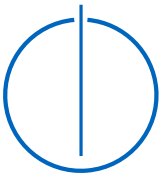
DEPARTMENT OF INFORMATICS
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

CharWars: A New Heap

Clemens Jonischkeit



DEPARTMENT OF INFORMATICS
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

CharWars: Ein Neuer Heap

CharWars: A New Heap

Author:	Clemens Jonischkeit
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Julian Kirsch
Submission Date:	15. Oktober 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum	Clemens Jonischkeit

Acknowledgments

This thesis would not have been possible without the help of several people I wholeheartedly wish to thank:
First, I want to thank my family – for your constant support during my studies, and for offering me the opportunity to go to university to study the subject I like.
Second, I also want to thank my caring girlfriend, Heidi, for all the love and support she gives me and for cheering me up when i am stressed out. She also helped me a lot to overcome the last draining days of this thesis with ease. Moreover, I want to thank my supervisor, Prof. Eckert, for giving me the opportunity to write this thesis, with this title, and for raising my interest in IT security in general.
Finally, i wanted to thank my advisor Julian Kirsch for providing me expertise in exploiting binaries without his knowledge this thesis could not exist in its current state. Thank you for discussions and helpful input during the time.

Abstract

Despite many improvements and the development of safer programming languages memory corruption vulnerabilities, such as buffer overflows are still prevalent in current software. Attackers abusing memory corruption vulnerabilities most typically target control structures that govern the path of program execution. To counter this threat, in recent years, control-flow integrity techniques have been introduced, in an effort to mitigate corruption of these critical data structures. This is achieved by constraining the execution to valid paths calculated at compilation-time. Unfortunately, control-flow integrity techniques do not prevent memory corruptions in the first place and thus have no impact on attacks targeting non-control-flow relevant data.

This thesis evaluates the attack surface offered by dynamic memory allocators and demonstrates using the example of two popular implementations—ptmalloc and jemalloc—how the corruption of heap management structures (non-control-flow relevant data) can still be used to hijack the control-flow. To evaluate the security, this thesis proposes an approach that is capable of comparing heap implementations with regard to their resistance against memory corruption attacks. Then, the design of a POSIX compatible heap measurably improving security is presented. Security improvements are achieved by separating user-controlled allocated buffers from management data and severely limiting the type and amount of heap management information that traditional implementations place close to allocated memory under attacker control.

Depending on allocation characteristics such as size and order, the performance of our proposed heap implementation is competitive with the standard malloc implementation used by glibc on Linux, achieving 83 percent of its performance on average. Overall we conclude that dynamic memory allocators can be made more resistant against memory corruption attacks while still maintaining reasonable performance.

Contents

1 Introduction

This section starts by motivating the work done in this thesis by showing its relevance even in the presence of modern protection mechanism. Then the contributions done in this thesis are listed.

1.1 Motivation

One of the ways attackers can alter the behavior of applications is through exploitation of memory corruption vulnerabilities. These memory corruptions vary in form, location, and additionally may constrain the type of the data attackers might use during their attack: For example, from an attacker's point of view, the POSIX `gets` function offers the possibility of an arbitrary length buffer overflow, but stops at the first occurrence of a newline (ASCII `0x0a`) character (therefore constraining the usable data bytes during an overflow to anything except a newline character). As another example, the `scanf` function with a size modifier m (e.g. `scanf("%100s", buf)`) with 100 being the size modifier), only allows to write m attacker controlled bytes followed by an *additional* a terminating null byte to the destination.

One famous instance of a buffer overflow being used to attack wide parts of the internet was the Slammer worm [23]. The impact of a successful exploitation of corruptions depends on the memory that is corrupted; even though all memory within the address space of any computer program running on a von-Neumann architecture is equivalent, in practice, different allocation strategies are used for different memory types. For example, an architectural *stack* usually holds control flow relevant information such as a return address, indicating where execution should continue after the current subroutine finished executing. Consequently, if such a return pointer stored on the stack is altered, the control flow is also altered. Such deviation from the original control flow can often be used by a malicious attacker to achieve arbitrary code execution.

Surprisingly, also the heap, which is commonly used to store dynamically allocated data be attacked with the goal to alter the control flow. In the simplest case this is possible due to the presence of control flow governing data on the heap. For example, the C++ language supports the concept of class inheritance. Inheritance is implemented by modern compilers in such a way that they store a pointer to a function pointer table so that dispatches of virtual function calls find the exact function to execute. Again, these data

structures could be overwritten by a malicious attacker and could then be abused to alter the control flow.

To combat corruption of these control structures, control flow integrity (CFI) mechanisms can be employed. For example, one mechanism that found its way into modern compilers is a *Shadow Stack* implemented by the *clang* compiler. This shadow stack ensures that after the execution of each function, control flow eventually returns to the point directly behind the dispatching call. Furthermore, *clang-cfi* tries to rearrange virtual function pointer tables in such a way that corruptions thereof resulting in type confusions can be recognized by the runtime environment. The goal of CFI is to constrain the control flow to only take control flow transfers that are part of a statically verified control flow graph created ahead of execution [1]. This prevents attacker from injecting a forged virtual function table since the control flow transfers described by these tables are not part of the execution tree. The shadow stack on the other hand does not protect function dispatchs but rather control flow transfers returning from functions. The call stack is build in memory seperated from the local variables and thus are unreachable by buffer overflow attacks. Attacker therfore cannot overwrite these addresses and hijack the control flow.

Unfortunately, the protection offered by current CFI mechanisms however are incomplete, as they aim to counter only attacks targeting control structures and do not prevent memory corruptions in the first place [7]; modern compiled computer programs offer a plethora of control flow relevant data structures rather than just virtual function pointers on the heap or return addresses on the stack. These control structures are valuable targets in context of a memory corruption attacks. One such example of a control flow relevant data structure is part of the mechanisms used to enable dynamic linking: Due to address space layout randomization the exact position of shared libraries in memory cannot be known at compile time. Therefore, the addresses of external functions must be relocated and made known to any application intending to use them. To eliminate multiple relocations of the same symbol, compilers like the GNU C compiler (GCC) create a global offset table (GOT). This table is used to hold the addresses of external symbols imported from shared libraries. Attacks against HULL httpd, a Linux based http server, have demonstrated that heap based overflows can be used to corrupt entries in the GOT, and that this corruption of the GOT can be used to redirect the control flow to functions chosen by the attacker [8]. To prevent the GOT from being altered RELRO can be used. RELRO changes the page permissions of the pointer table to be read only, so that a malicious write access to this table cannot succeed [9]. Nevertheless, as will be demonstrated in this thesis, there exist many more valuable targets that can successfully attacked during the exploitation of a heap-based buffer overflow. Based on this we aim to solve the root cause of such attacks by proposing a new layout for a hardened dynamic memory allocator.

1.2 Contributions

There are multiple contributions done in this thesis. The first contribution is done by analyzing the attack vectors common dynamic memory allocators present by their management structures. The heaps analyzed are ptmalloc, currently implemented as part of the Gnu C library (glibc) that is the default C library on many Linux systems. The second implementation dissected is jemalloc, used by the popular web browser Mozilla Firefox, the MySQL database MariaDB and part of the FreeBSDs c library. To demonstrate lack of security provided by these libraries, attacks are presented that elevate a buffer overflow into a code execution attack. Before a more secure heap can be created, first a method has

to be found that is able to compare two dynamic memory allocators. Such method is the second contribution by this thesis. Then a heap is outlined that is able to withstand attacks that succeeded against ptmalloc and jemalloc. This new heap is hardened sufficiently so that exploiting its management structures cannot be used to elevate the capabilities of attackers, by increasing the requirements needed to successfully exploit its structures. Based on these principles a dynamic memory allocator is then implemented and compared to ptmalloc in terms of performance.

2 Background

In this section we briefly motivate and introduce the usage of dynamic memory allocators. Afterwards, we explain why dynamic memory allocators inherently can be attacked in case of errors introduced by a human programmer.

2.1 Dynamic Memory Allocation

Throughout their lifetime, any application stores and manipulates numerous different objects in memory. These can comprise global objects which remain in scope during the program's lifetime but also local variables with their lifetime starting at function call and ending upon return from the same function.

But not only the lifetime of these in-memory objects varies, also the storage location can be different depending on the type of usage. With the amount and size of global variables being computed at compile time, the compiler is able to reserve an appropriate amount of space in the program image itself. Functions, on the other hand, can be called recursively such that assigning fixed memory locations to local variables would introduce corruptions once a recursively called function tried to update its own variables. To circumvent this problem, function calls create a stack frame and local variables are stored inside that stack frame. This allows function calls to be recursive as each such call creates a small private memory area for its variables on the stack without affecting local variables of the calling function.

Apart from statically allocated global and volatile local variables sometimes programmer need to pass around objects of global scope with a size unknown at compilation time. An example could be the classical doubly-linked list which should be capable of storing an unknown amount of data at runtime. Placing such an object in statically allocated global memory is not possible as the size is not known at compile time, neither can they be stored in the function's stack frame, as returning from the function invalidates such memory, resulting in passing a reference to potentially invalid memory.

This gap is filled by dynamic memory allocators, also called *program heap*, not to be confused with the tree-based data structure. The *heap* provides variable sized storage via a standardized application programming interface. For example, in context of the C programming language, the *malloc* function can be used to allocate memory at runtime.

Such dynamic memory allocators are implemented as libraries or part of libraries and suffer from the same restrictions the original program suffers from—not being able to

have variably sized buffers of uncertain lifetimes. Dynamic memory allocators solve this problem by using operating system dependent system calls that map a chosen amount of virtual memory into the address space and presenting an abstract view on this memory: Allocations made by the program are then served using this, by the operating system provided, memory. To mitigate differences between different operating systems a common interface, the portable operating system interface (POSIX) has been created. POSIX does not specify the way heaps can acquire memory from operating systems, but rather parts of the API of the heap.

One caveat of dynamic memory allocation is, however, that the heap does not know the exact lifetime of the object, so the memory cannot be freed automatically. This puts the burden of keeping track of allocated memory on system developers requiring them to free allocated memory when appropriate (i.e. after no part of the program logic needs the information contained in the allocated memory). Failure to do so can come in different variants:

First an object can get lost, possibly by overwriting the only pointer holding a reference to it. This has no immediate impact on the application, as the memory is just not released, and no other object can take its space. This is exactly the problem an application may run into at some later point in time: As no other allocation can take the space of the object, the allocator can run into the problem, that it does not have enough memory available to server some allocation. While the heap can request increasingly large amounts of memory from the operating system, at some point (due to physical resource constraints) the system's memory will be exhausted, and the allocation will fail. Unexpectedly not being able to allocate objects can then cause problems for the program as it may be reliant on that memory to perform necessary operations. Possible outcomes of this scenario reach from unexpected program termination to bringing the entire system to a halt.

Second, another problem from manually freeing objects arises when the object is freed too early, meaning there still exists a reference to the object that is in active use. The outcome of this scenario is unpredictable, as the dynamic memory allocator may reuse the freed object to serve other allocation request. In this case the first reference can point to invalid memory or even different objects. Such a pointer is commonly referred to as *dangling pointer*, and can cause immediate problems for the program: Accessing a dangling pointer assumes the object types of the old and the new allocation to match with devastating results in practice.

