute *top_chunk* has been added to the *malloc_state* class, which not holds an address but an already instantiated *malloc_chunk* object for the actual top chunk and with correct address space. As the address spaces of the *malloc_state* object does not have to correlate with any objects contained in additional attributes, they can exist side by side.

The *top_chunk* attribute is the only additional attribute in the context of the dummy arena. While it would be necessary to do it for all pointer fields, the rest are not needed in the dummy arena scenario. There are no further identified arenas (hence the *next* field is not required), it was not possible to find any small or large bin chunks and fastbin chunks cannot easily be detected by solely looking at chunks in memory (see also Sections III-B and III-D). The only task left for the dummy arena is to set the *system_mem* member. As this is a not a pointer but a simple numeric field, it can be set without having to worry about address spaces.

F. MMAPPED Regions

While the identification of memory areas belonging to an arena is in most scenarios pretty reliable (see Section III-B on page 26 and Section III-D on page 28), identifying regions containing MMAPPED chunks is not. The reasons are:

- There is a lack of distinctive structs or reliable pointers to them.
- Any unnamed mapped memory region might contain MMAPPED chunks and they can be located anywhere in the process space.

It seems like pointers to those chunks are at most saved in stack frames of functions working with them, but not in any book keeping struct. So one way to identify them could be to interpret all pointer-sized bytes from all stack segments (from all threads) as pointers, follow them and verify the information residing at that position. This approach has however two problems:

1) It is pretty error prone (interpreting arbitrary data as pointers) and, depending on the stack size, pretty time consuming (each pointer must be read, followed, the data it points to initiated as a chunk object and its values tested).
2) It might miss some MMAPPED chunks. In the case, where an MMAPPED chunk pointer is not used anymore, its value might get overwritten by newer stack frames, but if the chunk has never been freed, it still exists in the memory space, serving potentially valuable information.

Because of the lack of alternatives, the current approach to decide, whether or not a certain memory region contains MMAPPED chunks, is to perform a plausibility check. Because a memory region containing solely

MMAPPED chunks (differing examples are described later in this section) also begins with an MMAPPED chunk, the first bytes are treated as such, and tested for the following characteristics:

- The *prev_size* field must have a value of zero.
- The chunk's size (the value of the *size* field without any flags) must be at least as large as the page size.
- The chunk's size must be evenly divisible by the page size.
- The chunk's offset plus its size must not exceed the boundary of the containing memory region.
- The location of the chunk must be evenly divisible by the page size (primarily useful for subsequent and hidden MMAPPED chunks).
- The *PREV_INUSE* and *NON_MAIN_AREA* bits must be unset and the *IS_MMAPPED* bit set.

Only if all of those properties are given, the corresponding memory region is considered to contain MMAPPED chunks. Those checks are also done on any subsequent data of the same memory region, before they get included as chunks.

While MMAPPED chunks are normally within an exclusive mapped memory region, it can happen that those chunks are placed at the bottom of a mapped memory region, containing also different data such as stack segments (see Figure 16). As starting the search for MMAPPED chunks (in the stack scenario) at the beginning of the memory region might lead to false positives (interpreting stack data as chunks), a certain method is used, which is called *EBP unrolling* within this work. The basic approach is to follow all saved EBP pointers, starting with the base pointer gathered from the *pt_regs* struct (used to save register values on context changes). As each EBP value points to the next saved EBP, just following those pointers leads to the first saved EBP, probably located near the beginning of the stack segment. This process is also illustrated in Figure 16.

Once the offset of the first saved EBP is identified, the next step is to search backwards, from this point on, for the first MMAPPED chunk. As MMAPPED chunks are only located at addresses evenly divisible by 4096 (minimum page size), only such addresses are examined. In cases where the EBP value does not point to a saved EBP in the stack segment (because it is e.g. used for carrying different data) the method stays basically the same, except it does no *EBP unrolling* and starts at the beginning of the memory region with the backwards search.

In cases where the missing MMAPPED chunks are not located after stack segments but *hide* somewhere else, the search scope must be extended while increasing the risk of false positives. The only regions that can be excluded are the stack and heap segments that have been already examined and those holding the content of mapped files (the *vm_area_struct* struct references a file object), as they could not be identified to ever contain other data (which would also be very unexpected, as those regions represent the file's content). The search process for this case stays the same, but without the *EBP unrolling*. After the first hit within a memory region (no matter if after a stack or somewhere else), this first MMAPPED chunk is being used to walk the potentially following chunks.

If after the search process the values still do not correspond, but some new MMAPPED have been identified, there is no simple way of verifying the validity of those chunks (they could be arbitrary data mistakenly interpreted as MMAPPED chunks). This is why an additional verification step is performed that gives the investigator an indication whether or not the newly identified MMAPPED chunks seem to be valid. As mentioned in the beginning, pointer to MMAPPED chunks are at most saved in stack frames, which is why all stack segments are now searched for pointers to those chunks. When at least one pointer to a potential MMAPPED chunk can be identified, this fact is treated as a good indicator for this chunk's validity. Because pointer used in user space processes point not to the beginning of a chunk but to the beginning of the data part (see also Section II-D3 on page 17), this needs to be taken into account (the appropriate offset must be added to the pointer, before starting to search).

The logging mechanism offered by Rekall is used to inform the investigator about the whole process described in this section. This includes the initialization of the search, the result of the stack pointer search and whether or not the identified MMAPPED chunks correspond at the end with the information from the *malloc_par* struct. The output about the stack pointer search reports for how many of the new MMAPPED chunks a pointer on the stack has been found. Listing 17 shows an example of this verification step while using the *heapinfo* plugin on a process with *hidden* MMAPPED chunks. Because all MMAPPED chunks could be found (and hence the final values correspond with the *mp_* values), no stack pointer search is performed.

In order to prevent false positives, the *HeapAnalysis* class starts a search for hidden MMAPPED chunks only if the current information about MMAPPED chunks seem to be incorrect (will be explained in the following Section).

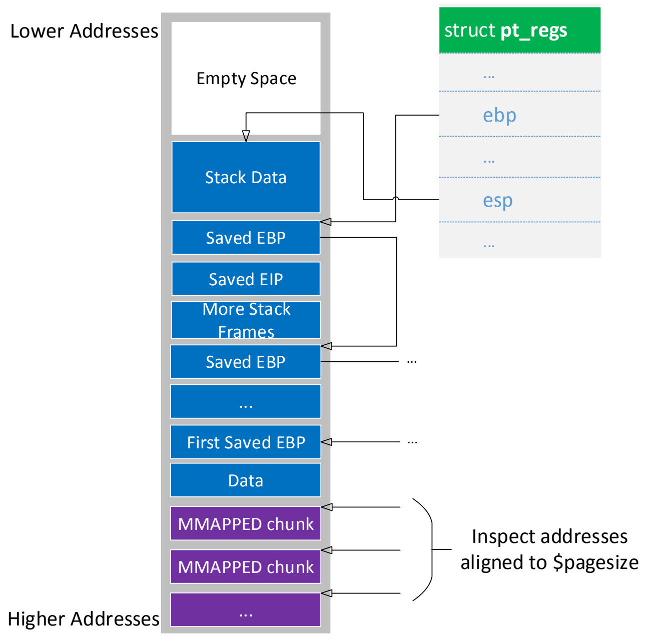


Fig. 16: Hidden MMAPPED chunks - EBP Unrolling

```
2016-09-25 15:39:42,346:WARNING:rekall.1:The values from the malloc_par struct don't correspond to our
    found MMAPPED chunks. This indicates we didn't find all MMAPPED chunks and that they probably hide
    somewhere in a vm_area. So we now try to carve them, what might lead to false postives.
2016-09-25 15:39:42,346:WARNING:rekall.1:Seems like we didn't find (all) MMAPPED chunks behind stack
    frames. We now search in all anonymous vm_areas for them, which might however lead to false
    positives.
2016-09-25 15:39:42,827:WARNING:rekall.1:Seems like all missing MMAPPED chunks have been found.
```

Listing 17: Warning messages while searching for hidden MMAPPED chunks

G. The Heap Analysis Plugins

The four main heap analysis plugins are:

heapinfo Provides an abstract overview over the number of arenas, chunks and their sizes.
heapdump Dumps all allocated and freed chunks to disk in separate files for further analysis.
heapsearch Searches all chunks for the given string, regex or pointer(s).
heaprefs Examines the data part of the given chunk(s) for any references to other chunks.

They will be demonstrated in Section V.

1) The heapinfo Plugin: An example output for the *heapinfo* plugin can be seen in Listing 18. The first line shows the command line call to execute the plugin, followed by the plugin's analysis result. The plugin output has been split up into two parts to make it more readable.

The command line arguments are as follows:

mem.dump The ram dump from the *Arch* instance.
arch.json The Linux profile generated for the *Arch* instance.
heapinfo The plugin name to be executed with the *Rekall* framework.
arch-libc_2.23.json The Glibc profile, containing debug information like struct definitions in Vtype and constant offsets.
8703 The process ID to be analyzed.

For each analyzed process, exactly one line of output is generated. The following list explains all columns of that output.

PID The PID of the analyzed process.
Arenas The amount of discovered *malloc_state* instances.
Heap I. The amount of discovered *heap_info* instances.
Non MMAPPED chunks The amount of all main and thread heap chunks, excluding MMAPPED chunks.
N.M. chunks size The summarized size of all main and

```
rekall -f mem.dump --profile arch.json heapinfo --glibc_profile arch-libc_2.23.json 8703

 PID   Arenas Heap I. Non MMAPPED chunks N.M. chunks size
------ -------- --------- -------------------- ------------------
8703  3      12      448             16368064

 MMAPPED chunks MMAPPED size Freed chunks Freed size
---------------- -------------- -------------- ------------
20          10043392   159         28464
```

Listing 18: Heapinfo example Output

thread heap chunks (excluding MMAPPED chunks), taken from their *size* member.

MMAPPED chunks The amount of all MMAPPED chunks.

MMAPPED size The size of all MMAPPED chunks, taken from their *size* member.

Freed chunks The amount of all freed bin and fastbin chunks, not including top chunks.

Freed size The size of all freed bin and fastbin chunks (not including top chunks), taken from their *size* member.

Besides the standard output, this plugin is capable of printing further details like struct information. This is accomplished by command line options, that can be specified by the investigator. The following list shows these options:

print_objects Prints each arena struct (*malloc_state*) and its top chunk and for each thread arena the corresponding *heap_info* structs and their first chunk.

print_allocated, print_freed, print_mmapped These options print the *malloc_chunk* structs of the corresponding type (*allocated*: all allocated chunks including MMAPPED chunks, but makes them distinguishable with special markers; *freed*: bin and fastbin chunks, which are also marked; *mmapped*: only MMAPPED chunks).

print_mallinfo Prints the content of the *mallinfo* struct according to the description from Section IV-A on page 35 (useful for manual result verification).

There are also two command line options that are directly served by the *HeapAnalysis* class and hence available for every plugin using this class:

glibc_profile This option expects a file containing the Glibc debug information, that are loaded internally (see Section III-C on page 27).

prevent_chunk_preservation This option prevents internally any chunk preservation mechanism. It is useful in cases where a process has a huge amount of chunks and the memory resources of the analysis system are limited.

2) The heapdump Plugin: The *heapdump* plugin dumps all chunks in separate files using unique filenames. The relevant function to get a chunk's data is *to_string*, mentioned in Section III-A on page 25. As it uses the formulas discussed in the subsections of Section II-D on page 14, it e.g. omits the content of the *fd* and *bk* and the next chunk's *prev_size* field, when dumping a small bin chunk and additionally the *fd_nextsize* and *bk_nextsize* members for large bin chunks. The omitted data is indicated within the filename by *CHUNKSIZE* and *DUMPEDSIZE*. Some file name examples can be seen in Listing 20. The format used is: `PID.CHUNK-TYPE_OFFSET_CHUNKSIZE_DUMPEDSIZE`.

PID The PID of the process. It is useful when dumping multiple processes in the same directory.

CHUNK-TYPE Can be one of the following: *allocated-main*, *allocated-thread*, *allocated-mmapped*, *freed-bin*, *freed-fastbin*, *top* and *bottom*.

OFFSET The address of the *malloc_chunk* struct within the virtual address space.

CHUNKSIZE The size taken from the chunk's *size* member (no flag bits).

DUMPEDSIZE The amount of bytes that have been dumped into the file. This value can in some cases be zero (most often with bottom chunks, but also e.g. with a freed bin chunk with a size of 16; see Section II-D4 on page 18), but the file is created anyways to not hide the existence of that chunk from the investigator.

The only command line option this plugin introduces is *dump_dir*, which specifies the destination folder in which all chunks should be dumped.