A third problem can arise when the same allocated memory region is freed multiple times, commonly simply referred to as *double free*. While the first occurrence of the free call does behave in the intended way, namely returning the memory to the allocator, the result of the second call is unpredictable and depends on the allocator itself and the fact whether the memory has already been reused. The latter case is similar to freeing the target object too early and is not necessarily detectable by the allocator. This can also lead to an object being freed twice, once by an erroneous free and a second time when the lifetime of the object originally ends. In the second case, when the object has been freed and the memory has been reclaimed by the dynamic memory allocator, but has not been used to serve another allocation, the second free is detectable. This is due to the pointer passed to free pointing to memory that had not been handed out by the allocator and thus the object cannot be valid.Possible results include a call to free having zero effect, for example when objects are managed in a bitmap, or can lead to allocating the same memory for multiple objects or even to the corruption of allocator structures along unpredictable results.

This work discusses all the scenarios painted above in context of popular allocators in Section 3.

2.2 Attacker Model

When designing security mechanisms it is important to specify the attacker model that the designed mechanism should protect against. This attacker model describes the capabilities an imaginary adversary has attacking the protected application.

One prominent attacker model regarding the security of cryptographic protocols for example is the Dolev-Yao intruder [12]. For the purpose of exploitation mitigation, many mechanisms proposed unfortunately lack a clear specification of the attacker model they protect against [1]. In the context of application security, meaningful capabilities include the ability to perform buffer overflows, buffer underflows, format string attacks, memory corruptions relative to objects on the heap or the stack. Apart from the capabilities the attacker has, the model also specifies which knowledge an attacker has prior to exploitation.

In context of our work mainly the knowledge about the layout of the address space is significant to exploitation, as it allows deduction of the full addresses of attacker controlled data or the distance between the heap and other targets of interest in memory.

3 Security of Existing Designs

Building on the definition of attacker models introduced in section 2.2 this section outlines a method that can be used to qualitatively assess the security of different heap implementations provided against erroneous usage. Afterwards, two widely used dynamic memory allocators are analyzed, namely *ptmalloc*, implemented in the glibc, and *jemalloc*, used by multiple free software projects, such as the OpenBSD C standard library, the popular web-browser Firefox, and mariadb, a mysql implementation. We conclude by identifying flaws in these allocation mechanisms and explain their security implications.

3.1 Ranking the Security

To quantify the resistance any heap implementation offers against malicious adversaries a method has to be created first to compare the security of two implementations of dynamic memory allocators. We base our security measure on the strength of an attacker succeeding to exploit a given implementation: A heap ranking lower on this measure is vulnerable to all attackers any higher ranking heap is vulnerable to and a more secure heap has to resist all attackers a less secure heap resist. One difficulty in finding such a measure is that heaps could be resistant to very different attacker types. Conceivably, one design might be able to resist all types of buffer overflow attacks, but be vulnerable to buffer underflows, while a second heap might be exploited by buffer overflows but resist any underflow. To our perception, neither of the designs may rank higher than the other, nor are they even directly comparable.

This thesis proposes to directly use attacker models for comparing the security of heaps. The goal is to rank the heaps' security while disregarding the program utilizing the dynamic memory allocator. For this reason it is assumed that any attacker can choose any function of the dynamic memory allocator to be invoked with objects he chooses. This includes the ability to read out the contents of an object and fill it with any content he specifies. While the attacker can control the creation, their size and the destruction of objects, he may not access invalid objects and does not know the virtual address of the object. One can think of the attacker controlling every function call to the heap, but all addresses are substituted by objects or object ids. An example how this can be implemented can be found in the appendix.

In order not to test excessive amounts of attacker models to determine the security properties of a heap attacker models can be viewed as sets of capabilities. This set of

```
struct malloc_chunk {
INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
INTERNAL_SIZE_T      mchunk_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;
};
```

Listing 1: *Structure used by ptmalloc to organize allocations as implemented in glibc 2.27[15]*

capabilities forms a lattice under set inclusion. The bottom element is the attacker that has only the aforementioned capabilities of controlling the interface. The top element is an attacker who has knowledge of the address space layout and can perform arbitrary read and write operations in the entire address range. This attacker is as powerful as one that has full knowledge of the address space and has the capability to read and write memory at arbitrary offsets to objects. This is due to every exploit of the weaker attacker can be simulated by the stronger attacker by calculating the absolute addresses he knows because of his knowledge of the address space layout from the offsets. Any attacker with arbitrary memory access can also skip exploiting the dynamic memory allocator and instead directly corrupt the data one would try to get access to. With this notion of attacker models as a lattice one can now compare multiple dynamic memory allocator implementations. A heap A is at least as secure as a heap B iff for all attacker models B resists against A also resist against them.

3.2 Ptmalloc

The first memory allocator under analysis is *ptmalloc*. This Heap is implemented in the glibc and widely used on Linux operating systems. I will describe its design in a bottom up manner, starting with allocated areas of memory, then explain how free memory areas are tracked, and finally elaborating on the top level structure 'arena' and its purpose.

Based on the size of an allocation there are three different ways an allocation can be handled. Small allocations are handled using so-called *fastbins*, and especially large memory requests are served using *huge bins*. All other allocations are handled with *unsorted bins*. All allocations share a common structure in memory that can be seen in listing 1. Depending on the type of chunk, its allocation state, and the preceding chunks allocation state, fields are unused or shadowed by user data. The only field that must always be present is `mchunk_size`. Apart from the size, this field also carries further information in the form of flags. As the granularity for allocations in ptmalloc is `0x10`, the four least significant bits of the size field can be used as flags. When the chunk's size is read, they can simply be masked out. The least significant bit is the `PREV_INUSE` flag, the next one is called `IS_MMAPED` flag, the third is `NON_MAIN_ARENA` flag, and the last bit is unused.

`PREV_INUSE` indicates whether the preceding chunk is allocated with the bit being set meaning the predecessor of the current chunk is in use. This flag also has a direct impact on the `mchunk_prev_size` field, as it is only valid if the bit is set, otherwise it is used by the

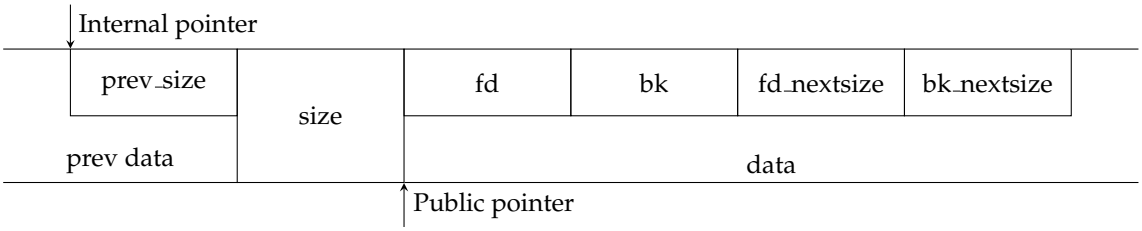


Figure 1: *The struct malloc_chunk residing in memory*

preceding chunk as part of its user data, resulting in it being unpredictable if `PREV_INUSE` being set. This mechanic saves a non-negligible amount of memory.

The flag `IS_MMAPED` indicates the use of the `mmap` system call to serve this allocation. It is used for huge chunks that are individually mapped into memory. This piece of information is crucial when it comes to freeing an allocation having that flag set. The main portion of the heap is built around the `sbrk` mechanic that can be efficiently used to expand or shrink the heap. Chunks that are mapped into memory via `mmap` are not part of the heap managed via `sbrk` and therefore have to be reclaimed manually by unmapping the appropriate address range.

Lastly, `NON_MAIN_ARENA` indicates that the chunk is part of an arena not being the main arena. The main arena is the original arena created. In multithreaded environments additional arenas may be present and this bit indicates that the owning arena must be determined. If this bit is not set, the chunk is part of the main arena stripping away the complexity of finding the owning arena and thus improving the performance if only one arena is required.

While ptmalloc internally works with pointer to the beginning of the struct, as size must always be valid and may never be overwritten by user data, the pointer returned points to the forward pointer. This results in the field `fd` and all successive fields of the struct to be shadowed by user data. Also sizes for the allocations are internally calculated, so that user data also may shadow the `prev_size` field of the *next* allocation. The position of the struct in memory and the potential shadowing by user data can be seen in figure 1. Depending on the size of the chunk, free elements are handled differently. Small chunks are grouped in fastbins, and there exists a fastbin for each chunk size smaller than a certain threshold. These fastbins handle free chunks using a singly linked list utilizing the `fd` field of the `malloc_chunk` struct. This allows for efficient freeing and allocating of these chunks, and as a fastbin only contains chunks of the same size, the singly linked list builds a stack of chunks handled in last in first out order. Free unsorted bins on the other hand store free chunks via doubly linked lists. This is necessary as chunks in unsorted bins can be of various sizes and upon an allocation, a chunk of the right size must be found, which not necessarily is the first and then it has to be removed from the list. It is worth to note, that while the singly linked list of the fastbins is terminated via a `NULL` pointer, the linked list of an unsorted bin is terminated via a pointer into the arena. The main arena is part of the libc static data, and such unsorted bins are terminated using pointer into the libc. There also exists a special chunk, namely the topchunk. It is the last chunk on the heap and contains the rest of the memory. It can never be allocated, there can just be memory split of to allocate chunks. If the top chunk is too small to accommodate an allocation or if it would be used up, the heap instead is extended via the `sbrk` mechanic and thus the size of the top chunk is increased. That way it is possible to serve these allocations while keeping the top chunk.

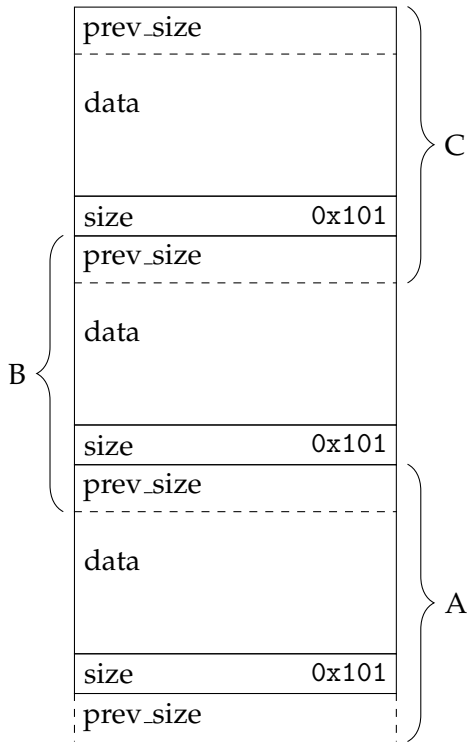


Figure 2: *Forced Heap layout prior to exploitation*

In glibc 2.27 a further technique was introduced to dlmallc called tcaches. The idea is to reduce contention for the arena lock by keeping a list of chunks freed per thread. By not freeing the chunk globally, the arena has not to be locked and the chunk will not be handed out to other allocations. This thread locally freed chunk can only be allocated by the same thread freeing it, and it will only be used if the exact size of the chunk is requested[11]. To enable this mechanic the thread local storage holds a configurable amount of singly linked lists. Each linked list is assigned a size, and all elements in the list are of that size. Upon an element being freed, it is first checked whether there exists a list for elements of that size in thread local storage. If such list exists, and the list has not reached its maximum depth, instead of passing the free to the arena, the chunk is prepended to the list. If no such list exists, but not all lists are used, a new list is created and the element being freed is prepended instead of globally freed.

There are multiple ways to exploit the heap management structures of ptmalloc[16, 25]. When heap allocations and frees are controlled, the heap layout can be forced and overflowing a single null character can be used to escalate the exploit to remote code execution. The exploit can be broken down into four subsections, first two chunks are overlapped, so that access to the pointer of a free list is gained. Then the memory layout is leaked via the pointer of the unsorted bin linked list yielding the addresses of the heap and libc. Third, the free list is corrupted, to allocate a chunk containing the `__free_hook` in the libc, which is subsequently overwritten with a pointer to `system`. From there on freeing an object instead calls system, with the argument being the content of the chunk, thus freeing an element with contents ”sh” spawns a shell.

There are many different variations to do each step in the exploit, so in this part my personal flavor will be used. This variant starts with a Heap layout, like seen in figure 2.

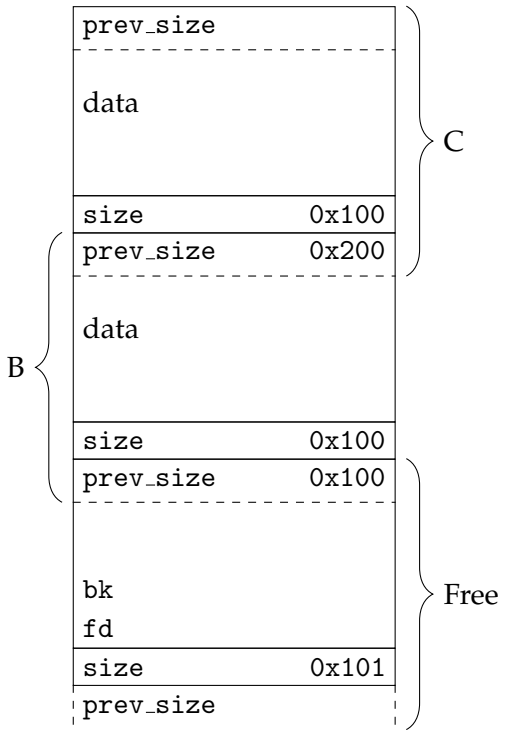


Figure 3: *State of the Heap after overflowing B and freeing A*

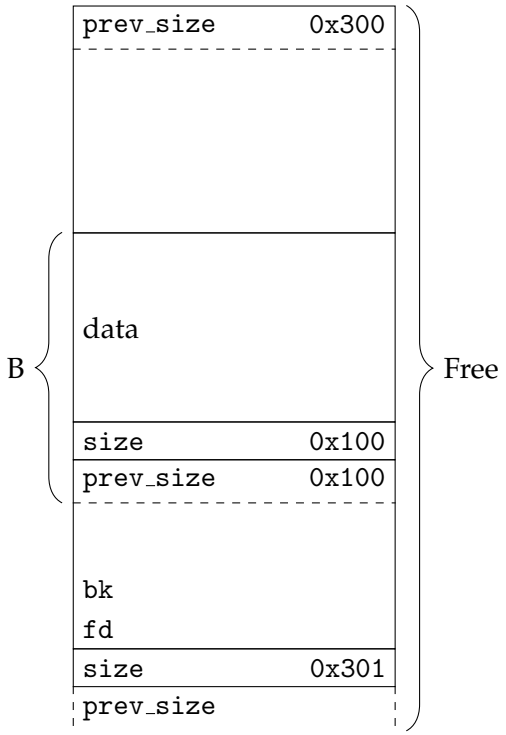


Figure 4: *Chunk B gets swallowed due to the forged* `prev_size` *field*

Here ”A”, ”B”, and ”C” are individual allocations. The size of ”B” does not matter much, but to use the full allocation and have an overflow of the allocation also overflow the chunk, it must be of a size $0x10 * n + 8$ where $1 \leq n$. ”A” has to be larger than $0x80$, so it is inserted into an unsorted bin upon free. Additionally, to be handled by unsorted bins, the size of ”C” must be a multiple of $0x100$. This is due to internal checks in ptmalloc that would detect the missing header of the next allocation and abort the program. If a size other than a multiple of `0x100` is used, overflowing the least byte will alter the size, not only the flags. Detection can be mitigated by placing a fake chunk in the Body of ”C”. First ”A” is freed, so that it is inserted into the unsorted bin list. Then from ”B” the terminating `null` byte is overflown into the size field of ”C”, residing directly after ”B”. While this does not change the size stored in the size field of ”C”, it overwrites its flags, clearing the `PREV_INUSE` flag, indicating ”B” to be free. Since management structs overlap, overflowing into the size field also overwrites the `prev_size` field which overlaps the last bytes of the buffer. It is important to not set the value of this field to the size of ”B”, but rather to the size of ”A” and ”B” together. This brings the heap into the state depicted in figure 3, right bevor two overlapping chunks are created. When ”C” is now freed, ptmalloc checks to see if there are any free adjacent chunks to consolidate them into one big free chunk. By clearing the `PREV_IN_USE` flag, ptmalloc recognizes that the chunk preceding ”C” must be free and uses the value stored in `prev_size` to find the chunk header to update its size. Since the value of this field has been forged to contain the combined size of ”A” and ”B”, ptmalloc tries to consolidate ”A” with ”C”, swallowing ”B” in the process. It is worth to note, that the `prev_size` field of ”C” and the size field of ”A” are not consistent at this point, however ptmalloc lacks the consistency check for this scenario and assumes the size of ”A” to be equal to the `prev_size` field of ”C”. This creates a free chunk, spanning from ”A” to ”C” embedding ”B”. Allocating a chunk of the appropriate size will return this combined chunk and yields full control over the contents of ”B”, regardless of ”B” being allocated or not, which concludes stage one of the exploit and can be seen in figure 4.

At this point, the second stage of the exploit begins, with the goal to leak the libc address. If the attacker can perform arbitrary overflows and overreads, he could also skip the first stage and start at this point. The newly allocated chunk, named ”D”, is used to change the embedded chunks (”B”) size to be of unsorted bin size. One additional requirement is that ”B” is followed by two valid chunks. This can be achieved by either choosing the size to line up with an already existing, valid chunk, or by faking them. Fakes only need to have a valid size field, so they can be simulated by two small chunk headers. This requirement also stems from the libc trying to consolidate multiple chunks upon free. To detect if the following chunk is free or not, the `PREV_IN_USE` flag of the next following chunk has to be read, thus two chunk header are required. Next the thread local free list must be filled, so that all slots are taken or the list for objects with the same size as ”B” is already saturated. Then ”B” is freed, and as there is no slot in the thread local free list left to put the chunk into, it is globally freed. Because of the mechanics detailed above, ”B” will now be part of an unsorted bin, and if that linked list was empty prior, both fields, `fd`, and `bk` contain pointer into the libc. As ”B” is fully contained in ”D”, ”D” can be used to read out the data from ”B” even when ”B” is freed and reveals the libc address to the attacker.

Here the third stage starts, attacker that additionally to arbitrary overflows and overreads already know the libc base address could begin exploitation at this point. From the libc address leaked prior, the address of the `__free_hook` is calculated. This `__free_hook` is a hook provided by the libc to load a different heap implementation at runtime by

```
55 struct arena_run_s {
56     /* Index of bin this run is associated with. */
57     szind_t      binind;
58
59     /* Number of free regions in run. */
60     unsigned     nfree;
61
62     /* Per region allocated/deallocated bitmap. */
63     bitmap_t     bitmap[BITMAP_GROUPS_MAX];
64 };
```

Listing 2: *Runs are used to manage the regions, which are the units of memory returned by jemalloc*

intercepting the library call free and instead executing a function provided by the application. As this `__free_hook` is not stored on the heap, the heap overflow must be escalated into a write anywhere primitive to enable an attacker to write chosen values at arbitrary memory addresses. This can be done by tricking ptmalloc to return an address specified by the attacker when performing an allocation. The introduction of thread local free lists simplified this task, as it misses crucial sanity checking when returning elements of that list. This step is also possible to perform on older versions of ptmalloc but is more complex as a valid size field is required in the correct location, which can be crafted but requires additional trickery. First ”B” is allocated again, and the size of ”B” is changed by updating its `size` field, so it is eligible for a spot in the thread local chunk list. Then it is freed, and with it entered into the free list, the `fd` pointer of the chunk struct gets filled with the address of the next free chunk of the same size. Since ”D” overlaps ”B”, the attacker has access to this list and can modify the `fd` pointer to point to the `__free_hook`, so the second next allocation of the same size as ”B” will return the chunk chosen by the attacker that contains the hook.

From the libc address leak in stage two the address of `system` can also be calculated and the value of the free hook can be replaced by this address. Any time free is called from then on, instead of releasing the memory a chunk holds, its contents is interpreted as a shell command and executed by the system function.

3.3 jemalloc

The second dynamic memory allocator analyzed in the thesis is jemalloc. It was intended to be a scalable malloc implementation for freeBSD[14] but found its way into Mozilla Firefox and Mariadb as well[17]. The version this thesis focuses on is from GitHub[1], the latest stable version (stable-4).

Jemalloc uses a different naming scheme for its management structure then ptmalloc. This is fine for the most part, but the chunk has a different semantic in jemalloc than it had in ptmalloc. In the context of jemalloc the units of memory returned by the allocator are called region. An array of same sized regions is managed in a run. The structure used to manage these regions can be seen in listing 2. It does not only store the number of free regions of the run in `nfree`, but also keeps a map where each bin corresponds to the status of a region. If a bit is set in `bitmap`, the corresponding region is free. Similar to ptmalloc, bins are used to manage runs that store regions of the same size, so that each

[1] https://github.com/jemalloc/jemalloc/tree/stable-4

```
312 struct arena_bin_s {
313     /*
314     * All operations on runcur, runs, and stats require that lock be
315     * locked.  Run allocation/deallocation are protected by the arena lock,
316     * which may be acquired while holding one or more bin locks, but not
317     * vise versa.
318     */
319     malloc_mutex_t       lock;
320
321     /*
322     * Current run being used to service allocations of this bin's size
323     * class.
324     */
325     arena_run_t     *runcur;
326
327     /*
328     * Heap of non-full runs.  This heap is used when looking for an
329     * existing run when runcur is no longer usable.  We choose the
330     * non-full run that is lowest in memory; this policy tends to keep
331     * objects packed well, and it can also help reduce the number of
332     * almost-empty chunks.
333     */
334     arena_run_heap_t    runs;
335
336     /* Bin statistics. */
337     malloc_bin_stats_t  stats;
338 };
```

Listing 3: *Bins are used to manage Runs for regions of the same size*

region managed by a bin has a uniform size. Each run is associated to exactly one bin using the `binid` field. There exist different types of runs that depend on the regions size. Small runs are for object smaller than the page size, the other ones are large.