3) The heapsearch Plugin: The heapsearch plugin helps the investigator in identifying a chunk of interest. This can e.g. be done by searching for a specific string or pointer, which is expected to be contained in a chunk. When a match is found, the according *malloc_chunk*

```
rekall -f mem.dump --profile arch.json heapdump --glibc_profile arch-libc_2.23.json
-D destinationFolder 8703

2016-09-19 01:37:37,039:WARNING:rekall.1:Chunk preservation has been activated. This might consume large
     amounts of memory depending on the chunk count. If you are low on free memory space (RAM), you
    might want to deactivate this feature with the prevent_chunk_preservation cmd option. The only
    downside of deactivation is in some cases a longer plugin runtime.

Pid: 8703 - Dumped 478 allocated, 139 freed bin, 20 freed fastbin and 3 top chunks.
```

Listing 19: Heapdump example Output

```
8703.allocated-main-chunk_offset-0x9125408_size-16_dumped-12.dmp
8703.freed-bin-chunk_offset-0x9126df8_size-88_dumped-72.dmp
8703.freed-fastbin-chunk_offset-0xb56fffe0_size-16_dumped-8.dmp
8703.top-chunk_offset-0x965cd38_size-123592_dumped-123584.dmp
```

Listing 20: Filename examples for the heapdump Plugin

struct is printed, including its virtual address (see Listing 21 for an example output).

The following list shows the plugin's options:

pointers Prints chunks that contain exactly the given pointer(s). The pointer(s) can be given as (hexa)decimal numbers.

regex Searches all chunks with the given regex and prints all hits.

string Searches all chunks for the given string and prints all hits.

chunk_addresses Expects address(es) belonging to a chunk(s) of interest, and prints all chunks having a pointer somewhere into the data part of that chunk(s).

search_struct Includes the *malloc_struct* fields in the search process, which means the *size* field for all chunks and *prev_size, fd, bk, fd_nextsize* and *bk_nextsize* for bin chunks. This is normally not desired and hence deactivated by default.

4) The heaprefs Plugin: The heaprefs plugin analyzes the data part of a chunk for pointer(s) to other chunks. If it finds one, it marks the data part containing the pointer and prints the address of the target chunk in the *Comment* column (see Listing 24 for an example output). The heaprefs plugin introduces only one option:

chunk_addresses The address(es) belonging to chunks of interest. Those chunks are then examined for references to other chunks.

```
Result for needle(s) 162898536

[malloc_chunk malloc_chunk] @ 0x09B5A3A8
  0x00 prev_size [unsigned int:prev_size]: 0xFFFFFFFF
  0x04 size     [unsigned int:size]: 0x00000061
  0x08 fd      <malloc_chunk Pointer to [0x09ad0c50] (fd)>
  0x0C bk      <malloc_chunk Pointer to [0x8ed2ca21] (bk)>
  0x10 fd_nextsize <malloc_chunk Pointer to [0x00000001] (fd_nextsize)>
  0x14 bk_nextsize <malloc_chunk Pointer to [0x099c75e0] (bk_nextsize)>
```

Listing 21: Example output for the heapsearch Plugin

IV. Evaluation

This section describes the evaluation of the *HeapAnalysis* class and its plugins.

A. Result Verification

The verification of results is an important and elementary step in order to show the reliability of our methods and techniques. All tests have been conducted in the following environments while using different Glibc versions:

- Arch Linux 32 bit, x86, Kernel Version 4.4.5-ARCH, Glibc Versions: 2.20, 2.21, 2.22, 2.23 and 2.24
- Arch Linux 64 bit, x64, Kernel Version 4.4.5-ARCH, Glibc Versions: 2.20, 2.21, 2.22, 2.23 and 2.24

The *HeapAnalysis* class implements multiple functions, which compare the currently examined data with our expectations on every chunk while processing memory:

- Test for correct flags (e.g. *NON_MAIN_ARENA* for chunks in a thread arena).
- Is the chunk's address aligned according to *aligned_ok* (see Section II-D2)?
- Size checks (see also Section II-C2):
  - Is the size larger or equal to the *MINSIZE*?
  - Does the size of the chunk exceed the boundaries of the current memory region?
  - Is the size evenly divisible by *MALLOC_ALIGNMENT*?
- Allocation status tests (for main and thread arena chunks): Is a presumably allocated chunk part of any bin or fastbin? Has a chunk following a freed bin chunk the *PREV_INUSE* flag set? ...
- In the case of MMAPPED chunks there are some additional tests (see Section III-F for more details).

There are two further tests to mention, which are done while walking chunks in memory. As described in Section II-D2 on page 15, the first chunk must not necessarily follow directly the *heap_info* struct, but depending on the *heap_info* and also the *malloc_state* struct's size, there can be a gap of some bytes (which consists solely of null bytes). The verification step in this case is, to predict the chunk's size, examine the bytes after the *heap_info* struct and look for the first non-zero bytes which should be the first chunk's size field. If that location does not meet the expectation, this indicates incorrect debug information and a warning is printed.

The second test verifies, that walking the chunks in memory leads to the expected end. This is in the case of the main arena and the last *heap_info* struct of a thread arena done by looking for the top chunk, whose size should point to the end of the current memory region. In the case of all other *heap_info* structs, the test searches for the bottom chunks (see Section II-E on page 20). If walking the chunks does not lead to the expected end (either top or bottom chunks), again a warning is printed.

Regarding MMAPPED chunks, there is another verification step which involves the global variable *mp_* (an instance of the *malloc_par* struct; see also Section II-C1). While it does not offer any details about main or thread arenas and their chunks, its fields *n_mmaps* and *mmapped_mem* hold the number and size of all MMAPPED chunks, respectively. If the offset for *mp_* is provided, the *HeapAnalysis* class uses the struct to verify the number and size of all identified MMAPPED chunks. If there are any discrepancies, it tries to identify hidden MMAPPED chunks and if that does not resolve the issue, it prints a warning (see Section III-F).

Regarding size comparisons, there are two further verification steps. The first one compares the size of all identified chunks (arena related but also MMAPPED chunks) and the corresponding *heap_info* and *malloc_state* structs with the size of their memory regions (described by *vm_area_struct* structs), while taking slack space for the heap and MMAPPED regions into account (see Section II-D1 and Section II-D3). The second one compares the *system_mem* values of the identified arenas with their related memory regions.On any deviation, a warning is printed.

The last two checks that should be mentioned, try to verify the amount of identified arenas (*malloc_state* structs) and *heap_info* structs. As described in Section II-C1 on page 5, the maximum number of arenas is either determined by the one CPU core scenario (3 on a 32 bit and 9 on a 64 bit architectures), defined by the macro *NARENAS_FROM_NCORES* or manually set via *mallopt* (the *malloc_par* struct's *arena_max* field). The arena verification functionality first tries to determine which one of the three is relevant and afterwards compares that value with the number of identified arenas. The workflow is as follows:

- First, the availability of *mp_* is tested. If that fails, it aborts, as without access to the *arena_max* field, no reliable statement about the correct number of arenas can be made.
- The next step is to test the *arena_max* value. If this field has not been set manually, it typically has a value of zero.
- If it is not zero, this value is used for verification.
- Otherwise, the number of CPU cores is gathered from the plugin *cpuinfo*, offered by Rekall.
- If it is more than one, the maximum number is the result from the *NARENAS_FROM_NCORES* macro, if not, it is `NARENAS_FROM_NCORES` $+ 1$.
- The last step is to compare the maximum number with the actual number of identified arenas. If it is higher than the maximum, a warning is printed.

As it was not possible to identify a maximum value for *heap_info* structs, the verification in this case is different. The goal is to verify, whether or not some *heap_info* structs have been mistakenly not found (e.g. from an unidentified arena), while assuming that all relevant arenas have been already identified. The general approach is to investigate all anonymous mappings except for the one containing the main heap, interpret the beginning of that region as a *heap_info* struct, gather all potentially following *heap_info* struct and examine their *ar_ptr* field. If the field of one of those potential *heap_info* structs points to a known arena, it is treated as a valid instance. If one of those valid *heap_info* structs is not in the list of already known *heap_info* instances, a warning is printed.

B. Completeness

While the results in this work do not cover 100% of Glibc's heap implementation, various steps have been performed to identify all relevant information and scenarios, relevant for the memory forensics perspective.

- Various tests with self-written programs.
  - Varying order of allocations (e.g. at first MMAPPED chunks, then main arena chunks; has led to the hidden MMAPPED chunks scenario).
  - Allocation of thousands of chunks in all arenas (led to the main heap distributed over two *vm_area_struct* structs and to multiple *heap_info* structs within one memory region; see e.g. Section II-D1).
  - All special cases we are aware of.
  - ...
- Source code analysis and verification with proof of concept code.
- *HeapAnalysis*'s internal verifications, that revealed e.g. the bottom chunk scenario (the test for hitting the bottom of a memory region, described in Section IV-A on the previous page).
- The evaluation done in Section V, which showed for the given applications that it was possible to completely gather the information in question from the heap.
- Tests on varying operating systems, architectures and Glibc versions (see also Section III-G on page 32).
- Performing a heap analysis on every applicable process (all processes except kernel threads) in the environments described in Section III-G on page 32, while performing the verifications described in Section IV-A on the previous page.

V. Application on Real World Scenarios

This section illustrates the application of our plugins on real world examples. This is on the one hand done by describing the analysis process itself and on the other hand by highlighting the advantage of using our plugins instead of a raw search through the entire heap. The analysis performed in the following subsections was done using a black box approach, which means that, if not specified otherwise, no process related details from the source code, like e.g. struct definitions, were necessary to be gathered beforehand.

A. zsh

The previously described *bash* plugin of Rekall and Volatility searches the whole heap space for timestamp strings that are prefixed with a hashtag, and afterwards searches it again for a history struct that points to the timestamp, in order to identify the issued command

```
Result for needle(s) #$@%&*()

[malloc_chunk malloc_chunk] @ 0x09BCF830
```

Listing 22: Using heapsearch to search for chunks containing issued zsh commands

```
2324 4025 262a 2829 0000 0000       #$@%&*()....
```

Listing 23: Hexdump of a chunk containing an issued zsh command

strings. The output of the plugin is a list of command entries, each consisting of the issued command and the corresponding timestamp. Our goal is to identify the same information for the zsh. The corresponding analysis in this section has been done in the same environments listed in Section IV-A.

The zsh process to analyze contained 142 executed commands in its history (when examining the history with the *history* command); the first one was `ps aux` and the last one `#$@%&*()`. The first part of the analysis process has been done in a black box approach, meaning no internal information about how zsh stores commands or time information have been gathered in any way beforehand. The only knowledge basis for this approach was the information already available regarding the bash command analysis. The first attempt was to find timestamp strings that are prefixed with a hashtag. Zsh does not however seem to store timestamps in the same way as bash. The next step was to search for issued commands somewhere in chunks using the *heapsearch* plugin. Listing 22 shows a result excerpt for searching the string `#$@%&*()` with *heapsearch*, revealing a chunk at address 0x09BCF830 which contains the command of interest.

While commands could sometimes be identified in more than one chunk, each command seemed to be at least in one allocated chunk that only holds the command and some trailing bytes at the end. Listing 23 shows a hexdump of a chunk containing an issued zsh command.

As those chunks did not offer any meta information (e.g. at which time the command was issued), the next step was to find pointers to those chunks by again using the *heapsearch* plugin, but this time providing the address of the chunk that contains the issued command. It was possible to find exactly one pointer for each tested command in a separate allocated chunk with a size of 56 byte (for the x86 environment).

After examining multiple of those chunks (using the dumped content from the *heapdump* plugin), the following information could be derived:

- Bytes 5-8 contain a pointer to the issued command.
- Bytes 25-32 are 2 timestamps, stored as four byte integers. The first four byte are the start time at which the command has been issued and the last four bytes are the time when the command ended.
- Bytes 41-44 contain the command counter.

When now examining those chunks for references using the *heaprefs* plugin (see Listing 24; the output has been stripped and modified for this work), it shows that bytes 1-4, 5-8 (the command pointer), 13-16, 17-20 and 33-36 point to other chunks. The start addresses of those chunks are listed in the *Comment* column, while the bytes containing the pointers are marked with square brackets in the *Data* column.

By combining those insights with zsh's source code, the relevant history entry struct *histent* reveals two of those pointers: the fields *down* (bytes 17-20) and *up* (bytes 13-16). Those fields are used to reference the previous/next *histent* entry and hence allow a reliable traversal of *histent* instances. As the linked list of *histent* entries is circular, just walking one direction is sufficient to get all *histent* entries. The other pointers are not important for the current examination.

The last task at this point was to build a plugin, that automatically extracts those command information. To be able to traverse the *histent* list, the first step is to reliably identify one *histent* entry. As commands can be contained in chunks of various sizes and do not offer any searchable pattern, the approach is to find chunks containing the *histent* struct. The containing chunk's size is 56 byte for x86 and 96 byte for x64 architectures (the size results from the struct's size plus the bytes required to get an aligned chunk size; see Section II-C2). Because there are also non relevant chunks with the same size, they need to be distinguished. As each *histent* entry should have a pointer to a chunk containing the command and a pointer to the next and previous *histent* entry, the test consists of checking whether or not those pointers reference an already known chunk. If the test result is positive, the last check is to walk the *up* and *down* pointers to the next and previous *histent* struct and test if their *down*/*up* member points to the current chunk. If this is the case too, the current chunk is treated as a *histent* struct and the command history is walked using the *down* member.