Bins also keep track of the associated runs using the `runs` field and keeps track of statistics using `stats`. One of the runs part of the bin has a special role, as it is the run currently used to serve allocations. When that run has been used up and no more empty regions are available any more, it is exchanged for another, nonempty run. Swapping out the runs in `runcur` happens lazy, meaning the exchange is not triggered when the last element is taken out, but when an allocation can't be served any more.

Many bins of different sizes are stored in an arena, the top level structure in jemalloc. The number of arenas present in the dynamic memory allocator is configured at compile time. Each thread can be associated to one arena, resulting in the threat using that arena to statisfy its memory requirements. This mechanic is meant to spread the load of the allocator over multiple arenas, and as each arena is locked individually, lock contention can be reduced.

Arenas not only handle bins, but also do chunks. Unfortunately, the chunk in jemalloc has a entirely different semantic from that in ptmalloc. Chunks are an abstraction to mapping memory in the context of jemalloc. These chunks have all the same size fixed at compile time and are aligned to a multiple of their size. When a new run must be allocated, unused pages of a chunk are used to create the run. To keep track of all the runs that are part of the chunk and do some additional book keeping the structure in listing 4 is used. The `node` element is used to associate a chunk with an arena. While any chunk can only be connected to one arena, arenas can be associated with any number of chunks. The last field in the chunk, `map_bits` is of variable length and stores information, such as

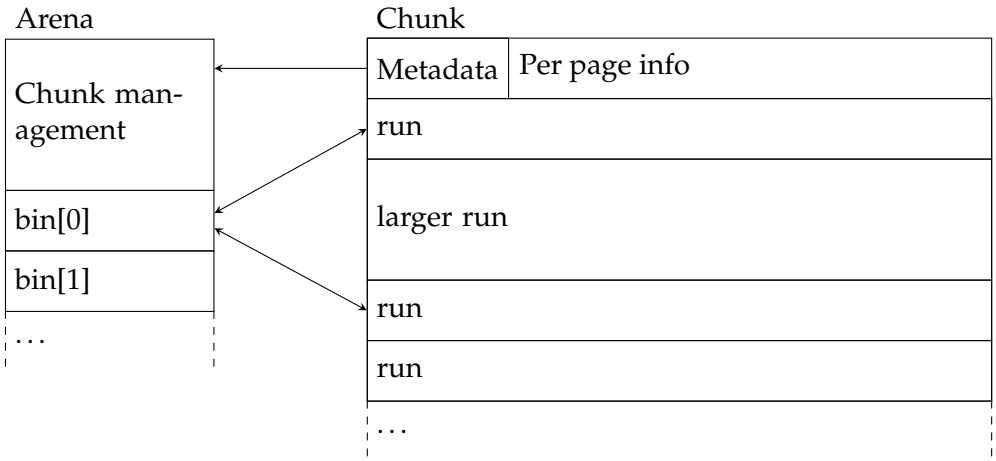


Figure 5: *Design of the jemalloc dynamic memory allocator*

```
185 struct arena_chunk_s {
186     /*
187     * A pointer to the arena that owns the chunk is stored within the node.
188     * This field as a whole is used by chunks_rtree to support both
189     * ivsalloc() and core-based debugging.
190     */
191     extent_node_t       node;
192
193     /*
194     * True if memory could be backed by transparent huge pages.  This is
195     * only directly relevant to Linux, since it is the only supported
196     * platform on which jemalloc interacts with explicit transparent huge
197     * page controls.
198     */
199     bool            hugepage;
200
201     /*
202     * Map of pages within chunk that keeps track of free/large/small.  The
203     * first map_bias entries are omitted, since the chunk header does not
204     * need to be tracked in the map.  This omission saves a header page
205     * for common chunk sizes (e.g. 4 MiB).
206     */
207     arena_chunk_map_bits_t  map_bits[1]; /* Dynamically sized. */
208 };
```

Listing 4: *Chunks are used to alloacte memory to split of runs*

how many pages a run spans, if it is allocated, or the pages being dirty. Missing from this structs definition is a second array, seen in listing 5, also containing information about each page. The `map_bits` array is used to keep track of the size of large allocations and bin id of small ones. Additionally, information is stored indicating if the page is allocated, unzeroed, dirty, or large. The meaning of bits in this bitmap can change depending on the run being small, large, and unallocated. The secon array, `arena_chunk_map_misc` is used for different purposes. First it manages the heap of available runs stored an arena, but it also contains the run header for small runs. The basic design of jemalloc is also presented in figure 5.

The last structures important to the design of jemalloc are these, used for thread local caching. Management structures for these tcaches are allocated using the same mechanisms that are used for serving allocations to the user and can therefor end up

```
156 /*
157 * Each arena_chunk_map_misc_t corresponds to one page within the chunk, just
158 * like arena_chunk_map_bits_t.  Two separate arrays are stored within each
159 * chunk header in order to improve cache locality.
160 */
161 struct arena_chunk_map_misc_s {
162     /*
163     * Linkage for run heaps.  There are two disjoint uses:
164     *
165     * 1) arena_t's runs_avail heaps.
166     * 2) arena_run_t conceptually uses this linkage for in-use non-full
167     *    runs, rather than directly embedding linkage.
168     */
169     phn(arena_chunk_map_misc_t)      ph_link;
170
171     union {
172         /* Linkage for list of dirty runs. */
173         arena_runs_dirty_link_t      rd;
174
175         /* Profile counters, used for large object runs. */
176         union {
177             void            *prof_tctx_pun;
178             prof_tctx_t     *prof_tctx;
179         };
180
181         /* Small region run metadata. */
182         arena_run_t          run;
183     };
184 };
```

Listing 5: *The map misc structure is used to form heaps of available runs and to store information about small runs*

```
83 struct tcache_s {
84     ql_elm(tcache_t) link;      /* Used for aggregating stats. */
85     uint64_t    prof_accumbytes;/* Cleared after arena_prof_accum(). */
86     ticker_t    gc_ticker;  /* Drives incremental GC. */
87     szind_t     next_gc_bin;    /* Next bin to GC. */
88     tcache_bin_t    tbins[1];   /* Dynamically sized. */
89     /*
90     * The pointer stacks associated with tbins follow as a contiguous
91     * array.  During tcache initialization, the avail pointer in each
92     * element of tbins is initialized to point to the proper offset within
93     * this array.
94     */
95 };
```

Listing 6: *Thread local caching is used to reduce lock contention even further*

following another allocation. The most important part of the `tcache_s` struct from listing 6 is the `tbins` field holding tcache bins, that can be seen in listing 7.

Like the regular bins managing objects of the same size, regions handled by thread local bins also have a uniform size. The pointer `avail` points to past an array of region pointer. When memory is requested, avail is accessed at a negative index and a corresponding pointer is returned.

Multiple attack vectors have been identified that can be used to achieve jemalloc allocating the same region multiple times. This could be done by overflowing into the run header or corrupting the chunks meta data. Also a write anywhere can be achieved by corrupting thread local caches[3, 4]. To demonstrate the validity of attacking thread local

```
83 struct tcache_bin_s {
84     tcache_bin_stats_t tstats;
85     int      low_water;  /* Min # cached since last GC. */
86     unsigned     lg_fill_div;    /* Fill (ncached_max >> lg_fill_div). */
87     unsigned     ncached;    /* # of cached objects. */
88     /*
89     * To make use of adjacent cacheline prefetch, the items in the avail
90     * stack goes to higher address for newer allocations.  avail points
91     * just above the available space, which means that
92     * avail[-ncached, ... -1] are available items and the lowest item will
93     * be allocated first.
94     */
95     void         **avail;    /* Stack of available objects. */
96 };
```

Listing 7: *Thread local cache bins manage a pointer to array of regions used for allocation*

caches, this thesis outlines an exploit that overwrites the hooks of an arena. For this attack the attacker needs only to overrun buffer additional to his base capabilities, and there have to be multiple threads present and the attacker can choose which thread is used to perform the action.

The goal of the first stage is to leak an arena pointer and the base address of a chunk. Conveniently both information can be found in the first sixteen byte of the chunk header, as the content part of `node`. Fortunately, this chunk header is stored in the beginning of chunks which are aligned to their size. The exploit starts by allocating a small region "A" using the main thread. This creates a `tcache_t` followed by the attackers allocation. As the attacker is restricted to overruns, this thread local caching structure is not of interest for the attacker. A second `tcache_t` structure however can be allocated after "A" when an allocation is done in a different thread. Allocating all regions of the run "A" is part of, the last allocation will use a region adjacent to the thread local cache management of the second thread. As seen in listing6 this structure starts with a linked list. However, this list is not accessed when an allocation is served using the thread local mechanic. The next three fields can be predicted or altered without consequences as well. The structure that follows is the `tcache_bin_t` from listing 7 that is actually used to manage cached regions. These `tbins` are ordered by size, resulting in the first one managing the smallest possible allocations, in this case eight bytes, followed by a one that manages sixteen bytes small allocations and so on. Before explaining which values these structures must take after the overflow it is important to take a close look at the machanic of allocating a small bin from a thread local cache. The field `avail` is used to point past an array of pointer to regions. When a region is allocated, avail is accessed at the negative offset with the absolute value of `ncached` and that pointer is returned. To get jemalloc to return a sixteen bytes wide region at the beginning of a chunk, a pointer `avail` must be found that, when accessed at negative `ncached` returns a pointer to that chunk. Luckily the chunk header itself can be used for that. As the second pointer in the `node` field of the chunk header also points to the chunk header, the last three bytes of `avail` can be overwritten to point to the chunk header plus sixteen. Depending on the exact size of the chunk up to 7 bits must be guessed correctly for this attack to succeed. For the version tested in this thesis however the chunk has a size of two megabytes which results in only three bit that have to be guessed, yielding a success probability of one in eight. When `ncached` is set to one, accessing `avail` at index minus one will yield the second pointer from the start of the chunk, which is a chunk pointer. It is important to use the tcache bin that handles sixteen

bytes small regions for this trick as this will then leak both, the arena pointer and the chunk pointer. The first `tbin` entry is trashed by the continues overflow. This however does not matter if no object of the smallest size is allocated by that thread.

The second stage builds on the first stage, as it uses the same overflow to further modify the thread local cache. As the absolute address of a chunk is known, that chunk can be sprayed with addresses pointing to the hooks of the arena that was leaked. When `avail` is overflown to point into the leaked chunk and hits one of the sprayed pointer, the hook array of the arena will be returned. The attacker now must fill the chunk with function addresses he likes to call and when chunks should be created or destroyed, the attacker defined function will be executed instead. It is worth to mention, that this is still less comfortable than overwriting `__free_hook` in ptmalloc, as jemalloc hands a pointer to a chunk as the first argument, not a pointer to a region.

3.4 Security flaws

Now that the ways memory is managed has been explained, it is possible to deduce weaknesses in the security of these dynamic memory allocators.

3.4.1 Not detecting invalid free calls

Trying to free some object that is invalid is a sure indication, that the program encountered some error. This error can be relatively harmless, such as a random bitflip in non error correcting ram. Other times, this error is the result of a logic error or even the result of an attack attempting to corrupt the state of the heap. The challenge for the dynamic memory allocator is to correctly deduce whether the object about to be freed is valid or not. There are multiple reasons why an invalid object may be passed to the allocator to be freed. Objects that are not returned from the allocator, such as pointer to global or local variables. The object is also invalid when it has already been freed. The danger in these cases does not only lie in the invalid call to free, but also in the existence of a pointer to a freed object, a dangling pointer.

It is important to catch these invalid free calls, as they can be used by attackers to perform memory corruption attacks and even in when not attacked, undefined behavior may occur. Unfortunately, jemalloc compiled in the release configuration does just assume the validity of the object and thus is totally unable to detect any form of invalid or even malicious call to free.

For ptmalloc, detection of these erroneous calls heavily depends on the object being freed and the current state of the heap. There is one requirement an object must always fulfill for the free call to accept it. It must be preceded by a valid chunk header, which in the most basic case boils down to an aligned size field. If the object can be put in one thread local free list, then there are no further checks performed, otherwise chunk header around the object must be consistent as well. Is the object of a small size, thus handled by the fist bin, the chunk must not be the same as the head of the free list of the fast bin.

A recent addition to glibcs ptmalloc were thread local caches. These tcaches aim to improve the performance of the heap by not freeing the memory globally and thereby dropping the requirement of locking the main arena[11]. Two threads could attempt to free the same object. While this is detectable by ptmalloc when no thread local caches are employed, it becomes undetectable if one of them can cache it thread locally, independent of the order of both frees. Thread local caching, implemented as in ptmalloc, results in

an inconsistent few of the heap state by different threads. The thread caching it, sees the chunk as being free, while it is still allocated for the other one, so the second thread may still free it without raising a heap error.

3.4.2 Leaking memory layout

Address space layout randomization (ASLR) was introduced to significantly increase the difficulty of successfully exploitation by adding entropy to the virtual addresses of libraries, stack, and heap. Since the introduction of position independent executables (PIE), also the virtual address of the program image is protected. While older systems with a small virtual address space, such as `x86` do not provide enough possibility to randomize addresses, systems with a 64 bit addresses space render attacks relying on guesses infeasible[30]. Saving pointer into other memory regions in locations accessible to the attacker provides an information leak vulnerability. Ptmalloc for example terminates its unsorted bin free lists using pointer into the arena. This address is also not necessarily cleared when the chunk is later acquired via malloc. For jemalloc, addresses to the main arena are stored in the meta data stored in the beginning of areas that encapsulate allocations.

As ASLR works on modules, these information leak vulnerabilities enable attacker to deduce the virtual address of any other symbol part of the same module. This knowledge can be used to increase the attack surface, as memory corruption attack can target additional symbols. The glibc for example uses the `__free_hook` so that applications can load their own heap implementation. As soon as the base address of the glibc is learned, the address of the `__free_hook` can be deduced. When the hook is not equal to `NULL`, it is interpreted as function pointer and called with the original argument provided to `free`. Corruption of this hook gives a convenient way of redirecting the control flow to an attacker defined address.

3.4.3 Missing overflow detection

Jemalloc allocates objects of the same size in runs that are arrays of allocable regions. In these runs no additional data or probes are inserted in between any two allocations. This makes detecting buffer overflows by the heap impossible. Ptmalloc stores the size of the allocation in front of the user data. This additional data does not provide significant entropy, especially to attacker able to influence allocations oh the heap. If an attacker knows the size of the object he tries to overflow into, he can compute the size field and overwrite it with its original value, hiding the overflow from the heap. Additionally, this size field proves to be an additional attack vector that can be used to alter heap behavior.

3.4.4 Heap management in predictable locations

The next security weakness jemalloc and ptmalloc have in common, is the possibility to predict the location of crucial heap management structs. Ptmalloc manages free elements of the heap by reusing the space of free chunks. While this makes efficient use of memory that is not allocated by the program, it also makes these structures accessible to attacker that can overflow the chunk or control its contents by other means. While jemalloc does not reuse its regions to store control data, it is laid out to have metadata in the beginning of each chunk. Stored in this metadata are not only the bins, but also information about the individual runs, and the arena the chunk is linked to. Not only is the position of the chunk header predictable, but also the structures used for thread local caching of regions.

These structures are allocated using the same allocation primitives as used for serving the programs allocations. This results in these thread local cache management structures ending up next to user allocations and being targetable by buffer overruns.

Using buffer overflows for ptmalloc or relative read write vulnerabilities, such as out of bound array access could grant an attacker access to these structures. When the control data stored there is modified, the attacker could force the heap to return invalid objects or alter the behavior in some other way. But not only corruption of these control structures poses a risk for the heaps safety, but the data stored there is also rich in information. Ptmallocs linked lists and jemallocs arena pointer and current run pointer for example leak heap and library addresses. As mentioned before, this information can be used to increase the attack surface and target other libraries or parts of the same library that have a large impact on the flow of execution.

3.4.5 Function hooks

Built into ptmalloc are multiple function hooks that can be used to intercept different common allocation and deallocation functions. The purpose of these hooks is to grant programs the ability to install custom dynamic memory allocators. Apart from using hooks there also exists a second technique to use a custom heap. The glibc exports all its dynamic memory allocator routines as weak function symbols. This indicates to the loader, that these symbols may be shadowed by other libraries and applications. This allows a program to replace the default heap implementation provided by glibc without the need for writeable function hooks. While jemalloc does not has hooks for the basic memory allocation and deallocation functions, it has hooks that govern the allocation and deallocation of chunks. These hooks are part of the arena and thus have read and write privileges.

These function hooks come with major risks. These hooks grant attacker a simple way to hijack the control flow and execute arbitrary code. It has been shown, that with minor constraint a single call into the glibc is enough to spawn a shell[26]. There even exist tools that search for these so called "one gadgets"[2]. To make matters worse, hooks like glibcs `__free_hook` are very hard to cover via control flow integrity checks. These hooks are meant to be overwritten by arbitrary functions not part of the original control flow of the library. No control flow graph can be computed when the library is compiled as the function that is used for the hook may not even been written yet.

3.4.6 Allocate anywhere possible

The last vulnerability identified is the possibility to force the dynamic memory allocator to return any attacker defined address when allocating an object. Ptmalloc uses linked lists to keep track of free chunks on the heap. When an attacker manipulates a node of this list he can introduce new objects into the free list that may not be valid. In case of tcaches or chunks handled by fast bins, this list is only single linked, and a partial overwrite can be enough to alter the location of future allocations. When the absolute address of a target is known, getting the heap to perform an allocation at that target is easily forced by overwriting the next pointer of a thread local cached free list.

It is important to notice, that the returned chunk can lay outside of the heap and any area ptmalloc manages. In the case of jemalloc, thread local caches can be used to achieve

[2]https://github.com/david942j/one_gadget

an allocate anywhere primitive, as demonstrated earlier. Thread local caches are managed using the same allocation mechanism as exposed to the program itself, which results in these structures being adjacent to user allocations. As the content of these structs is trusted blindly and they contain addresses that are handed out in future allocations, altering these pointers will result in jemalloc return regions at any address.

4 Design

To help understanding the design of the Secure Heap and the Leap, both use the same terminology as ptmalloc. A chunk is conceptually the smallest unit and is the memory returned by an individual call of malloc or a similar function. Chunks are logically grouped by bins, and many bins make up the arena. Both the secure heap and Leap consist of only one arena, so all bins are implicitly grouped in the main arena and further explanation of the arena is omitted.

In order to avoid flaws made by other dynamic memory allocator implementations, as described in section 3, we have to put special emphasis on some parts of the Design. This section will describe one secure heap implementation and show its security properties and also its infeasibility. Then with relaxed constraints a second heap will be designed and implemented.

4.1 Secure Heap

For a heap to be secure, the heap must not have any of the flaws listed in section 3.4. Avoiding these structural problems is the most important part in the design of the secure heap. Runtime overhead and memory consumption on the other hand are of no concern for this dynamic memory allocator.

The main design principle used for the secure heap is to strictly distinguish between the memory space used to manage all the internal data used by heap, and the memory that is handed to the executable program or other libraries. A boundary is created by limiting the type of information that can be stored by the memory management in areas that are used to serve allocations. Any Information that can be used to deduce the address space layout or to alter the internal state of the heap may never be stored in these areas of memory. And any information that is stored in band with the allocations has to be checked for validity to prevent attackers from altering this information. This strict segregation entails that when the heap itself requires dynamic memory it may not use the same routines that are used to serve the allocation of the program. Managing of allocations can be seen in figure 6. Allocations are tracked via a linked list and pointer are checked on free to see if they were the exact pointer returned by malloc. There is no information stored in the memory area returned by malloc and additionally a guard page with no access permissions is created following the allocation. Any two allocations are completely

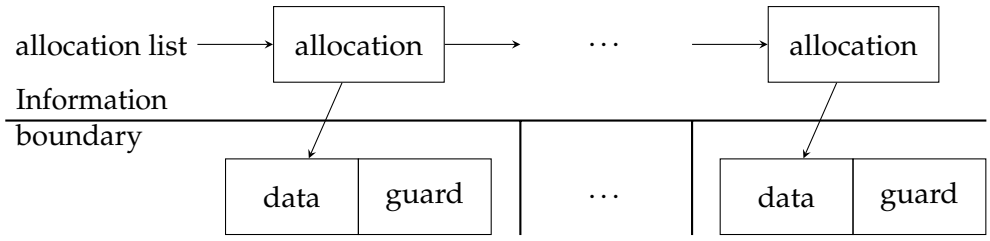


Figure 6: *Design of the Secure Heap*

separate in the sense, that they do not share the same mapping and there is no cross reference or dependency between these two.

To circumvent the problem of dangling pointer it is important to detect access to objects that lie outside of the lifetime of that object. The problem is to detect these accesses as they can look like a valid access to a different object residing at the same place in memory. One solution could be to search for pointer to every object being freed and only successfully free the object when no such pointer is found. This however is an impossible to solve problem as pointer are indistinguishable from integers that have the same value by chance[27]. The Secure heap circumvents this problem by never reusing memory and setting the memory protection of freed objects to be not accessible. From this decision rise two new problems that are connected. First memory protection can only be set on page granularity resulting in adjusting the size of the allocation to the next multiple of the page size. Not increasing the allocation size would result in multiple allocations on the same page and changing the protection status of that page would also result in loosing read write access to the other allocations. The second problem has an even bigger impact on the memory usage as not reusing allocations leads a memory leak. The pages have to remain mapped to prevent the Operating System to reuse the addresses in further memory requests, however they don't need to be backed by physical memory. Assuming the program creates a infinite series of mallocs and frees, this will result in virtual address space to be used up and at some point in time no more memory allocations by the program can be served any more.

This design allows the Heap to detect calls to free passing invalid objects, since the reference has to be part of the list of allocations. When the object that is attempted to be freed is in that list, the passed reference is valid and if not, it is invalid. Freeing the object requires searching for the element in the list and if none is found the invalid parameter is detected and the program is terminated.

The strict segregation between management structures and user data avoids multiple problems at once. No information about the address space layout is leaked as no pointer or any value derived from one is stored with an absolute address or fixed relative offset to any allocation. Not only the position of pointer and derived values cannot be predicted by the attacker, but also the heap management structures as well for the same reason. The problem implicitly avoided is the possibility of allocation anywhere primitives. With the management structures being unreachable by the attacker since their location is unknown and not in the same memory areas as the allocations. Hence to modify the heap structures, the attacker has to have information about the address space layout and be able to perform read and write operations at arbitrary addresses. For this reason, I argue that while the structures would technically allow for chunks to be allocated anywhere an attacker has no gain from using such primitive.