Listing 25 shows an example output of the zsh plugin (the output has been stripped).

```
Examining chunk at offset 0x9C22938, belonging to the given address(es): 0x9c22938

                 Data                                                  Comment
--------------------------------------------      -------------------------------------------------------
[b046c009] [38f8bc09] 02000000 [7081c209]      Chunk pointer(s): 0x9c046a8, 0x9bcf830, 0x9c28168
[c042c109]  00000000  65b27658  65b27658       Chunk pointer(s): 0x9c142b8
[a0d4c009]  01000000  e7060000  00000000       Chunk pointer(s): 0x9c0d498
 38000000   31000000  01000000  98510000
```

Listing 24: Analyzing a chunk for references with heapref

```
PID     #          Started                  Ended            Command
---    ---  ---------------------- ---------------------- ------------
277      1   2016-03-31 23:51:09Z  2016-03-31 23:51:09Z  'ps aux'
...
277    142   2016-08-31 11:55:43Z  2016-08-31 11:55:43Z  '#!$@%&*()'
```

Listing 25: Example output for the zsh plugin

While it was possible to reconstruct the *bash* history with a raw search (because of the timestamp string), this approach would not have worked for the *zsh*, as the timestamp is not saved as a string with an additional hashtag but only as a four byte integer. Because, in addition, no further searchable patterns could be identified during the analysis, a raw search is most probably not applicable in this context and hence shows the advantage of using the heap analysis plugins.

B. KeePassX

The second tool examined was the password manager *KeePassX* (version 0.4.3) and has been tested in the following environments:

- Ubuntu 15.10 32 bit, x86, Kernel Version 4.2.0-16-generic, Glibc Version 2.21
- Ubuntu 15.10 64 bit, x64, Kernel Version 4.2.0-16-generic, Glibc Version 2.21

The setup for the following analysis consisted of a *KeePassX* database, which contained several password entries that have been separated in two folders. Each password entry had a value for *Title*, *Username*, *URL* and *Comment*. When the database has been opened for analysis purposes, only the first folder was opened while leaving the second folder completely untouched.

Our first attempt was to find the unencrypted master password and the passwords of entries somewhere in the process space, but they could not be found. The unhidden password for a currently open password entry however, has been successfully observed in three allocated chunks during 5 tests with different password manager entries. The three chunks persist as long as the password entry shows the unhidden password. If the password is hidden again, two of the three chunks are freed but one stays allocated. Only if the entry window is closed, all chunks containing the password are freed. Depending on the size of the freed chunk, the password is overwritten within milliseconds up to a few minutes or even hours by a new allocation. While freed chunks, containing passwords of a length range from 1 to at least 40, are normally reallocated within a few seconds or minutes, there have been instances where a freed chunk containing a password of that size has been consolidated in a bigger freed chunk. As some bigger chunk sizes are not allocated that often (e.g. in the range of a few hundred bytes), the password might remain for probably a few hours in this chunk, but is also harder to find (it is surrounded by other data). Furthermore, the actual password has never been observed in the first 18 bytes of the chunks data part (see also the following analysis), which means that even in a case where the freed chunk is placed in a large bin, the password is not overwritten by any bin pointers on an x86 architecture.

After the password field, the next step was to look for further fields of interest. The fields selected for this analysis were *Title*, *Username*, *URL* and *Comment*. *KeePassX* stores the full field content in allocated chunks right after the database has been opened. This is not only true for fields like the title, URL and comment, but also the username field, which in the case of *KeePassX* is shown only in asterisks in the overview and hence should not be needed unencrypted within the heap at that moment. To sum up: If a password database is opened and not locked, all fields from the overview (except the password field) from all password manager entries in all folders can be extracted from the heap. In order to analyze and compare the data from different chunks (containing the field strings), the *heapdump* plugin has been used to dump them in separate files. The following Listing 26 shows the hex dump output of a dumped chunk, containing a username (in this case `yyyyyyyy_user5_AAAAAAAAB`).

After comparing various chunks containing strings from the same type (such as usernames) and strings from other types, the following properties can be derived (which are also true for the unhidden password):

- The string is always 16-bit little endian encoded.
- The string does not start at the beginning of the chunk's data part, but exactly after byte 18. Most probably because the string is part of a struct/object.
- Bytes 5-8 and 9-12, respectively, correlate with the string's size, while bytes 9-12 state the correct size (the size is the number of characters represented by the encoded byte sequence, not the number of bytes). Both are probably instances of a four byte unsigned integer. The value from bytes 4-8 has been exactly by one larger than the value from bytes 9-12 (see also Listing 26: 0x19 vs. 0x18).
- Bytes 13-16 point to the beginning of the string.
- The string is followed by 4 null bytes.
- Depending on the size of the string, there were additional bytes at the end (kind of padding bytes), ranging in the most cases from zero till 6 byte. There have been however seldom cases, in which this number went up to 14 bytes (6 plus the amount of bytes until the next higher chunk size).

Bytes 1-4 did not change and while bytes 13-18 and the bytes after the string at the end changed a lot, they did not show any reliable coherence to a certain type or password manager entry.

The next step was to search for any pointers to a field string using the *heapsearch* plugin. While a search for the string's start address did not reveal any references (except for the one contained in the same chunk), searching for the beginning of the data part of that chunk revealed at least one pointer in another chunk. When analyzing this chunk with the *heaprefs* plugin, it reveals 12 pointers to other chunks. Following those pointers shows that four of them point to the chunks containing the *Title*, *Username*, *URL* and *Comment* strings. After analyzing more password entries, it was possible to make the fair assumption that for each password entry, there is a chunk of size 96 byte (in the x86 environment) that references at least those four fields. That means, by searching for chunks of the same size and examining the pointers at the given offsets, it is possible to gather the *Title*, *Username*, *URL* and *Comment* string of the same password entry. This information was used to create a proof of concept plugin by using the *HeapAnalysis* class, which automatically extracts these four fields for all password entries. Listing 27 shows an example output of that plugin (the output has been stripped, especially regarding the strings to fit in one line).

It should be mentioned that without the information about the start address from the chunk context, finding references to the field strings would have been more difficult, while in the worst case preventing the possibility to correlate the various strings to one password entry.

C. Wget

The tests in this section have been done in the following environments:

- Arch Linux 32 bit, Kernel Version 4.4.5-ARCH, Glibc Version 2.23, Wget Version 1.17.1
- Arch Linux 32 bit, Kernel Version 4.4.5-ARCH, Glibc Version 2.23, Wget Version 1.18.1
- Arch Linux 64 bit, Kernel Version 4.4.5-ARCH, Glibc Version 2.23, Wget Version 1.17.1
- Ubuntu 15.10 32 bit, Kernel Version 4.2.0-16-generic, Glibc Version 2.21, Wget Version 1.16.1

Wget is a utility to download resources from a specified URL. It can be used for that purpose by simply calling it with an URL, which will immediately result in an attempt to download the desired resource and store it in a local file for further usage. As online resources can be protected by authentication mechanisms, it offers options to provide e.g. username and password information. The fact that it exists on nearly every Linux instance makes it interesting for attackers who have compromised a Linux system and e.g. want to download additional malware for their attack.

In the case of an incident, where attackers downloaded certain files from external resources using *Wget* and erased those traces afterwards from the file system, an investigator might want try to download those resources for analysis purposes. If the URLs are known, but the access is protected by authentication, the need for credentials arises. When the HTTP Basic Authentication (Franks, 1999) is used, extracting those information is fairly easy. It can e.g. be established by simply searching over the whole memory dump for the command line call which immediately reveals the username and password as shown in Listing 28.

Even when the password is not given on the command line, it can simply be decoded from the Base64 encoded authentication string contained within the HTTP

```
0100 0000 1900 0000 1800 0000 7258 4b09 ............rXK.
e0ff 7900 7900 7900 7900 7900 7900 7900 ..y.y.y.y.y.y.y.
7900 5f00 7500 7300 6500 7200 3500 5f00 y._.u.s.e.r.5._.
4100 4100 4100 4100 4100 4100 4100 4100 A.A.A.A.A.A.A.A.
4200 0000 0000 ffff ffff ffff         B...........
```

Listing 26: Hex dump of a chunk's data part, containing a KeePassX username field

```
Entry    Title       URL      Username   Comment
------ ---------- -------- --------- ------------
1      y_title1_A y_url1_A y_user1_A y_comment1_A
2      y_title2_A y_url2_A y_user2_A y_comment2_A
...
```

Listing 27: Example output for the KeePassX plugin

```
# strings memory_dump.raw | grep wget | grep http
wget --http-user=root --http-password=S3cret http://172.16.239.1:8000/malware.exe
...
```

Listing 28: Extracting credentials from command line calls

Authorization header (Franks, 1999). Listing 29 shows an example extraction.

If however, an authentication mechanism like HTTP Digest authentication (Franks, 1999) is used, and the password is not provided on the command line, the necessary password might still be in the memory dump but cannot simply be spotted. A replay attack, using the HTTP Authorization header retrieved from the memory dump does not normally work either, as the server sends on each authentication request a new nonce, which is incorporated in the digest generation and hence leads each time to different authentication material. It is however possible to reliably retrieve the password from a certain allocated chunk within the heap.

While the amount of allocated chunks ranges from about 86 for *Wget* versions 1.16.1 and 1.17.1 up to 2445 for version 1.18.1, the amount of chunks for the relevant sizes containing the password ranges only from one up to four. As in all 21 test runs (using the same or differing passwords), all non-relevant chunks of the same size contained only scattered printable characters (see Listing 30 for an example), it should be most of the time pretty easy to find the correct one.

The password was, in all 21 test runs located at the beginning of the chunk's data part, followed by some null bytes and arbitrary further content (in some cases only null bytes, in other cases strings). An example output can be seen in Listing 31. The password in this case is `asdfghjklZXasdfghjklZX`.

The relevant chunk sizes differ mainly in relation to the password length. The following list summarizes the results.

- In all tested environments with a password length smaller or equal to 118, the relevant chunk size was 128.
- In all tested environments except for the 64 bit Arch Linux with a password length greater than 118, the relevant chunk size was 248.
- In the 64 bit Arch Linux environment with a password length greater than 118, the relevant chunk size was 256.

As can be seen in Listing 31, there is no leading pattern before the password, which would be easily recognizable in a memory dump while performing a raw search (without the chunk details). Also the content after the password within the same chunk does not offer a reliable search pattern, as this content changed multiple times. A reliable raw memory search would hence only be possible, if the string to search (in this case the password) would be known up front. As this is a rather theoretical scenario, this analysis shows the advantage of using the introduced plugins. Instead of relying on a searchable pattern or the content of the data, it is possible to identify the password by simply focusing on a chunk size.

It should be noted that *Wget* processes normally only live for a few seconds and their data might in most cases already be overwritten by other processes. But if the process data is still available or the *Wget* call e.g. did not reach the endpoint yet and hence is still running (the timeouts last in many cases quite long), the password might be extractable.

```
# strings memory_dump.raw | grep 'Authorization: Basic'
Authorization: Basic cm9vdDpTM2NyZXQ=
```

Listing 29: Extracting credentials from the HTTP Authorization Header

```
0000000: 40c0 f501 0000 0000 b034 d57a fd7e 0000 @........4.z.~..
0000010: 4d00 0000 0000 0000 0200 0000 0000 0000 M...............
0000020: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000030: ffff ffff 0000 0000 0700 0000 0000 0000 ................
0000040: d434 d57a fd7e 0000 e334 d57a fd7e 0000 .4.z.~...4.z.~..
0000050: e434 d57a fd7e 0000 e834 d57a fd7e 0000 .4.z.~...4.z.~..
0000060: ed34 d57a fd7e 0000 f334 d57a fd7e 0000 .4.z.~...4.z.~..
0000070: f734 d57a fd7e 0000          .4.z.~..
```

Listing 30: Hex dump of a chunk with size 128, containing no password

```
0000000: 6173 6466 6768 6a6b 6c5a 5861 7364 6667 asdfghjklZXasdfg
0000010: 686a 6b6c 5a58 0000 0000 0000 0000 0000 hjklZX..........
0000020: 7563 006b 6f5f 4b52 2e65 7563 4b52 006b uc.ko_KR.eucKR.k
0000030: 6f5f 4b52 006b 6f5f 4b52 2e65 7563 4b52 o_KR.ko_KR.eucKR
0000040: 006c 6974 6875 616e 6961 6e00 6c74 5f4c .lithuanian.lt_L
0000050: 542e 4953 4f2d 3838 3539 2d31 3300 6e6f T.ISO-8859-13.no
0000060: 5f4e 4f00 6e62 5f4e 4f2e 4953 4f2d 3838 _NO.nb_NO.ISO-88
0000070: 3539 2d31 006e 6f5f          59-1.no_
```

Listing 31: Hex dump of a chunk with size 128, containing the password

D. Building a Plugin

This section shows how a new plugin can be created using the *HeapAnalysis* class (see Section III-G on page 32). The current example creates a plugin to extract field information for password entries within the password manager tool *KeePassX* version 0.4.3, based on the results from Section V-B on page 38. These fields include the title, username, URL and comment string and are arranged according to the password entry they belong to. It should be noted, that this plugin has only been tested in the environment described in Section V-B on page 38.

The plugin works in essence as follows. It retrieves all allocated chunks and searches for the ones containing the references to field strings. If it finds one, it gathers those field strings and prints them. The following explanations refer to Listing 32 and describe the plugin in more detail.

Lines 2 and 4 The new plugin imports the heap analysis module and extends the *HeapAnalysis* class.

Line 9 The name of the new plugin, which is necessary to be able to call it from the command line.

Line 11 The framework's function to render the output, which is the starting point of this plugin.

Lines 12 and 13 For each given task, the *HeapAnalysis* instance is initialized.

Lines 15, 17 and 20 This dictionary is filled with all allocated chunks, which can be accessed by their data offset. This eases the access to chunks containing field strings later on.