Overflow detection is achieved by placing each allocation on one page as far to the back as possible. By design the following page is a non-accessible guard page and any reading or writing access there will abort the program. Here it is important to take minimum alignment requirements into consideration as allocations have to sufficiently aligned for any build in type. The last security flaw also is avoided by design. The secure heap does not implement any hooks.

4.2 Leap

While the Secure Heap is mostly a concept to explore the possibility of techniques needed to create a secure heap while sacrificing usability, Leap aims to be a drop-in replacement for any heap conforming to the POSIX API. To achieve this some requirements have to be relaxed, like not reusing memory, as this would necessary lead to exhausting the virtual address space as discussed in the design of the Secure Heap. The second constraint relaxed is to mark memory that is freed to be not accessible. The memory protection mechanism in `x86` and `arm` work only on page granularity and granting or revoking memory access rights would also grant or revoke these for all chunks sharing the same page, and for the special case that one of these chunks spans more than one page this would lead to inconsistent access privileges for that chunk.

Not writing sensitive information to the chunk however is not relaxed and also information stored in these regions of memory have to be checked to detect any alteration of these information. This ensures not leaking the address space layout and thereby prevents an attacker from escalating the attack surface of the program from elements on the heap to elements in libraries. Additionally, as information cannot be altered the heap management cannot be disrupted by an attacker that can only corrupt memory of the heap. Modification of heap management data in the allocation regions, has to be detected by checking this information and also requires these information to be predictable and non-ambiguous.

The Leap discerns two different types of chunks, small chunks and huge chunks. All allocations smaller than four times the page size are served using small chunks, and all other allocations will use huge chunks. These chunks are grouped in bins, and for small chunks bins contain only chunks of the same size. This allows small chunks to be treated as an array of equally sized chunks and the allocation status can be tracked using a bitmap, as seen in figure 7. As huge chunks are individually mapped, there is not necessarily a corelation between any two allocations, so they are just grouped together in one bin. These bins are stored in an array. This allows for the array index to be stored with the allocation and does not reveal the address of the bin list but only its size. Storing this information with the allocation brings the advantage of not needing to search for the bin that owns the allocation and is also easily checked by testing, if the chunk that is currently handled falls into the memory range of chunks owned by the bin.

Bins of the same size are stored in a singly linked list. This simplifies finding the right sized bin for an allocation and in multi-threaded environments also improves the performance, as threads can lock individual bins, and other threads skip these locked bins when trying to allocate an object. This however requires the entire heap to be locked when changes to the linked list itself are made. These changes include creating new bins to accommodate further allocations, or bins being destroyed, when the last chunk is freed in the bin.

To reduce the time needed to look up the right bin on a free, the index of the bin that served an allocation is stored in front of each chunk together with a canary. The canary

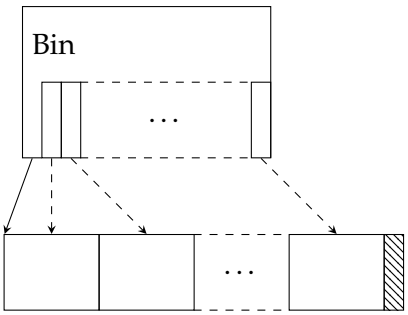


Figure 7: *Small bins manage arrays of chunks using bitmaps*

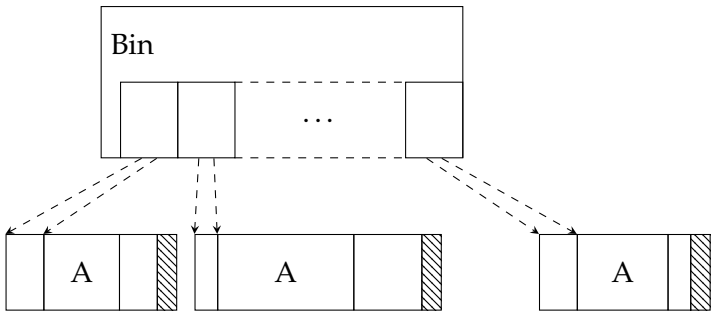


Figure 8: *Huge bins group multiple individual mappings of chunks trailed by guard pages*

can be used to detect buffer overruns when the attacker cannot leak it beforehand. Storing the owning bins index next the chunk does not reveal relevant information, as it only leaks the size of the bin list, but not its position. Leaking the position is not critical, as an attacker can realistically guess it, or have the list grow by allocating data to force a chosen bin to be allocated at some offset. Altering the bin index next to the chunk however could have consequences when the bin is freed. The heap would assume, that the chunk is part of another bin and subsequently mark a chunk of the wrong bin to be free, or in the worst case could result in an out of bounds access of the bitmap in the other bin. For this reason, this bin id stored next to the chunk is not to be trusted and it must be checked if the chunk is actually at a valid address for chunks of that bin.

Algorithm 1 Algorithm to allocate a small chunk

```
procedure ALLOC_SMALL(size)
    curBin ← findAndLockNotEmptyBinOfSize(size)
    index ← findAndClearFirstSetBin(curBin → inUseBitmap)
    ret ← nthChunk(curBin, index)
    if¬canaryOk(ret) ∨ ¬allZeored(ret)
        raiseError()

    release(curBin)
    return ret
end procedure
```

When a new chunk is requested by the program, first the heap decides if it will be served using a huge or a small allocation. When a huge allocation is used, the huge bin

list is searched for a nonempty bin, enough memory to hold the chunk and meta data is requested from the operating system. Information about the location, and size of the mapping is stored in the huge bin together with a reference to the chunk returned, which may reside at an arbitrary offset from the beginning of the mapping. In case of a small allocation, the algorithm 1 is used to not only deduce the chunk that is returned, but also to check the afore mentioned metadata. The algorithm assumes the size to be compensated for the size granularity and the overhead introduced by the chunk header. First a bin has to be found that isn't empty or currently used by another thread to allocate or free some chunk. Then the first set bit in the bitmap, holding information about the allocation state, is searched and cleared. This makes the allocation visible for threads querying the bitmap in the future. Then the chunk is checked to be in the state the heap expects it to be. The canary and the bin id have to be correct and the user data area has to be zeroed out. When these checks are passed, the bin is released and the chunk is returned to the caller.

Algorithm 2 Algorithm used to free a small chunk

procedure FREE_SMALL(chunk)
  $bin \leftarrow chunk \rightarrow bin$
  $index \leftarrow getIndex(bin, chunk)$
  if$\neg addressValid(bin, chunk) \vee \neg canaryOk(chunk)$
    $raiseError()$

  $clearUserData(chunk)$
  $lock(bin)$
  if$\neg isFree(bin, index)$
    $raiseError()$

  $setBit(curBin \Rightarrow inUseBitmap, index)$
  $release(bin)$
end procedure

Algorithm 2 shows the procedure for freeing a chunk and sanitizing the bin id. When a chunk is freed, the bin id and canary are extracted from the chunk header are extracted. Before any further action is performed on the chunk, the bin ids validity has to be checked first. To do this, not only a range check is performed, to see if the chunk falls into the address range handled by the bin, but also the chunk has to be at an address, chunks from that bin can start. If it is a valid chunk start, the chunk address minus the arrays base must divide the element size without remainder. The next step in checking the chunks header is to check the canary. In the current implementation, there only exists one global canary. If the value stored in the field differs from that canary, an error is raised and the program is terminated. After the chunk header has been validated, the user data portion of the chunk is cleared by overwriting it with Zeros. The idea is to not only bring the chunks content into a predictable state that can be checked on allocation, but also to destroy the data stored. Apart from reducing the risk of information leaks from access to uninitialized objects, this also aims at having the program fail faster in case of dangling pointer. As soon as the program tries to dereference a pointer stored inside an object that has been freed, the system will raise a segmentation fault. The reason for this is, that by overwriting the data of the chunk the pointer now points to the `NULL` address, that is never mapped on a modern Linux system.

5 Implementation

Before going over important parts of the heaps implementation, some general difficulties are addressed and how they are solved by this implementation of the heap. While the first difficulty is obvious, the circumstances it can occur under may not be so obvious. The problem are cyclic dependencies, specifically some functionality part of the allocation to depend on itself. The not so obvious problem here is, when using external functions, like `pthread_mutex_init` may do calls to the dynamic memory allocator, such as `malloc`, under some circumstances. If such function would be called in the context of creating a bin for chunks of size x, this creates an infinite recursion if `pthread_mutex_init`, trys to allocate a chunk of that size. For this reason the usage of external function is kept to a minimum and locking is implemented by the library as well, instead of using `pthread_mutext_t`.

The second problem stems from the loading process of dynamic libraries. Shared libraries may need to be initialized and that can be done using constructors. These constructors are functions called by the loader once the library has been mapped into memory and usually are used to initialize the library. This initialization however could need dynamic memory and thus try to allocate some. While this would be fine under normal circumstances, but when the dynamic memory allocator also depends on initialization via constructors, these constructors could be called in the wrong order and the heap may not be initialized when the other library tries to allocate memory. This problem is avoided by not depending on a constructor to initialize the library, but rather initializing it lazy. This means the heap is not initialized until the first call to an allocation function.

One of the problems many dynamic memory allocators face is the reliance on dynamic memory by themselves. The length of free lists in the case of ptmalloc grows linearly with the number of free objects and jemalloc relies on supplementary bit vectors to hold information about each run and their regions. These two allocators solve the problem by using memory mapped to serve allocations in a dual manner. Ptmalloc stores its linked lists managing free chunks in these chunks themselves. While jemalloc does not have memory potentially returned to the program in a dual use, it reserves a sufficient amount of space in the beginning of each memory region.

The dynamic memory allocators proposed in this thesis however do not store any data apart from a canary and a hint in any memory mapping that is meant to serve user allocations. These allocators therefore require additional infrastructure to accommodate growing data required to manage additional memory. For this reason, Leap uses a nested allocator design, where the allocator visible to the program is supported by a second one

private to the library. Strict segregation of user allocations and internal allocations allow the constraints for this internal heap to be relaxed. This allows this internal allocator to intersperse allocation and management data, and thus enables the heap to utilize a free list style design, like ptmalloc where the management data naturally grows with the size of the heap.

```
16 struct trusted_area {
17     uint8_t *begin;
18     uint8_t *cur;
19     uint8_t *end;
20     struct trusted_area *nxt;
21 };
```

Listing 8: *Struct to store information about areas owned by the internal allocator*

This internal allocator, called trusted allocator, manages larger areas of memory, called `trusted_area`. Each of these areas starts with a common header, which can be seen in listing 8. Stored in this header are all necessary information to deallocate it and split of memory in case it is not fully allocated yet. The field `cur` holds a pointer to the lowest address that is neither allocated nor part of a free list, covered later. In case the free list is empty this pointer is used to allocate objects from the arenas end. The third field of the `trusted_area` is a pointer, pointing to the first address past the memory mapping. This pointer is used to determine the size total size, but also the size left of the memory area. There are two useful other options to store the size. Either the total size of the area could be saved there, or the remaining size could be saved. To deduce the total size from the remaining size, first the used size would have to be calculated from the base pointer and the `cur` pointer, then the two sizes could be added. This would arguably not add measurable overhead, as the total size is only needed when the area is freed. When objects are allocated using the `cur` pointer, this would save one pointer subtraction, but add writing to one additional memory cell, as the end pointer does not need to be updated. The other option, storing the total size would not yield a benefit either, as the remaining size would have to be calculated each time requiring pointer subtraction as well.

```
27 struct used_area {
28     size_t size;
29     uint8_t data[];
30 } PACKED;
31
32 struct free_area {
33     size_t size;
34     struct free_area *nxt;
35 } PACKED;
```

Listing 9: *Structs used by the internal allocator to manage allocated and free chunks*

When a bin is destroyed, its bitmap must be freed as well. This can result in arbitrary ranges of the area used by the trusted allocator to be freed. The internal heap keeps track of these regions and also these that are in use by utilizing the structures seen in listing 9. Both structs are meant to be placed such that the `size` field precedes the allocation followed by either the data or a pointer to the next free region. Ideally both structs would be combined in a union, but variable length arrays are only allowed as the last field of a struct. Unfortunately, having such variable length members inside a union would lead to an incomplete type error. To mitigate this, both structs are marked as packed, telling the compiler to not add padding to the struct and thus arranges these structs such that `data` is at the same offset as `nxt`.

```
29 struct bin {
30     enum bin_type type;
31     void *data;
32     // index of next bin. Not using pointers enables resizing by remapping
33     size_t nxt;
34     /**
35      * the union ensures that all bin types have the same size,
36      * so it can be efficiently allocated as an area with different
37      * bin types interleaved
38      */
39     union {
40         struct small_bin {
41             lock_t lock ACQUIRED_AFTER(core_lock);
42             uint64_t *in_use GUARDED_BY(this->lock);
43             uint32_t esize;
44             uint32_t nelem;
45             size_t first_free GUARDED_BY(this->lock);
46             size_t free_eles GUARDED_BY(this->lock);
47         } small;
48         struct huge_bin {
49             lock_t lock ACQUIRED_AFTER(core_lock);
50             size_t nentry;
51             size_t nfree GUARDED_BY(this->lock);
52             struct huge_allocation *allocs GUARDED_BY(this->lock);
53         } huge;
54     } detail;
55 };
```

Listing 10: *The `struct bin` used to manage groups of objects in bin.h*

To serve the programs allocations a different allocator is used. It manages arrays of same sized chunks using `struct bin`. All bins handling chunks of the same size form a linked list, with the list head being part of the global data of Leap. This list however does not use pointer to indicate the next element, but rather an index. This can be done since bins not only form a linked list, but also an array that is contained by an individual memory mapping. This, together with not using pointer to link the individual elements allows for efficient resizing the array of bins and thus allocation of new `bin` structs. Reallocation is done via remapping the memory so that the array can grow or shrink. The field storing the index of the next list element is named `nxt`.

The `type` field is used to distinguish between the different of bins. As mentioned when describing the design of Leap in section 4.2, there are two different bin types, small and huge. The field `type` can also hold two different values. The first other value is `BIN_TYPE_FREE` and indicates, that the element is currently not used and part of a free list of bins. When a new bin is required the first element of this bin free list is taken and the bin indexed by `nxt` becomes the new list head. The other value is `BIN_TYPE_NONE` and is used to indicate that this element of the bins array has not been used yet. This enables the elements of the `bin` array to be initialized lazy. A dedicated value indicating that the bin has not been initialized yet is necessary because the memory returned when pages are mapped into memory is defined to be zeroed out. If there was only one value to indicate a free element, this would that, either all bins had to be initialized, or the `nxt` field is not suitable to hold the free list.

What element of the `union detail` is used depends on the `type` field. Managing small and huge bins in the same struct using a union results in the management structures for both bin types to be of the same size. This makes storing these bins as an array simple, not requiring pointer arithmetic. Both, the small and the huge bins contain a `lock` inside their detail structures. These locks are placed there to enable clangs static thread safety analysis

on the code. Placing them in the parent struct has the result, that in the specialized structs attributes, such as `GUARDED_BY(this->lock)` would be illegal, as this would reference the detail struct and not the `struct bin`.

For the small bin, `esize` holds the size of each chunk, including the chunk header. The field `nelem` stores the total amount of elements held by the bin. These numbers are used to calculate the size of the memory mapped region if the bin is to be destroyed. The number of free chunks in this bin are stored in the `free_eles` field. A bitmap allocated using the internal allocator is used to keep track of the allocation state of the individual chunks. For fast access to this bitmap `first_free` holds the smallest index a free chunk can occur at. It is updated when a chunk is freed, and the chunks index is smaller than `first_free`, or when an element is allocated. The bitmap itself is stored in the `in_use` field.

```
23 struct huge_allocation {
24     void *base;
25     struct chunk *chunk_ptr;
26     size_t size;
27 };
```

Listing 11: *The `huge_allocation` struct is used to manage allocations larger than four times the page size*

When more the four pages worth of memory is requested in a single allocation, it is handled using a huge bin. These allocations are served by mapping the requested amount of memory followed by the guard page. The operating system chooses the virtual address the memory will be mapped at, so each of these huge allocations must be managed individually. The structure that does so is called `huge_allocation` and stores the address returned by the operating system in `base`, and the size in `size`. Huge allocations are also preceded by the chunk header, which is placed into memory, so that the allocation is aligned to the size of a cache line or a specified alignment in cases of `aligned_alloc` or `posix_memalign`. To be able to handle this non constant offset into the mapping, the `chunk_ptr` is used. One huge bin doesn't store just one, but an array of these huge allocations in the `allocs` field. Like the `free_eles` variable in the small bins context, `nfree` keeps track of the number of free entries in the `allocs` array. The last field, `nentry` holds the amount of entries the `allocs` consists of.

Huge bins are also utilized to handle allocations requesting special alignment. While chunks are naturally aligned to some degree, some operations require alignment to larger powers of two. Vectorized instructions of the `x86_64` Instruction Set Architecture for example include the `MOVDIR64B` instruction requiring 64 bit of alignment. To ensure such instruction can be used the program can request specifically aligned memory. To serve these allocations, the alignment requirement is added to the size of the allocation when doing the huge allocation. This ensures, that at least one address within the allocation is a multiple of the alignment and is followed by enough memory to serve the request. Before this huge allocation is returned to the program however, the chunk pointer has to be adjusted to this address, so that the allocation does not only contain an address sufficiently aligned, but starts at one.

```
64 struct chunk {
65     uint64_t canarie_bin_id;
66     uint8_t data[];
67 };
```

Listing 12: *Structure containing user allocations in bin.h*

To associate each chunk with its bin, every allocation is handled by the `struct chunk`. It has only two fields, with `data`, the second, variable size field, containing the user data

that can be stored in the chunk. The other field `canarie_bin_id` serves two purposes. It holds a canary, a random value generated at runtime that is meant to detect buffer overflows. In the current implementation, this canary is the same for all chunks of one bin. The second piece of information it holds is the index of the `struct bin` managing this chunk.

6 Evaluation

This section evaluates our work. In the following we will conduct benchmarks to estimate applicability of our work to real-world problems.

To generate reproducible benchmarks, the same computer has been used to perform the benchmarks. This computer is equipped with a solid-state drive and has an Intel Core i7 6700 clocked at 3.4 GHz with four cores and 8 threads as the CPU. The underlying operating system is Debian stretch. To evaluate the performance of leap, two different benchmarks are used testing the performance of dynamic memory allocators in different scenarios. Jemalloc and ptmalloc are also tested using the same computer and benchmark. The first benchmark, called Super Smack[3], performs a stress test selecting and updating rows in a 90,000 element sized PostgreSQL database. While the amount of queries each client performs is fixed, the number of threads attempting queries in parallel is increased in steps of five. Figure 9 shows the results from this benchmark. All three allocators perform similar, with the number of queries per second first steeply rising. Then the performance levels off at around 20 threads before slowly declining until they stabilize around 26000 of jemalloc and ptmalloc and 22000 for leap respectively. The relative performance of Leap to Ptmalloc starts with 90 percent at 5 threads and drops to 77 percent at 40 threads. After that it climbs back to 84 percent at 85 threads where it remains steady for the last tests. The average performance is 83 percent of ptmalloc.

The second benchmark is not a real server but aims to simulate one. It utilizes multiple threads where allocated objects are passed between threads and each thread randomly frees some objects and allocate new ones to take their place. This resembles a realistic workload in the context of multithreaded servers[19]. In this benchmark the number of allocation and deallocation units is increased in each step while the runtime of ten seconds remains constant. Here ptmalloc and jemalloc perform similar with regards to the curve that is formed, namely first improving the performance until four units are used, then the performance drops back to the single performance for ptmalloc and slowly but steadily decreases for jemalloc. Interestingly the performance of Leap is almost constant over the entire range of units.

Both benchmarks also are used to determine the performance of SuperMalloc[17], a memory allocator that has some similarities with jemalloc and makes heavy use of hardware transactional memory.

[3] https://github.com/winebarrel/super-smack

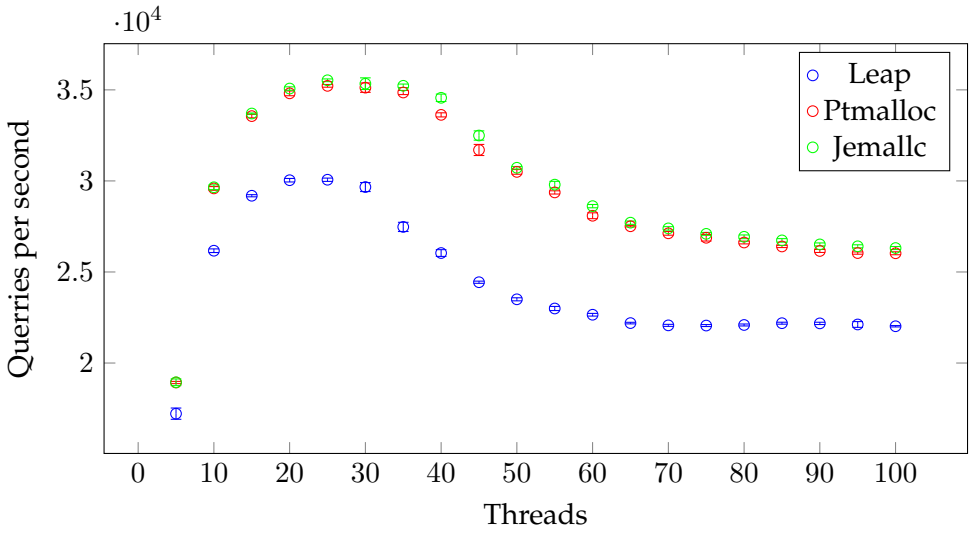


Figure 9: *Super smack benchmark stress testing postgresql with different heap implementations*

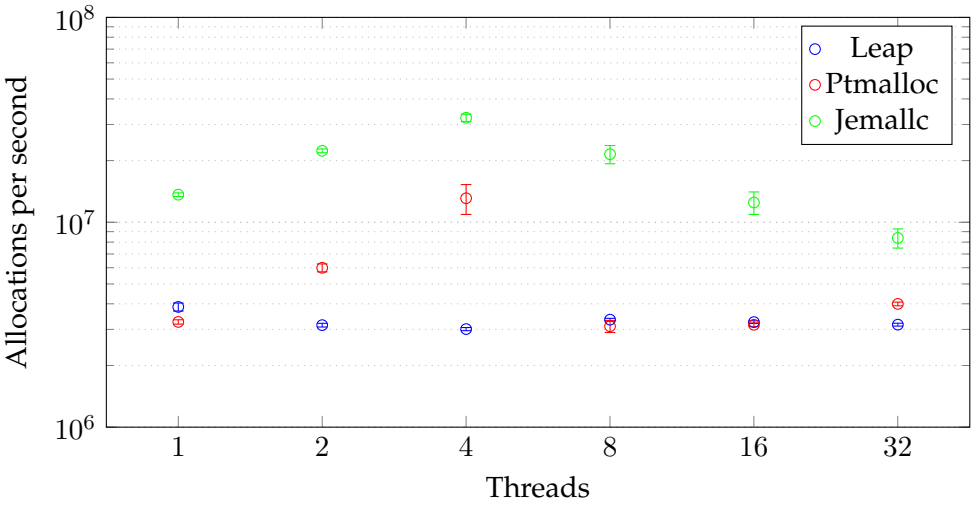


Figure 10: *The Mallo-test benchmark tests a producer consumer scenario, where the allocation and deallocation of objects is done in different threads*

7 Discussion

After describing work related to this thesis, possible improvements of the Leap dynamic memory allocator are outlined. Then this section is finished by drawing a conclusion on the applicability of the heap designed in this thesis.