Line 19 As the field strings are contained in allocated chunks, this plugin searches only in those chunks (including main arena, thread arena and MMAPPED chunks), by using the public function `get_all_allocated_chunks` from the *HeapAnalysis* class.

Lines 23 - 27 Defines the columns of this plugin's output.

Line 29 The offset where the extractable strings start, within the chunk's data part (see also Section V-B on page 38).

Line 30, 54 and 55 Responsible for printing each password entry.

Lines 32, 35 and 36 The first step is to find the chunks containing pointers to the field strings. Those chunks have typically a size of 96 byte.

Line 38 Retrieves the data part of the current chunk.

Line 40 and 52 Holds the title, username, URL and comment string for the current password entry, by using the Python method `repr` . The advantage of using `repr` is that no data is hidden from the user's eyes (if e.g. null bytes are at the beginning of the string, they would normally not be recognizable).

Line 34, 42, 43, 45, 56 and 57 Each chunk with a size of 96 byte is examined for potential chunk pointers at the offsets mentioned in Section V-B on page 38. If there is a chunk for each pointer, this the current chunk is considered as a password entry. If one of the pointers does not belong to a known chunk, the next chunk with a size of 96 byte is examined.

Line 47 and 48 Calculates the size of the current field's string.

Line 50 and 51 The currend field string is extracted and decoded.

Listing 33 shows its ouput, which has been stripped to fit in one line (marked with three dots before and after a string). Entry 16 shows a complete output, but also an entry that is not a password entry created for the test scenario. The same goes with the entries 8, 9 and 11. These have also been observed after creating a new database and adding only one entry. The assumption is that they serve internal purposes and are held in all *KeePassX* processes, while probably using the same or a similar struct as password entries (which is why they are appearing in the output).

The `u` in the beginning and the surrounding single quotes within Listing 33 are the result from using the *repr* function (as mentioned earlier) and are around every string (only in this case stripped for most of the strings). The advantage of using *repr* is that no data is hidden from the user's eyes (if e.g. non-printable characters are part of the string), as can be seen in entry 16 for the URL string.

```python
import struct
from rekall.plugins.linux import heap_analysis

class Keepassx(heap_analysis.HeapAnalysis):
   """Gathers password entries for keepassx.
   The retrieved content of those entries comprises the username, title, URL and Comment.
   """

   __name = "keepassx"

   def render(self, renderer):
      for task in self.filter_processes():
         if self.init_for_task(task):

            chunks_dict = dict()

            data_offset = self._libc_profile.get_obj_offset("malloc_chunk", "fd")

            for chunk in self.get_all_allocated_chunks():
               chunks_dict[chunk.v() + data_offset] = chunk


            renderer.table_header([("Entry", "entry", "6"),
                                 ("Title", "title", "25"),
                                 ("URL", "url", "25"),
                                 ("Username", "username", "25"),
                                 ("Comment", "comment", "44")])

            string_offset = 18
            entry_number = 1

            for chunk in chunks_dict.values():

               try:
                  if not chunk.chunksize() == 96:
                     continue

                  p_entry_data = chunk.to_string()

                  field_strings = []

                  for i in [12, 16, 20, 36]:
                     pointer = struct.unpack("I", p_entry_data[i:i+4])[0]

                     curr_chunk_data = chunks_dict[pointer].to_string()

                     string_size = struct.unpack("I", curr_chunk_data[8:12])[0]
                     string_size *= 2

                     curr_string = curr_chunk_data[string_offset:string_offset+string_size]
                     curr_string = curr_string.decode("utf-16-le")
                     field_strings.append(repr(curr_string))

                  renderer.table_row(entry_number, *field_strings)
                  entry_number += 1
               except (KeyError, UnicodeDecodeError):
                  pass
```

Listing 32: Example KeePassX Extractor Plugin

```
Entry         Title                 URL              Username              Comment
-----  -------------------- ----------------- ------------------- ------------------------
1       ...yy_title3_AA...   ...yy_url3_AA...   ...yy_user3_AA...   ...yy_comment3_AA...
2       ...yy_title4_AA...   ...yy_url4_AA...   ...yy_user4_AA...   ...yy_comment4_AA...
3       ...yy_title5_AA...   ...yy_url5_AA...   ...yy_user5_AA...   ...yy_comment5_AA...
4       ...yy_title10_AA...  ...yy_url10_AA...  ...yy_user10_AA...  ...yy_comment10_AA...
5       ...yy_title11_AA...  ...yy_url11_AA...  ...yy_user11_AA...  ...yy_comment11_AA...
6       ...yy_title13_AA...  ...yy_url13_AA...  ...yy_user13_AA...  ...yy_comment13_AA...
7       ...yy_title1_AA...   ...yy_url1_AA...   ...yy_user1_AA...   ...yy_comment1_AA...
8          u''                 u''                 u''                 u''
9          u''                 u''                 u''                 u''
10      ...yy_title9_AA...   ...yy_url9_AA...   ...yy_user9_AA...   ...yy_comment9_AA...
11         u''                 u''                 u''                 u''
12      ...yy_title14_AA...  ...yy_url14_AA...  ...yy_user14_AA...  ...yy_comment14_AA...
13      ...yy_title7_AA...   ...yy_url7_AA...   ...yy_user7_AA...   ...yy_comment7_AA...
14      ...yy_title12_AA...  ...yy_url12_AA...  ...yy_user12_AA...  ...yy_comment12_AA...
15      ...yy_title2_AA...   ...yy_url2_AA...   ...yy_user2_AA...   ...yy_comment2_AA...
16         u'q'              u'/\x00\u0814 \x00'   u''                 u''
17      ...yy_title6_AA...   ...yy_url6_AA...   ...yy_user6_AA...   ...yy_comment6_AA...
18      ...yy_title8_AA...   ...yy_url8_AA...   ...yy_user8_AA...   ...yy_comment8_AA...
```

Listing 33: Example KeePassX Extractor Plugin Output

VI. Conclusion and Future Work

This section summarizes this work, highlights some limitations, gives a prospect on future work and concludes the results.

A. Summary

This paper focuses on analyzing the heap in the context of Linux processes with the research objective to support an investigator in analyzing data contained in user space processes. First, an in-depth understanding of Glibc's heap implementation was established, which is documented in Section II. The analysis focused on how and where heap related data is stored from a memory forensics perspective. Second, this knowledge has been used to build plugins for the memory forensics framework Rekall (Google Inc, 2016c), which analyze the heap of a Linux user space process and offer access to the identified chunks. The implementation details are documented in Section III and describe in particular an algorithm to identify hidden MMAPPED chunks. As producing reliable results is a crucial requirement in computer forensics, Section IV covers information that enable the verification of the gathered results. To illustrate the usefulness of our implementation, Section V describes the black box analysis process of userspace applications, using the example of *zsh* and *KeePassX*.

B. Limitations

As already explained by Cohen (2015) for pagefiles, swap space is in some scenarios a crucial resource during the user space process analysis, which is not addressed in our work so far. When the plugins introduced in this work are used on a process with swapped out pages, containing heap related data, they will most probably fail in reliably analyzing the heap and extracting all chunks.

One of the goals of this work was to serve a process-like view on data contained in the memory. While this has been accomplished to the maximum extent of details that the heap is offering in this context (the location of a specific information and its size), it was not possible to extract information about the type of data. The reason is that the heap does not store any data type information. One way to still correlate a certain chunk with a specific type exists in a scenario where the data type and the size of the data itself is known up front. By searching for chunks of that size, the investigator might be able to gather data of a specific type. This approach requires however further tests on those chunks (as shown in Section V-A), as there might be more chunks with the same size containing different data.

As stated in Section II-A, the usage of a certain heap implementation is not bound to a specific operating system. In cases where a different heap implementation is used, the findings and plugins from this work will most probably not be applicable, but instead need to be performed analogue for those implementations.

To this date, our *HeapAnalysis* class and the plugins only support the analysis of Linux processes on the mentioned architectures and Glibc versions. The information provided in this work and the technical report can however be used to add support for further architectures or operating systems.

While it is possible to analyze the heap of a user space process without supplying debug information, it is not reliable in all cases. Beside the fact that the plugins will most certainly fail in analyzing the heap when any of the relevant structs has changed (*malloc_chunk*, *malloc_state* or *heap_info*), the results might still be incomplete if the pointer to the global variable *mp_* is missing. Without *mp_* it is not possible to reliably determine if all MMAPPED chunks have been discovered and as the search for hidden MMAPPED chunks is not initiated in this case, there might be at least one MMAPPED chunk missing in the output.

Besides hidden MMAPPED chunks, there is one further scenario, in which the *HeapAnalysis* class might miss chunks: If the analysis process mistakenly left out a *vm_area_struct* containing a whole arena, the results seem still to be correct but miss the arena's chunks. This case might happen in a scenario, where no debug information for the main arena's location have been provided and the main arena search process is not able to find it, which also means that this process was unable to find any thread arena. As no valid pointer to any arena is available in that scenario, there might be an undetected arena and hence unnoticed chunks. While this case is theoretically possible, the implemented functionality to detect such scenarios did not miss any arena during our evaluation.

C. Future Work

The limitation considered most important is the missing support for swap space. In order to fully analyze the heap of a process, the access to all pages containing heap related data must be ensured. The acquisition and integration of swap space into the memory forensics process is hence considered a fundamental step for further user space process analysis.

To increase the support of the *HeapAnalysis* class, the plugins could be tested on further architectures and adjusted appropriately if required. An analysis of further heap implementations such as *jemalloc* would in addition, allow to analyze the heap allocations of processes from applications such as Firefox. Furthermore, when conducting an appropriate analysis in the context of FreeBSD processes, it would also include another operating system.

D. Conclusion

The plugins introduced in this paper simplify the analysis process and enable the identification of information in memory that cannot easily be found using a pattern matching approach. These plugins and the documented details about heap objects in memory can also be used to support further research in the field of memory forensics and help forensic investigators to clarify an incident or crime. Furthermore, this paper demonstrated the analysis of user space processes while illustrating the advantage of having heap details during the analysis process.

VII. Appendix

A. Documentation in comments

```
/*
 This is a version (aka ptmalloc2) of malloc/free/realloc written by
 Doug Lea and adapted to multiple threads/arenas by Wolfram Gloger.

 There have been substantial changes made after the integration into
 glibc in all parts of the code. Do not look for much commonality
 with the ptmalloc2 version.

* Version ptmalloc2-20011215
 based on:
 VERSION 2.7.0 Sun Mar 11 14:14:06 2001 Doug Lea (dl at gee)

* Quickstart

 In order to compile this implementation, a Makefile is provided with
 the ptmalloc2 distribution, which has pre-defined targets for some
 popular systems (e.g. "make posix" for Posix threads). All that is
 typically required with regard to compiler flags is the selection of
 the thread package via defining one out of USE_PTHREADS, USE_THR or
 USE_SPROC. Check the thread-m.h file for what effects this has.
 Many/most systems will additionally require USE_TSD_DATA_HACK to be
 defined, so this is the default for "make posix".

* Why use this malloc?

 This is not the fastest, most space-conserving, most portable, or
 most tunable malloc ever written. However it is among the fastest
 while also being among the most space-conserving, portable and tunable.
 Consistent balance across these factors results in a good general-purpose
 allocator for malloc-intensive programs.

 The main properties of the algorithms are:
 * For large (>= 512 bytes) requests, it is a pure best-fit allocator,
   with ties normally decided via FIFO (i.e. least recently used).
 * For small (<= 64 bytes by default) requests, it is a caching
   allocator, that maintains pools of quickly recycled chunks.
 * In between, and for combinations of large and small requests, it does
   the best it can trying to meet both goals at once.
 * For very large requests (>= 128KB by default), it relies on system
   memory mapping facilities, if supported.

 For a longer but slightly out of date high-level description, see
    http://gee.cs.oswego.edu/dl/html/malloc.html

 You may already by default be using a C library containing a malloc
 that is based on some version of this malloc (for example in
 linux). You might still want to use the one in this file in order to
 customize settings or to avoid overheads associated with library
 versions.

* Contents, described in more detail in "description of public routines" below.

 Standard (ANSI/SVID/...) functions:
   malloc(size_t n);
   calloc(size_t n_elements, size_t element_size);
   free(void* p);
   realloc(void* p, size_t n);
   memalign(size_t alignment, size_t n);
   valloc(size_t n);
   mallinfo()
   mallopt(int parameter_number, int parameter_value)

 Additional functions:
   independent_calloc(size_t n_elements, size_t size, void* chunks[]);
   independent_comalloc(size_t n_elements, size_t sizes[], void* chunks[]);
   pvalloc(size_t n);
   cfree(void* p);
   malloc_trim(size_t pad);
   malloc_usable_size(void* p);
   malloc_stats();
```

```

* Vital statistics:

  Supported pointer representation: 4 or 8 bytes
  Supported size_t representation: 4 or 8 bytes
      Note that size_t is allowed to be 4 bytes even if pointers are 8.
      You can adjust this by defining INTERNAL_SIZE_T

  Alignment:                    2 * sizeof(size_t) (default)
      (i.e., 8 byte alignment with 4byte size_t). This suffices for
      nearly all current machines and C compilers. However, you can
      define MALLOC_ALIGNMENT to be wider than this if necessary.

  Minimum overhead per allocated chunk: 4 or 8 bytes
      Each malloced chunk has a hidden word of overhead holding size
      and status information.

  Minimum allocated size: 4-byte ptrs: 16 bytes (including 4 overhead)
                    8-byte ptrs: 24/32 bytes (including, 4/8 overhead)

      When a chunk is freed, 12 (for 4byte ptrs) or 20 (for 8 byte
      ptrs but 4 byte size) or 24 (for 8/8) additional bytes are
      needed; 4 (8) for a trailing size field and 8 (16) bytes for
      free list pointers. Thus, the minimum allocatable size is
      16/24/32 bytes.

      Even a request for zero bytes (i.e., malloc(0)) returns a
      pointer to something of the minimum allocatable size.

      The maximum overhead wastage (i.e., number of extra bytes
      allocated than were requested in malloc) is less than or equal
      to the minimum size, except for requests >= mmap_threshold that
      are serviced via mmap(), where the worst case wastage is 2 *
      sizeof(size_t) bytes plus the remainder from a system page (the
      minimal mmap unit); typically 4096 or 8192 bytes.

  Maximum allocated size: 4-byte size_t: 2^32 minus about two pages
                    8-byte size_t: 2^64 minus about two pages

      It is assumed that (possibly signed) size_t values suffice to
      represent chunk sizes. `Possibly signed' is due to the fact
      that `size_t' may be defined on a system as either a signed or
      an unsigned type. The ISO C standard says that it must be
      unsigned, but a few systems are known not to adhere to this.
      Additionally, even when size_t is unsigned, sbrk (which is by
      default used to obtain memory from system) accepts signed
      arguments, and may not be able to handle size_t-wide arguments
      with negative sign bit. Generally, values that would
      appear as negative after accounting for overhead and alignment
      are supported only via mmap(), which does not have this
      limitation.

      Requests for sizes outside the allowed range will perform an optional
      failure action and then return null. (Requests may also
      also fail because a system is out of memory.)

  Thread-safety: thread-safe

  Compliance: I believe it is compliant with the 1997 Single Unix Specification
      Also SVID/XPG, ANSI C, and probably others as well.

* Synopsis of compile-time options:

    People have reported using previous versions of this malloc on all
    versions of Unix, sometimes by tweaking some of the defines
    below. It has been tested most extensively on Solaris and Linux.
    People also report using it in stand-alone embedded systems.

    The implementation is in straight, hand-tuned ANSI C. It is not
    at all modular. (Sorry!) It uses a lot of macros. To be at all
    usable, this code should be compiled using an optimizing compiler
    (for example gcc -O3) that can simplify expressions and control
    paths. (FAQ: some macros import variables as arguments rather than
    declare locals because people reported that some debuggers
    otherwise get confused.)

    OPTION               DEFAULT VALUE

    Compilation Environment options:

    HAVE_MREMAP          0

    Changing default word sizes:

    INTERNAL_SIZE_T      size_t
    MALLOC_ALIGNMENT     MAX (2 * sizeof(INTERNAL_SIZE_T),
                              __alignof__ (long double))

    Configuration and functionality options:

    USE_PUBLIC_MALLOC_WRAPPERS NOT defined
    USE_MALLOC_LOCK      NOT defined
    MALLOC_DEBUG         NOT defined
    REALLOC_ZERO_BYTES_FREES 1
    TRIM_FASTBINS        0

    Options for customizing MORECORE:

    MORECORE             sbrk
    MORECORE_FAILURE     -1
    MORECORE_CONTIGUOUS  1
    MORECORE_CANNOT_TRIM NOT defined
    MORECORE_CLEARS      1
    MMAP_AS_MORECORE_SIZE (1024 * 1024)

    Tuning options that are also dynamically changeable via mallopt:

    DEFAULT_MXFAST       64 (for 32bit), 128 (for 64bit)
    DEFAULT_TRIM_THRESHOLD 128 * 1024
    DEFAULT_TOP_PAD      0
    DEFAULT_MMAP_THRESHOLD 128 * 1024
    DEFAULT_MMAP_MAX     65536

    There are several other #defined constants and macros that you
    probably don't want to touch unless you are extending or adapting malloc. */

/*
  void* is the pointer type that malloc should say it returns
*/
```

Listing 34: Glibc 2.23(malloc/malloc.c): Documentation at the beginning

```
/*
   malloc_chunk details:

    (The following includes lightly edited explanations by Colin Plumb.)

    Chunks of memory are maintained using a 'boundary tag' method as
    described in e.g., Knuth or Standish. (See the paper by Paul
    Wilson ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps for a
    survey of such techniques.) Sizes of free chunks are stored both
    in the front of each chunk and at the end. This makes
    consolidating fragmented chunks into bigger chunks very fast. The
    size fields also hold bits representing whether chunks are free or
    in use.

    An allocated chunk looks like this:


    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if allocated            | |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                       |M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
```

```
          .           (malloc_usable_size() bytes)          .
          .                                                 |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |           Size of chunk                         |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+


   Where "chunk" is the front of the chunk for the purpose of most of
   the malloc code, but "mem" is the pointer that is returned to the
   user. "Nextchunk" is the beginning of the next contiguous chunk.

   Chunks always begin on even word boundaries, so the mem portion
   (which is returned to the user) is also on an even word boundary, and
   thus at least double-word aligned.

   Free chunks are stored in circular doubly-linked lists, and look like this:

   chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |           Size of previous chunk                |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   `head:' |          Size of chunk, in bytes             |P|
     mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |           Forward pointer to next chunk in list |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |           Back pointer to previous chunk in list |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |           Unused space (may be 0 bytes long)    .
          .                                                 .
          .                                                 |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   `foot:' |          Size of chunk, in bytes               |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

   The P (PREV_INUSE) bit, stored in the unused low-order bit of the
   chunk size (which is always a multiple of two words), is an in-use
   bit for the *previous* chunk. If that bit is *clear*, then the
   word before the current chunk size contains the previous chunk
   size, and can be used to find the front of the previous chunk.
   The very first chunk allocated always has this bit set,
   preventing access to non-existent (or non-owned) memory. If
   prev_inuse is set for any given chunk, then you CANNOT determine
   the size of the previous chunk, and might even get a memory
   addressing fault when trying to do so.

   Note that the `foot' of the current chunk is actually represented
   as the prev_size of the NEXT chunk. This makes it easier to
   deal with alignments etc but can be very confusing when trying
   to extend or adapt this code.

   The two exceptions to all this are

    1. The special chunk `top' doesn't bother using the
       trailing size field since there is no next contiguous chunk
       that would have to index off it. After initialization, `top'
       is forced to always exist. If it would become less than
       MINSIZE bytes long, it is replenished.

    2. Chunks allocated via mmap, which have the second-lowest-order
       bit M (IS_MMAPPED) set in their size fields. Because they are
       allocated one-by-one, each must contain its own trailing size field.

*/

/*
  ---------- Size and alignment checks and conversions ----------
*/

/* conversion from malloc headers to user pointers, and back */
```

Listing 35: Glibc 2.23(malloc/malloc.c): Chunk documentation

B. Mmap threshold for mallopt

```
#define HEAP_MIN_SIZE (32 * 1024)
```

```
#ifndef HEAP_MAX_SIZE
# ifdef DEFAULT_MMAP_THRESHOLD_MAX
#  define HEAP_MAX_SIZE (2 * DEFAULT_MMAP_THRESHOLD_MAX)
# else
#  define HEAP_MAX_SIZE (1024 * 1024) /* must be a power of two */
# endif
#endif
```

Listing 36: Glibc 2.23(malloc/arena.c): HEAP_MAX_SIZE definition

```
    case M_MMAP_THRESHOLD:
      /* Forbid setting the threshold too high. */
      if ((unsigned long) value > HEAP_MAX_SIZE / 2)
        res = 0;
      else
        {
          LIBC_PROBE (memory_mallopt_mmap_threshold, 3, value,
                      mp_.mmap_threshold, mp_.no_dyn_threshold);
          mp_.mmap_threshold = value;
          mp_.no_dyn_threshold = 1;
        }
      break;
```

Listing 37: Glibc 2.23(malloc/malloc.c): Part of mallopt responsible for setting new mmap threshold

C. Bin_index Macros

```
#define NBINS             128
#define NSMALLBINS         64
#define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
#define SMALLBIN_CORRECTION (MALLOC_ALIGNMENT > 2 * SIZE_SZ)
#define MIN_LARGE_SIZE    ((NSMALLBINS - SMALLBIN_CORRECTION) * SMALLBIN_WIDTH)

#define in_smallbin_range(sz)  \
  ((unsigned long) (sz) < (unsigned long) MIN_LARGE_SIZE)

#define smallbin_index(sz) \
  ((SMALLBIN_WIDTH == 16 ? (((unsigned) (sz)) >> 4) : (((unsigned) (sz)) >> 3))\
   + SMALLBIN_CORRECTION)

#define largebin_index_32(sz)                                                \
  (((((unsigned long) (sz)) >> 6) <= 38) ?  56 + (((unsigned long) (sz)) >> 6) :\
   ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9) :\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
   126)

#define largebin_index_32_big(sz)                                            \
  (((((unsigned long) (sz)) >> 6) <= 45) ?  49 + (((unsigned long) (sz)) >> 6) :\
   ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9) :\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
   126)

// XXX It remains to be seen whether it is good to keep the widths of
// XXX the buckets the same or whether it should be scaled by a factor
// XXX of two as well.
#define largebin_index_64(sz)                                                \
  (((((unsigned long) (sz)) >> 6) <= 48) ?  48 + (((unsigned long) (sz)) >> 6) :\
   ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9) :\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
   126)

#define largebin_index(sz) \
  (SIZE_SZ == 8 ? largebin_index_64 (sz)                                     \
   : MALLOC_ALIGNMENT == 16 ? largebin_index_32_big (sz)                     \
   : largebin_index_32 (sz))

#define bin_index(sz) \
  ((in_smallbin_range (sz)) ? smallbin_index (sz) : largebin_index (sz))
```

Listing 38: Glibc 2.23(malloc/malloc.c): bin_index Macros

D. Bottom chunks

The source code for the ”Bottom chunks” program, referenced in Section II-E.

```
#include <pthread.h>
#include "library.c"



void* threadFunc(void* arg) {
      printf("Before malloc in thread \n");
      display_mallinfo();
      getchar();
      fillHeapWithChunks(1,0);
      printf("After first call to fillHeapWithChunks\n");
      display_mallinfo();
      getchar();
      fillHeapWithChunks(0,8);
      printf("After second call to fillHeapWithChunks\n");
      display_mallinfo();
      getchar();
      fillHeapWithChunks(0,16);
      printf("After third call to fillHeapWithChunks\n");
      display_mallinfo();
      getchar();
      fillHeapWithChunks(0,20);
      printf("After fourth call to fillHeapWithChunks\n");
      display_mallinfo();
      getchar();
      char* addr;
      addr = malloc(1000);
      printf("After additional allocation to create bottom chunks for fourth filled heap.\n");
      display_mallinfo();
      malloc_stats();
      getchar();
      printf("Right before Thread is going to die\n");
      getchar();
      printf("Thread is dead\n");
}

int main(int argc, char** argv) {
      pthread_t t1;
      void* s;
      int ret;
      char* addr;

      printf("This example creates the different states of bottom chunks:%d\n",getpid());
      addr = malloc(1000);
      strncpy(addr, "Main Arena bytes", malloc_usable_size(addr));
      display_mallinfo();
      printf("\nMallinfo, right before thread creation\n");
      getchar();


      ret = pthread_create(&t1, NULL, threadFunc, NULL);
      if(ret)
      {
            printf("Thread creation error\n");
            return -1;
      }
      ret = pthread_join(t1, &s);
      if(ret)
      {
            printf("Thread join error\n");
            return -1;
      }
      return 0;
}
```

Listing 39: Creates different bottom chunk scenarios

E. Bottom chunk contains data Program

```
#include <pthread.h>
#include <stdlib.h>
#include "library.c"

#define BASE 128
#if __WORDSIZE == 32
# define SUBTRACT 8
#else
# define SUBTRACT 16
#endif

int glibc_minor_version;


void* threadFunc(void* arg) {
   // fills heap but leaves some space for a last chunk
   fillHeapWithChunks(1, BASE, glibc_minor_version);

   char* addr;

   // creates the last chunk, which contains only H characters
   addr = malloc(BASE - SUBTRACT);
   memset(addr, 'H', malloc_usable_size(addr));

   // last chunk gets freed, while leaving the H's in memory
   free(addr);

   // creating another chunk but leaving enough space for the extractable part of the first bottom chunk
   addr = malloc(BASE - SUBTRACT * 2);

   // simply creates a large chunk, so the next heap_info and the bottom chunks are created
   addr = malloc(BASE * 100);

   printf("\nThere should be now one bottom chunk, containing data of the last freed chunk\n\n");

   display_mallinfo();
   getchar();
   printf("Right before Thread is going to die\n");
   getchar();
   printf("Thread is dead\n");
}

int main(int argc, char** argv) {
   pthread_t t1;
   void* s;
   int ret;
   char* addr;

   if (argc < 2){
      printf("This program expects the glibc minor version as first argument. E.g. 23\n");
      exit(1);
   }

   printf("This example creates a bottom chunk that contains data ('H' characters). PID:%d\n",getpid());

   glibc_minor_version = atoi(argv[1]);


   ret = pthread_create(&t1, NULL, threadFunc, NULL);
   if(ret)
   {
      printf("Thread creation error\n");
      return -1;
   }
   ret = pthread_join(t1, &s);
   if(ret)
   {
      printf("Thread join error\n");
      return -1;
   }
   return 0;
}
```

Listing 40: The Bottom chunk contains data Program code

F. Bottom chunk contains data

The first Listing 41 shows the output for the program listed in Section VII-E on the preceding page when using debug information, and the second output in Listing 42 shows it without supplying debug information. The *Bottom chunk contains data* program has been started with an argument of `23` (for the Glibc Version) and the program has been run until the string *There should be now one bottom chunk, containing data of the last freed chunk* was printed. Additionaly in this case, the content of the dumped bottom chunk, created with the *heapdump* plugin is shown in Listing 43 (printed with the command line tool *xxd*), to verify the expected four `H` characters are in the dumped file.

```
 PID  Arenas Heap I. Non MMAPPED chunks N.M. chunks size MMAPPED chunks MMAPPED size Freed chunks Freed size
------ -------- --------- -------------------- ------------------ ---------------- -------------- -------------- ------------
30365 2     2      21             1062440       0            0          0          0
```

Listing 41: *heapinfo* Output for *Bottom chunk contains data* with debug information

```
 PID  Arenas Heap I. Non MMAPPED chunks N.M. chunks size MMAPPED chunks MMAPPED size Freed chunks Freed size
------ -------- --------- -------------------- ------------------ ---------------- -------------- -------------- ------------
2016-09-24 19:57:57,578:WARNING:rekall.1:It seems like the debug information for the mp_ offset are missing. This means some checks/
      verifications can't be done.
2016-09-24 19:57:57,579:WARNING:rekall.1:As it seems like we don't have debug information for the main arena and/or we didn't find the libc
      filename in the vm_areas, we now try to retrieve the main_arena via some different techniques for pid 30365.
30365 2     2      21             1062440       0            0          0          0
```

Listing 42: *heapinfo* Output for *Bottom chunk contains data* without debug information

```
xxd 30365.bottom-chunk_offset-0xb6cfffe8_size-12_dumped-4.dmp

0000000: 4848 4848                          HHHH
```

Listing 43: *heapinfo* Output for *Bottom chunk contains data* without debug information

G. Bin Distribution

This Section shows on the one hand the source code for the *library* program in Listing 44, defining various structs and macros used during the research, and on the other hand the output of the *generate_bin_map* function for different scenarios (mentioned later) in the following Listings (see also Section II-C4 on page 11). This function uses the *bin_index* macro defined in the *library* program, which origins from the Glibc source code (see Section VII-C on page 50).

```c
#include <malloc.h>
#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include <unistd.h>


#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

struct malloc_chunk;

struct malloc_chunk {

  INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
  INTERNAL_SIZE_T size;     /* Size in bytes, including overhead. */

  struct malloc_chunk* fd; /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size. */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};


/* The corresponding word size */
#define SIZE_SZ          (sizeof(INTERNAL_SIZE_T))

//# define MALLOC_ALIGNMENT 16
# define MALLOC_ALIGNMENT (2 *SIZE_SZ)

/* The corresponding bit mask value */
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)

#define NBINS       128
#define NSMALLBINS   64
#define SMALLBIN_WIDTH MALLOC_ALIGNMENT
#define SMALLBIN_CORRECTION (MALLOC_ALIGNMENT > 2 * SIZE_SZ)
#define MIN_LARGE_SIZE ((NSMALLBINS - SMALLBIN_CORRECTION) * SMALLBIN_WIDTH)

#define FASTCHUNKS_BIT (1U)

#define MIN_CHUNK_SIZE (offsetof(struct malloc_chunk, fd_nextsize))
#define MINSIZE (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK))

#define REQUEST_OUT_OF_RANGE(req)                 \
  ((unsigned long) (req) >=                             \
    (unsigned long) (INTERNAL_SIZE_T) (-2 * MINSIZE))

#define request2size(req)                         \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE) ? \
   MINSIZE :                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)


#define checked_request2size(req, sz)             \
  if (REQUEST_OUT_OF_RANGE (req)) {               \
              (sz) = 0;                           \
              }                                   \
              else                                \
                (sz) = request2size (req);

#define OUT_OF_RANGE ((unsigned long) (INTERNAL_SIZE_T) (-2 * MINSIZE))


#define largebin_index_32(sz)                               \
  (((((unsigned long) (sz)) >> 6) <= 38) ? 56 + (((unsigned long) (sz)) >> 6) :\
   ((((unsigned long) (sz)) >> 9) <= 20) ? 91 + (((unsigned long) (sz)) >> 9) :\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
   126)

#define largebin_index_32_big(sz)                           \
  (((((unsigned long) (sz)) >> 6) <= 45) ? 49 + (((unsigned long) (sz)) >> 6) :\
   ((((unsigned long) (sz)) >> 9) <= 20) ? 91 + (((unsigned long) (sz)) >> 9) :\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
   126)

// XXX It remains to be seen whether it is good to keep the widths of
// XXX the buckets the same or whether it should be scaled by a factor
// XXX of two as well.
#define largebin_index_64(sz)                                    \
  (((((unsigned long) (sz)) >> 6) <= 48) ? 48 + (((unsigned long) (sz)) >> 6) :\
   ((((unsigned long) (sz)) >> 9) <= 20) ? 91 + (((unsigned long) (sz)) >> 9) :\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
   126)

#define largebin_index(sz) \
  (SIZE_SZ == 8 ? largebin_index_64 (sz)                        \
   : MALLOC_ALIGNMENT == 16 ? largebin_index_32_big (sz)   \
   : largebin_index_32 (sz))

#define bin_index(sz) \
  ((in_smallbin_range (sz)) ? smallbin_index (sz) : largebin_index (sz))

#define in_smallbin_range(sz) \
  ((unsigned long) (sz) < (unsigned long) MIN_LARGE_SIZE)

#define smallbin_index(sz) \
  ((SMALLBIN_WIDTH == 16 ? (((unsigned) (sz)) >> 4) : (((unsigned) (sz)) >> 3))\
   + SMALLBIN_CORRECTION)


static INTERNAL_SIZE_T global_max_fast;

#define DEFAULT_MXFAST (64 * SIZE_SZ / 4)

#define set_max_fast(s) \
  global_max_fast = (((s) == 0)                                   \
                  ? SMALLBIN_WIDTH : ((s + SIZE_SZ) & ~MALLOC_ALIGN_MASK))

#define get_max_fast() global_max_fast

#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)


#define MAX_FAST_SIZE (80 * SIZE_SZ / 4)

#define NFASTBINS (fastbin_index (request2size (MAX_FAST_SIZE)) + 1)

/* Check if m has acceptable alignment */

#define aligned_OK(m) (((unsigned long)(m) & MALLOC_ALIGN_MASK) == 0)

#define misaligned_chunk(p) \
  ((uintptr_t)(MALLOC_ALIGNMENT == 2 * SIZE_SZ ? (p) : chunk2mem (p)) \
   & MALLOC_ALIGN_MASK)


#ifndef DEFAULT_MMAP_THRESHOLD_MAX
  /* For 32-bit platforms we cannot increase the maximum mmap
     threshold much because it is also the minimum value for the
     maximum heap size and its alignment. Going above 512k (i.e., 1M
     for new heaps) wastes too much address space. */
# if __WORDSIZE == 32
#  define DEFAULT_MMAP_THRESHOLD_MAX (512 * 1024)
# else
#  define DEFAULT_MMAP_THRESHOLD_MAX (4 * 1024 * 1024 * sizeof(long))
# endif
#endif
```

```


#define HEAP_MIN_SIZE (32 * 1024)
#ifndef HEAP_MAX_SIZE
# ifdef DEFAULT_MMAP_THRESHOLD_MAX
#  define HEAP_MAX_SIZE (2 * DEFAULT_MMAP_THRESHOLD_MAX)
# else
#  define HEAP_MAX_SIZE (1024 * 1024) /* must be a power of two */
# endif
#endif


#if __WORDSIZE == 32
# define MALLOC_STATE_SIZE 1108
# define HEAP_INFO_SIZE 16
# define ATTACHED_THREAD_SIZE sizeof(int)
#else
# define MALLOC_STATE_SIZE 2192
# define HEAP_INFO_SIZE 32
# define ATTACHED_THREAD_SIZE sizeof(long)
#endif




static void
display_mallinfo(void)
{
   struct mallinfo mi;

   mi = mallinfo();

   printf("Total non-mmapped bytes (arena): %d\n", mi.arena);
   printf("# of free chunks (ordblks): %d\n", mi.ordblks);
   printf("# of free fastbin blocks (smblks): %d\n", mi.smblks);
   printf("# of mapped regions (hblks): %d\n", mi.hblks);
   printf("Bytes in mapped regions (hblkhd): %d\n", mi.hblkhd);
   printf("Max. total allocated space (usmblks): %d\n", mi.usmblks);
   printf("Free bytes held in fastbins (fsmblks): %d\n", mi.fsmblks);
   printf("Total allocated space (uordblks): %d\n", mi.uordblks);
   printf("Total free space (fordblks): %d\n", mi.fordblks);
   printf("Topmost releasable block (keepcost): %d\n", mi.keepcost);
}

void fillHeapWithChunks(int first_heap, int subtract, int minorVersion){
   int mmap_threshold = 128 * 1024;
   int allocation_size = mmap_threshold / 2;
   char* addr;

   int i;
   int allocated_size = 0;
   int real_chunk_size;

   // First heap contains not only a heap_info struct but also the malloc_state struct,
   // so we need to subtract both sizes
   // malloc_state has a size of 1108 for glibc version 2.23, and as it gets aligned,
   // we subtract 1112 bytes here; 16 bytes for heap_info
   if (first_heap == 1){
      int malloc_state_size = MALLOC_STATE_SIZE;
      if (minorVersion < 23){
         malloc_state_size = malloc_state_size - ATTACHED_THREAD_SIZE;
         //subtract = subtract - ATTACHED_THREAD_SIZE;
      }
      malloc_state_size = (malloc_state_size + MALLOC_ALIGN_MASK) &~ MALLOC_ALIGN_MASK;
      subtract = subtract + malloc_state_size;
   }

   subtract = subtract + HEAP_INFO_SIZE;

   // at last, we make room for the bottom chunks
   subtract = subtract + MINSIZE + MINSIZE/2;

   while (allocated_size < (HEAP_MAX_SIZE - allocation_size - subtract)){
      addr = (char*) malloc(allocation_size);
      real_chunk_size = *(((long*)addr)-1) & 0xFFFFFFF8;
      allocated_size = allocated_size + real_chunk_size;
      //memset(addr, 65+i, malloc_usable_size(addr));
   }

   int last_size = HEAP_MAX_SIZE - allocated_size - subtract;


   addr = (char*) malloc(last_size);
   memset(addr, 65, malloc_usable_size(addr));
}

void generate_bin_map(){
   int i;
   int bin_counter = 0;
   int last_size = 0;
   for ( i=16; i <= 50000000; i = i + 8){
      if (bin_index(i) > bin_counter){
         //printf("Index for size %d is %d and has a distance of %d from the last bin.\n", i,
             bin_index(i), i-last_size);
         if (last_size != 0){
            printf("and contains chunks with a size range of %d.\n",i-last_size);
         }
         printf("Index for size %d is %d ", i, bin_index(i));
         bin_counter = bin_index(i);
         last_size = i;
      }
   }
   printf("and contains chunks with a size larger than %d\n", last_size);
}
```

Listing 44: The Program Code for the Library, used during research

The first output in Listing 45 represents the bin distributions for a 32 bit architecture, when *MALLOC_ALIGNMENT* is 8 byte (which is typically the case).

```
Index for size 16 is 2 and contains chunks with a size range of 8.
Index for size 24 is 3 and contains chunks with a size range of 8.
Index for size 32 is 4 and contains chunks with a size range of 8.
Index for size 40 is 5 and contains chunks with a size range of 8.
Index for size 48 is 6 and contains chunks with a size range of 8.
Index for size 56 is 7 and contains chunks with a size range of 8.
Index for size 64 is 8 and contains chunks with a size range of 8.
Index for size 72 is 9 and contains chunks with a size range of 8.
Index for size 80 is 10 and contains chunks with a size range of 8.
Index for size 88 is 11 and contains chunks with a size range of 8.
Index for size 96 is 12 and contains chunks with a size range of 8.
Index for size 104 is 13 and contains chunks with a size range of 8.
Index for size 112 is 14 and contains chunks with a size range of 8.
Index for size 120 is 15 and contains chunks with a size range of 8.
Index for size 128 is 16 and contains chunks with a size range of 8.
Index for size 136 is 17 and contains chunks with a size range of 8.
Index for size 144 is 18 and contains chunks with a size range of 8.
Index for size 152 is 19 and contains chunks with a size range of 8.
Index for size 160 is 20 and contains chunks with a size range of 8.
Index for size 168 is 21 and contains chunks with a size range of 8.
Index for size 176 is 22 and contains chunks with a size range of 8.
Index for size 184 is 23 and contains chunks with a size range of 8.
Index for size 192 is 24 and contains chunks with a size range of 8.
Index for size 200 is 25 and contains chunks with a size range of 8.
Index for size 208 is 26 and contains chunks with a size range of 8.
Index for size 216 is 27 and contains chunks with a size range of 8.
Index for size 224 is 28 and contains chunks with a size range of 8.
Index for size 232 is 29 and contains chunks with a size range of 8.
Index for size 240 is 30 and contains chunks with a size range of 8.
Index for size 248 is 31 and contains chunks with a size range of 8.
Index for size 256 is 32 and contains chunks with a size range of 8.
Index for size 264 is 33 and contains chunks with a size range of 8.
Index for size 272 is 34 and contains chunks with a size range of 8.
Index for size 280 is 35 and contains chunks with a size range of 8.
Index for size 288 is 36 and contains chunks with a size range of 8.
Index for size 296 is 37 and contains chunks with a size range of 8.
Index for size 304 is 38 and contains chunks with a size range of 8.
Index for size 312 is 39 and contains chunks with a size range of 8.
Index for size 320 is 40 and contains chunks with a size range of 8.
Index for size 328 is 41 and contains chunks with a size range of 8.
Index for size 336 is 42 and contains chunks with a size range of 8.
Index for size 344 is 43 and contains chunks with a size range of 8.
Index for size 352 is 44 and contains chunks with a size range of 8.
Index for size 360 is 45 and contains chunks with a size range of 8.
Index for size 368 is 46 and contains chunks with a size range of 8.
Index for size 376 is 47 and contains chunks with a size range of 8.
Index for size 384 is 48 and contains chunks with a size range of 8.
Index for size 392 is 49 and contains chunks with a size range of 8.
Index for size 400 is 50 and contains chunks with a size range of 8.
Index for size 408 is 51 and contains chunks with a size range of 8.
Index for size 416 is 52 and contains chunks with a size range of 8.
Index for size 424 is 53 and contains chunks with a size range of 8.
Index for size 432 is 54 and contains chunks with a size range of 8.
Index for size 440 is 55 and contains chunks with a size range of 8.
Index for size 448 is 56 and contains chunks with a size range of 8.
Index for size 456 is 57 and contains chunks with a size range of 8.
Index for size 464 is 58 and contains chunks with a size range of 8.
Index for size 472 is 59 and contains chunks with a size range of 8.
Index for size 480 is 60 and contains chunks with a size range of 8.
Index for size 488 is 61 and contains chunks with a size range of 8.
Index for size 496 is 62 and contains chunks with a size range of 8.
Index for size 504 is 63 and contains chunks with a size range of 8.
Index for size 512 is 64 and contains chunks with a size range of 64.
Index for size 576 is 65 and contains chunks with a size range of 64.
Index for size 640 is 66 and contains chunks with a size range of 64.
Index for size 704 is 67 and contains chunks with a size range of 64.
Index for size 768 is 68 and contains chunks with a size range of 64.
Index for size 832 is 69 and contains chunks with a size range of 64.
Index for size 896 is 70 and contains chunks with a size range of 64.
Index for size 960 is 71 and contains chunks with a size range of 64.
Index for size 1024 is 72 and contains chunks with a size range of 64.
Index for size 1088 is 73 and contains chunks with a size range of 64.
Index for size 1152 is 74 and contains chunks with a size range of 64.
Index for size 1216 is 75 and contains chunks with a size range of 64.
Index for size 1280 is 76 and contains chunks with a size range of 64.
Index for size 1344 is 77 and contains chunks with a size range of 64.
Index for size 1408 is 78 and contains chunks with a size range of 64.
Index for size 1472 is 79 and contains chunks with a size range of 64.
Index for size 1536 is 80 and contains chunks with a size range of 64.
Index for size 1600 is 81 and contains chunks with a size range of 64.
Index for size 1664 is 82 and contains chunks with a size range of 64.
Index for size 1728 is 83 and contains chunks with a size range of 64.
Index for size 1792 is 84 and contains chunks with a size range of 64.
Index for size 1856 is 85 and contains chunks with a size range of 64.
Index for size 1920 is 86 and contains chunks with a size range of 64.
Index for size 1984 is 87 and contains chunks with a size range of 64.
Index for size 2048 is 88 and contains chunks with a size range of 64.
Index for size 2112 is 89 and contains chunks with a size range of 64.
Index for size 2176 is 90 and contains chunks with a size range of 64.
Index for size 2240 is 91 and contains chunks with a size range of 64.
Index for size 2304 is 92 and contains chunks with a size range of 64.
Index for size 2368 is 93 and contains chunks with a size range of 64.
Index for size 2432 is 94 and contains chunks with a size range of 64.
Index for size 2496 is 95 and contains chunks with a size range of 64.
Index for size 2560 is 96 and contains chunks with a size range of 512.
Index for size 3072 is 97 and contains chunks with a size range of 512.
Index for size 3584 is 98 and contains chunks with a size range of 512.
Index for size 4096 is 99 and contains chunks with a size range of 512.
Index for size 4608 is 100 and contains chunks with a size range of 512.
Index for size 5120 is 101 and contains chunks with a size range of 512.
Index for size 5632 is 102 and contains chunks with a size range of 512.
Index for size 6144 is 103 and contains chunks with a size range of 512.
Index for size 6656 is 104 and contains chunks with a size range of 512.
Index for size 7168 is 105 and contains chunks with a size range of 512.
Index for size 7680 is 106 and contains chunks with a size range of 512.
Index for size 8192 is 107 and contains chunks with a size range of 512.
Index for size 8704 is 108 and contains chunks with a size range of 512.
Index for size 9216 is 109 and contains chunks with a size range of 512.
Index for size 9728 is 110 and contains chunks with a size range of 512.
Index for size 10240 is 111 and contains chunks with a size range of 512.
Index for size 10752 is 112 and contains chunks with a size range of 1536.
Index for size 12288 is 113 and contains chunks with a size range of 4096.
Index for size 16384 is 114 and contains chunks with a size range of 4096.
Index for size 20480 is 115 and contains chunks with a size range of 4096.
Index for size 24576 is 116 and contains chunks with a size range of 4096.
Index for size 28672 is 117 and contains chunks with a size range of 4096.
Index for size 32768 is 118 and contains chunks with a size range of 4096.
Index for size 36864 is 119 and contains chunks with a size range of 4096.
Index for size 40960 is 120 and contains chunks with a size range of 24576.
Index for size 65536 is 121 and contains chunks with a size range of 32768.
Index for size 98304 is 122 and contains chunks with a size range of 32768.
Index for size 131072 is 123 and contains chunks with a size range of 32768.
Index for size 163840 is 124 and contains chunks with a size range of 98304.
Index for size 262144 is 125 and contains chunks with a size range of 262144.
Index for size 524288 is 126 and contains chunks with a size larger than 524288
```

Listing 45: generate_bin_map output for 32 bit architecture

The second output in Listing 46 represents the bin distributions for a 32 bit architecture, when *MALLOC_ALIGNMENT* is set to 16 byte (which can be set via compile time flags; see Section II-D2 on page 15).

```
Index for size 16 is 2 and contains chunks with a size range of 16.
Index for size 32 is 3 and contains chunks with a size range of 16.
Index for size 48 is 4 and contains chunks with a size range of 16.
Index for size 64 is 5 and contains chunks with a size range of 16.
Index for size 80 is 6 and contains chunks with a size range of 16.
Index for size 96 is 7 and contains chunks with a size range of 16.
Index for size 112 is 8 and contains chunks with a size range of 16.
Index for size 128 is 9 and contains chunks with a size range of 16.
Index for size 144 is 10 and contains chunks with a size range of 16.
Index for size 160 is 11 and contains chunks with a size range of 16.
Index for size 176 is 12 and contains chunks with a size range of 16.
Index for size 192 is 13 and contains chunks with a size range of 16.
Index for size 208 is 14 and contains chunks with a size range of 16.
Index for size 224 is 15 and contains chunks with a size range of 16.
Index for size 240 is 16 and contains chunks with a size range of 16.
Index for size 256 is 17 and contains chunks with a size range of 16.
Index for size 272 is 18 and contains chunks with a size range of 16.
Index for size 288 is 19 and contains chunks with a size range of 16.
Index for size 304 is 20 and contains chunks with a size range of 16.
Index for size 320 is 21 and contains chunks with a size range of 16.
Index for size 336 is 22 and contains chunks with a size range of 16.
Index for size 352 is 23 and contains chunks with a size range of 16.
Index for size 368 is 24 and contains chunks with a size range of 16.
Index for size 384 is 25 and contains chunks with a size range of 16.
Index for size 400 is 26 and contains chunks with a size range of 16.
Index for size 416 is 27 and contains chunks with a size range of 16.
Index for size 432 is 28 and contains chunks with a size range of 16.
Index for size 448 is 29 and contains chunks with a size range of 16.
Index for size 464 is 30 and contains chunks with a size range of 16.
Index for size 480 is 31 and contains chunks with a size range of 16.
Index for size 496 is 32 and contains chunks with a size range of 16.
Index for size 512 is 33 and contains chunks with a size range of 16.
Index for size 528 is 34 and contains chunks with a size range of 16.
Index for size 544 is 35 and contains chunks with a size range of 16.
Index for size 560 is 36 and contains chunks with a size range of 16.
Index for size 576 is 37 and contains chunks with a size range of 16.
Index for size 592 is 38 and contains chunks with a size range of 16.
Index for size 608 is 39 and contains chunks with a size range of 16.
Index for size 624 is 40 and contains chunks with a size range of 16.
Index for size 640 is 41 and contains chunks with a size range of 16.
Index for size 656 is 42 and contains chunks with a size range of 16.
Index for size 672 is 43 and contains chunks with a size range of 16.
Index for size 688 is 44 and contains chunks with a size range of 16.
Index for size 704 is 45 and contains chunks with a size range of 16.
Index for size 720 is 46 and contains chunks with a size range of 16.
Index for size 736 is 47 and contains chunks with a size range of 16.
Index for size 752 is 48 and contains chunks with a size range of 16.
Index for size 768 is 49 and contains chunks with a size range of 16.
Index for size 784 is 50 and contains chunks with a size range of 16.
Index for size 800 is 51 and contains chunks with a size range of 16.
Index for size 816 is 52 and contains chunks with a size range of 16.
Index for size 832 is 53 and contains chunks with a size range of 16.
Index for size 848 is 54 and contains chunks with a size range of 16.
Index for size 864 is 55 and contains chunks with a size range of 16.
Index for size 880 is 56 and contains chunks with a size range of 16.
Index for size 896 is 57 and contains chunks with a size range of 16.
Index for size 912 is 58 and contains chunks with a size range of 16.
Index for size 928 is 59 and contains chunks with a size range of 16.
Index for size 944 is 60 and contains chunks with a size range of 16.
Index for size 960 is 61 and contains chunks with a size range of 16.
Index for size 976 is 62 and contains chunks with a size range of 16.
Index for size 992 is 63 and contains chunks with a size range of 16.
Index for size 1008 is 64 and contains chunks with a size range of 16.
Index for size 1024 is 65 and contains chunks with a size range of 64.
Index for size 1088 is 66 and contains chunks with a size range of 64.
Index for size 1152 is 67 and contains chunks with a size range of 64.
Index for size 1216 is 68 and contains chunks with a size range of 64.
Index for size 1280 is 69 and contains chunks with a size range of 64.
Index for size 1344 is 70 and contains chunks with a size range of 64.
Index for size 1408 is 71 and contains chunks with a size range of 64.
Index for size 1472 is 72 and contains chunks with a size range of 64.
Index for size 1536 is 73 and contains chunks with a size range of 64.
Index for size 1600 is 74 and contains chunks with a size range of 64.
Index for size 1664 is 75 and contains chunks with a size range of 64.
Index for size 1728 is 76 and contains chunks with a size range of 64.
Index for size 1792 is 77 and contains chunks with a size range of 64.
Index for size 1856 is 78 and contains chunks with a size range of 64.
Index for size 1920 is 79 and contains chunks with a size range of 64.
Index for size 1984 is 80 and contains chunks with a size range of 64.
Index for size 2048 is 81 and contains chunks with a size range of 64.
Index for size 2112 is 82 and contains chunks with a size range of 64.
Index for size 2176 is 83 and contains chunks with a size range of 64.
Index for size 2240 is 84 and contains chunks with a size range of 64.
Index for size 2304 is 85 and contains chunks with a size range of 64.
Index for size 2368 is 86 and contains chunks with a size range of 64.
Index for size 2432 is 87 and contains chunks with a size range of 64.
Index for size 2496 is 88 and contains chunks with a size range of 64.
Index for size 2560 is 89 and contains chunks with a size range of 64.
Index for size 2624 is 90 and contains chunks with a size range of 64.
Index for size 2688 is 91 and contains chunks with a size range of 64.
Index for size 2752 is 92 and contains chunks with a size range of 64.
Index for size 2816 is 93 and contains chunks with a size range of 64.
Index for size 2880 is 94 and contains chunks with a size range of 64.
Index for size 2944 is 96 and contains chunks with a size range of 128.
Index for size 3072 is 97 and contains chunks with a size range of 512.
Index for size 3584 is 98 and contains chunks with a size range of 512.
Index for size 4096 is 99 and contains chunks with a size range of 512.
Index for size 4608 is 100 and contains chunks with a size range of 512.
Index for size 5120 is 101 and contains chunks with a size range of 512.
Index for size 5632 is 102 and contains chunks with a size range of 512.
Index for size 6144 is 103 and contains chunks with a size range of 512.
Index for size 6656 is 104 and contains chunks with a size range of 512.
Index for size 7168 is 105 and contains chunks with a size range of 512.
Index for size 7680 is 106 and contains chunks with a size range of 512.
Index for size 8192 is 107 and contains chunks with a size range of 512.
Index for size 8704 is 108 and contains chunks with a size range of 512.
Index for size 9216 is 109 and contains chunks with a size range of 512.
Index for size 9728 is 110 and contains chunks with a size range of 512.
Index for size 10240 is 111 and contains chunks with a size range of 512.
Index for size 10752 is 112 and contains chunks with a size range of 1536.
Index for size 12288 is 113 and contains chunks with a size range of 4096.
Index for size 16384 is 114 and contains chunks with a size range of 4096.
Index for size 20480 is 115 and contains chunks with a size range of 4096.
Index for size 24576 is 116 and contains chunks with a size range of 4096.
Index for size 28672 is 117 and contains chunks with a size range of 4096.
Index for size 32768 is 118 and contains chunks with a size range of 4096.
Index for size 36864 is 119 and contains chunks with a size range of 4096.
Index for size 40960 is 120 and contains chunks with a size range of 24576.
Index for size 65536 is 121 and contains chunks with a size range of 32768.
Index for size 98304 is 122 and contains chunks with a size range of 32768.
Index for size 131072 is 123 and contains chunks with a size range of 32768.
Index for size 163840 is 124 and contains chunks with a size range of 98304.
Index for size 262144 is 125 and contains chunks with a size range of 262144.
Index for size 524288 is 126 and contains chunks with a size larger than 524288
```

Listing 46: generate_bin_map output for 32 bit architecture with large MALLOC_ALIGNMENT

The following last listing shows the typical distribution for 64 bit architectures.

```
Index for size 16 is 1 and contains chunks with a size range of 16.
Index for size 32 is 2 and contains chunks with a size range of 16.
Index for size 48 is 3 and contains chunks with a size range of 16.
Index for size 64 is 4 and contains chunks with a size range of 16.
Index for size 80 is 5 and contains chunks with a size range of 16.
Index for size 96 is 6 and contains chunks with a size range of 16.
Index for size 112 is 7 and contains chunks with a size range of 16.
Index for size 128 is 8 and contains chunks with a size range of 16.
Index for size 144 is 9 and contains chunks with a size range of 16.
Index for size 160 is 10 and contains chunks with a size range of 16.
Index for size 176 is 11 and contains chunks with a size range of 16.
Index for size 192 is 12 and contains chunks with a size range of 16.
Index for size 208 is 13 and contains chunks with a size range of 16.
Index for size 224 is 14 and contains chunks with a size range of 16.
Index for size 240 is 15 and contains chunks with a size range of 16.
Index for size 256 is 16 and contains chunks with a size range of 16.
Index for size 272 is 17 and contains chunks with a size range of 16.
Index for size 288 is 18 and contains chunks with a size range of 16.
Index for size 304 is 19 and contains chunks with a size range of 16.
Index for size 320 is 20 and contains chunks with a size range of 16.
Index for size 336 is 21 and contains chunks with a size range of 16.
Index for size 352 is 22 and contains chunks with a size range of 16.
Index for size 368 is 23 and contains chunks with a size range of 16.
Index for size 384 is 24 and contains chunks with a size range of 16.
Index for size 400 is 25 and contains chunks with a size range of 16.
Index for size 416 is 26 and contains chunks with a size range of 16.
Index for size 432 is 27 and contains chunks with a size range of 16.
Index for size 448 is 28 and contains chunks with a size range of 16.
Index for size 464 is 29 and contains chunks with a size range of 16.
Index for size 480 is 30 and contains chunks with a size range of 16.
Index for size 496 is 31 and contains chunks with a size range of 16.
Index for size 512 is 32 and contains chunks with a size range of 16.
Index for size 528 is 33 and contains chunks with a size range of 16.
Index for size 544 is 34 and contains chunks with a size range of 16.
Index for size 560 is 35 and contains chunks with a size range of 16.
Index for size 576 is 36 and contains chunks with a size range of 16.
Index for size 592 is 37 and contains chunks with a size range of 16.
Index for size 608 is 38 and contains chunks with a size range of 16.
Index for size 624 is 39 and contains chunks with a size range of 16.
Index for size 640 is 40 and contains chunks with a size range of 16.
Index for size 656 is 41 and contains chunks with a size range of 16.
Index for size 672 is 42 and contains chunks with a size range of 16.
Index for size 688 is 43 and contains chunks with a size range of 16.
Index for size 704 is 44 and contains chunks with a size range of 16.
Index for size 720 is 45 and contains chunks with a size range of 16.
Index for size 736 is 46 and contains chunks with a size range of 16.
Index for size 752 is 47 and contains chunks with a size range of 16.
Index for size 768 is 48 and contains chunks with a size range of 16.
Index for size 784 is 49 and contains chunks with a size range of 16.
Index for size 800 is 50 and contains chunks with a size range of 16.
Index for size 816 is 51 and contains chunks with a size range of 16.
Index for size 832 is 52 and contains chunks with a size range of 16.
Index for size 848 is 53 and contains chunks with a size range of 16.
Index for size 864 is 54 and contains chunks with a size range of 16.
Index for size 880 is 55 and contains chunks with a size range of 16.
Index for size 896 is 56 and contains chunks with a size range of 16.
Index for size 912 is 57 and contains chunks with a size range of 16.
Index for size 928 is 58 and contains chunks with a size range of 16.
Index for size 944 is 59 and contains chunks with a size range of 16.
Index for size 960 is 60 and contains chunks with a size range of 16.
Index for size 976 is 61 and contains chunks with a size range of 16.
Index for size 992 is 62 and contains chunks with a size range of 16.
Index for size 1008 is 63 and contains chunks with a size range of 16.
Index for size 1024 is 64 and contains chunks with a size range of 64.
Index for size 1088 is 65 and contains chunks with a size range of 64.
Index for size 1152 is 66 and contains chunks with a size range of 64.
Index for size 1216 is 67 and contains chunks with a size range of 64.
Index for size 1280 is 68 and contains chunks with a size range of 64.
Index for size 1344 is 69 and contains chunks with a size range of 64.
Index for size 1408 is 70 and contains chunks with a size range of 64.
Index for size 1472 is 71 and contains chunks with a size range of 64.
Index for size 1536 is 72 and contains chunks with a size range of 64.
Index for size 1600 is 73 and contains chunks with a size range of 64.
Index for size 1664 is 74 and contains chunks with a size range of 64.
Index for size 1728 is 75 and contains chunks with a size range of 64.
Index for size 1792 is 76 and contains chunks with a size range of 64.
Index for size 1856 is 77 and contains chunks with a size range of 64.
Index for size 1920 is 78 and contains chunks with a size range of 64.
Index for size 1984 is 79 and contains chunks with a size range of 64.
Index for size 2048 is 80 and contains chunks with a size range of 64.
Index for size 2112 is 81 and contains chunks with a size range of 64.
Index for size 2176 is 82 and contains chunks with a size range of 64.
Index for size 2240 is 83 and contains chunks with a size range of 64.
Index for size 2304 is 84 and contains chunks with a size range of 64.
Index for size 2368 is 85 and contains chunks with a size range of 64.
Index for size 2432 is 86 and contains chunks with a size range of 64.
Index for size 2496 is 87 and contains chunks with a size range of 64.
Index for size 2560 is 88 and contains chunks with a size range of 64.
Index for size 2624 is 89 and contains chunks with a size range of 64.
Index for size 2688 is 90 and contains chunks with a size range of 64.
Index for size 2752 is 91 and contains chunks with a size range of 64.
Index for size 2816 is 92 and contains chunks with a size range of 64.
Index for size 2880 is 93 and contains chunks with a size range of 64.
Index for size 2944 is 94 and contains chunks with a size range of 64.
Index for size 3008 is 95 and contains chunks with a size range of 64.
Index for size 3072 is 96 and contains chunks with a size range of 64.
Index for size 3136 is 97 and contains chunks with a size range of 448.
Index for size 3584 is 98 and contains chunks with a size range of 512.
Index for size 4096 is 99 and contains chunks with a size range of 512.
Index for size 4608 is 100 and contains chunks with a size range of 512.
Index for size 5120 is 101 and contains chunks with a size range of 512.
Index for size 5632 is 102 and contains chunks with a size range of 512.
Index for size 6144 is 103 and contains chunks with a size range of 512.
Index for size 6656 is 104 and contains chunks with a size range of 512.
Index for size 7168 is 105 and contains chunks with a size range of 512.
Index for size 7680 is 106 and contains chunks with a size range of 512.
Index for size 8192 is 107 and contains chunks with a size range of 512.
Index for size 8704 is 108 and contains chunks with a size range of 512.
Index for size 9216 is 109 and contains chunks with a size range of 512.
Index for size 9728 is 110 and contains chunks with a size range of 512.
Index for size 10240 is 111 and contains chunks with a size range of 512.
Index for size 10752 is 112 and contains chunks with a size range of 1536.
Index for size 12288 is 113 and contains chunks with a size range of 4096.
Index for size 16384 is 114 and contains chunks with a size range of 4096.
Index for size 20480 is 115 and contains chunks with a size range of 4096.
Index for size 24576 is 116 and contains chunks with a size range of 4096.
Index for size 28672 is 117 and contains chunks with a size range of 4096.
Index for size 32768 is 118 and contains chunks with a size range of 4096.
Index for size 36864 is 119 and contains chunks with a size range of 4096.
Index for size 40960 is 120 and contains chunks with a size range of 24576.
Index for size 65536 is 121 and contains chunks with a size range of 32768.
Index for size 98304 is 122 and contains chunks with a size range of 32768.
Index for size 131072 is 123 and contains chunks with a size range of 32768.
Index for size 163840 is 124 and contains chunks with a size range of 98304.
Index for size 262144 is 125 and contains chunks with a size range of 262144.
Index for size 524288 is 126 and contains chunks with a size larger than 524288
```

Listing 47: generate_bin_map output for 64 bit architecture

References

Block, F., Dewald, A., 2017. Linux memory forensics: Dissecting the user space process heap. In: DFRWS USA.

Case, A., Marziale, L., Neckar, C., Richard, G. G., 2010. Treasure and tragedy in kmem_cache mining for live forensics investigation. digital investigation 7, S41–S47.

Cohen, M., 2015. Forensic analysis of windows user space applications through heap allocations. 3rd IEEE International Workshop on Security and Forensics in Communication Systems 2015, 1138–1145 [Visited on

04.03.2016].
URL http://www.rekall-forensic.com/docs/References/Papers/p1138-cohen.pdf

Ferguson, J. N., 2007. Understanding the heap by breaking it. Black Hat USA [Visited on 22.03.2016].
URL http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf

Foundation, T. V., 2016. Volatility. [Visited on 04.03.2016].
URL http://www.volatilityfoundation.org

Franks, J., 1999. Rfc 2617 - http authentication. [Visited on 03.08.2016].
URL https://tools.ietf.org/html/rfc2617

Free Software Foundation Inc., 2016. Gnu c library (glibc). [Visited on 15.08.2016].
URL https://www.gnu.org/software/libc/

Ghemawat, S., Menage, P., 2015. Tcmalloc - claimed performance improvement over glibc. [Visited on 16.08.2016].
URL http://goog-perftools.sourceforge.net/doc/tcmalloc.html

Gloger, W., 2006. ptmalloc2. [Visited on 15.08.2016].
URL http://www.malloc.de/en/

Google Inc, 2016a. bash: Rekall plugin to extract the command history. [Visited on 04.03.2016].
URL http://www.rekall-forensic.com/docs/Manual/Plugins/Linux/#bash

Google Inc, 2016b. cmdscan: Rekall plugin to scan the bash process for history. [Visited on 04.03.2016].
URL http://www.rekall-forensic.com/docs/Manual/Plugins/Windows/#cmdscan

Google Inc, 2016c. Rekall memory forensic framework. [Visited on 04.03.2016].
URL http://www.rekall-forensic.com

Lea, D., 2006. dlmalloc. [Visited on 15.08.2016].
URL http://www.malloc.de/en/

Leppert, S., 2012. Android memory dump analysis. Master's thesis, [Visited on 04.03.2016].
URL https://www1.informatik.uni-erlangen.de/filepool/thesis/android_memory_forensics.pdf

Ligh, M. H., Case, A., Levy, J., Walters, A., 2014. The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory. John Wiley & Sons.

Macht, H., 2013. Live memory forensics on android with volatility. Master's thesis, [Visited on 04.03.2016].
URL https://www1.informatik.uni-erlangen.de/filepool/publications/Live_Memory_Forensics_on_Android_with_Volatility.pdf

Urrea, J. M., 2006. An analysis of linux ram forensics. Master's thesis, Monterey, California. Naval Postgraduate School, [Visited on 04.03.2016].
URL http://calhoun.nps.edu/bitstream/handle/10945/2933/06Mar_Urrea.pdf

Valasek, C., 2010. Understanding the low fragmentation heap. [Visited on 15.08.2016].
URL http://illmatics.com/Understanding_the_LFH.pdf

Wilson, P. R., Johnstone, M. S., Neely, M., Boles, D., 1995. Dynamic storage allocation: A survey and critical review. In: Memory Management. Springer, pp. 1–116.