7.1 Related Work

There is a substantial amount of different work done towards overflow prevention and detection. The class of overflow detection can be further divided into approaches *preventing* the memory corruption to appear and approaches detecting the symptoms of memory corruptions *after* their occurrence. Control-flow integrity falls into the latter category: By calculating all possible control flow transfers and thus creating a control flow graph (CFG), it can detect attacks that alter the control flow and take branches unfeasible in the original program. This effectively detects attacks using return-oriented programing (ROP) or counterfeit object oriented (COOP) programming. By design such approaches can not protect against attacks that stay within the allowed CFG [1]. Unfortunately exploiting structures of the dynamic memory allocator does not violate any constraints imposed on the control flow.

Furthermore, enforcing control flow integrity comes at a (sometimes) significant performance cost and thus different solutions with a varying degree of relaxation have been published. They can be divided into coarse grained CFI [33, 34], and fine grained CFI [31, 21, 22] approaches. The difference between fine and coarse grained CFI is the amount of overapproximatiion done when constructing the CFG. Unfortunately, constructing a precise CFG depends on a precise points to analysis which is undecidable [27]. It has been shown that even when the execution path is constrained to follow a valid path of such control flow graph, exploitation can still be successful [13].

StackGuard [10] can also be counted towards CFI enforcement. It works by placing a secret value in front of the return address, and when an attacker attempts to overwrite the return address saved on the stack, he has to also overwrite this secret value. This modification can be detected before the return address is read back from the architectural stack and instead of jumping to a (potentially) attacker-defined location, program execution is terminated. This technique, while only introducing minute runtime and memory overhead only detects stack based, continuous buffer overruns. Attackers who can access an array on the stack out of its bounds could skip this secret value and change the return undress

without alerting the detection mechanism. This secret value also has a disadvantage when the attacker is able to leak it, due to it being constant for a given run. The value can not only be leaked directly by printing, but also indirectly by guessing it step by step. This does however only work if the program creates a new process with the memory being cloned, like it is done in the link system call fork. In this cases it is impossible to re-randomize the secret value since the old value may be already used to protect some return address further down on the stack.

While both, the dynamic memory allocator and enforcing the integrity of the control flow have the goal of reducing the effective attack surface, the methodology of both approaches is fundamentally different. The secure heap and Leap both try to reduce the attack surface by moving interesting attack targets to memory locations that are more difficult to reach and removing mechanics that are prone to exploitation. Control flow integrity techniques, on the other hand, try to achieve the same by guarding these sensitive variables and mechanics. Similar to Leap, there are also attempts made to remove sensitive information from the stack. In case of the shadow stack protection mechanism, control structures like stack frame information and return pointers are saved on a secondary stack.

While control flow integrity does guard control structures, it does not prevent the memory corruption but rather detects it. There are other approaches that do not target these control structures specifically but try to detect overflows and access violations more generally. The solutions providing overflow detection differ in their goal and the kind of overflow detected. LibsafePlus [5] for example has the goal to detect stack based buffer overflows that would corrupt return or frame pointer. This is done by first gathering information about buffers in the original program, that has to be compiled with debug information. The gathered information are then added to the executable via binary rewriting. When the program is executed, LibsafePlus is preloaded into the memory and installs wrapper functions that intercept calls to dangerous library functions, such as `strcpy`. To prevent buffer overruns, the buffer sizes are compared, and the program is terminated when it attempts to write out of bounds. The sizes of stack buffers are overestimated to reach to the end of the stack frame which is determined by the frame pointer. For buffers on the heap, a red black tree is maintained containing the base addresses and sizes of all dynamic memory allocations [5]. A similar technique has been implemented by Lhee and Chapin that uses a modified compiler instead of extracting buffer information from debug information [20]. A drawback of this approach is the limited scope: Buffer overflows are only detected when library functions are used to overflow. Also, programs may have to be recompiled to use these techniques which requires the presence of source code.

While these schemes aim to protect valuable information and control structures on the program stack other solutions have been developed specifically hardening dynamic memory allocators. One such technique follows the same principles as StackGuard augmenting the chunk header of dlmalloc, a predecessor of ptmalloc, with a canary. This canary guards the size fields by hashing them using a global secret and adding that value to the chunk structure [28]. While this canary can protect from continuous overflows, it can be defeated in multiple ways: First, the canary is only checked when a heap operation is performed, which renders exploits that overflow this canary undetected until the victim chunk is passed to a heap management function. This point is not necessarily reached during exploitation [24]. Also, only the size fields are checked accessing arrays out of their bounds gives a read or write relative to the arrays base and such overwriting the canary can be skipped and the linked list be corrupted. Stage one of such an exploit is sketched in section 3.2.

Building on the insufficiency of canaries and separation of heap meta data from the allocation regions DieHarder[24] has been developed on the base of the DieHard[6] heap. DieHard takes a probabilistic approach to memory safety that tries to approximate a Heap similar to the Secure heap described in this thesis. Objects are placed at a random relative offset. Additionally, the program can be executed multiple times in parallel, with each instance other than the first one being a replica. The output from all replicas are compared and when the output deviates, DieHard decides which output shall be forwarded. Each replica not conforming to this output is terminated as this deviation indicates an error. DieHarder extends this implementation in terms of security by being less forgiving and also applying a sparser memory usage increasing the probability of buffer overflow exploits hitting an unmapped page. This technique significantly improves resistance against heap spray attacks, where a large amount of allocations and overflows are used to mitigate an unpredictable heap state.

A more general approach is taken by CRED [29] that introduces bound checking for buffers via binary instrumentation. This allows CRED to detect buffer overruns not only when a C library function is called. Unfortunately, this comes at a substantial performance overhead. While cred checks buffer accesses at runtime, another technique has been described that ensures memory safety of C programs via source code transformations [32]. Transforming programs written in C into memory safe programs unfortunately comes at a huge run time penalty for both static transformation and instrumentation. The performance overhead is larger than 100 percent in many cases. Source code transformation also is not always applicable as third party libraries may only be shipped in a binary format.

To counteract this performance deficit, Kurznetsov et al. proposed to only protect pointers that are important to the control flow. They propose to detect these pointer via static analysis at compile time with the *is sensitive* property being transitive. This means, not only those pointers are sensitive that directly impact the control flow, but also those that might point to sensitive pointers. All those sensitive pointers are then aggregated and stored in a separate section of the program that is not directly referenced or accessed. On `x86` accessing this section is done vie dedicated segment registers and on `x86_64`, with these segment registers not being available any more, an indirection via special stubs is done. This concept has been named Control pointer integrity (CPI) and adds a runtime overhead of about 8 percent. Dropping the transitivity when sensitive pointers are detected reduces this approach to code pointer separation (CPS) reducing the overhead to two percent [18].

7.2 Future Work

There are many aspects of Leap that can be improved upon. The main goal of Leap was to reduce the attack surface provided by the management structures and to see if and what kind of information can be allowed to reside in the same memory mappings as the allocations handed to the program. While exploiting these structures to gain overlapping chunks or allocations at arbitrary locations is not realistically any more as an attacker capable enough to perform these attacks could corrupt the target memory locations directly. The next step in improving the security would be to protect the allocated objects themselves. To achieve this, the chunk returned on allocation could be randomized to force an attacker to either guess correctly or spray the heap and overflow repeatedly. To mitigate the first scenario, the randomization has to have a sufficient amount of entropy, to make such an attack impractical. The second scenario relies on detecting the attack as it is

going on. A probabilistic approach could be taken here as well by reducing the bin's size. This would increase the number of allocations followed by non accessible memory and thus also increase the probability of accidentally overflowing into this non accessible area. Another possibility would be to randomly check foreign chunks for corruption when any function of the dynamic memory allocator is invoked. An improvement to that would be to have a dedicated thread checking each chunk header periodically in an asynchronous manner.

Not only the security of the Leap can be improved, but also its performance. Here different strategies could be tested. First, instead of one lock per bin, the total amount of locks could be reduced to one lock per linked list (chain) of bins. On first glance, this would increase the amount of contention on the lock. This can be counteracted by allowing multiple chains for the same bin size. In the current implementation all bins of the same size are put into the same linked list. Associating threads or even CPUs to a chain similar to how arenas are associated to threads in *jemalloc* could counteract this increase in contention. Binding chains to a CPU on the other hand has the advantage that cache performance could improve as the locks are not cached in different cores, overhead required to keep the caches consistent could be avoided. Coincidently this overhead is what makes locking instructions expensive in terms of runtime overhead[17].

Not only the runtime performance can be improved, but also the memory usage. The current implementation allows for a vast amount of different bin sizes which results in less space being wasted behind the object but when only few objects of a specific size are needed, the entire rest of the bin will stay unused. To reduce this issues of underutilizing the bins, the amount of space left free in the chunk could be approximated by a linear function. This would allow to use less bins to manage a larger number of chunks with a greater range of sizes. Reducing the number of bins could also reduce the overhead from mapping memory for the bins created as less bins would be needed. Second, the way aligned allocations are performed could be improved to not require huge allocations. Using masks on the bit vectors in bins could be used to filter out all chunks that are not suitably aligned with linear performance overhead with respect to the bit vectors length.

Another subject for investigation comes from the benchmark itself. Here it is very surprising how *jemalloc* dominated the microbenchmark where allocations and frees are done in a close loop but had no significant advantage over *ptmalloc* when testing the PostgreSQL server using Super Smack. To explain the huge difference between the results of both benchmarks, allocations and deallocations should be recorded for various programs to detect patterns. This could lead to heap implementations that adapt to programs on the fly when such pattern is detected. Adapting to the special needs of specific programs could yield substantial improvement of the allocation performance.

7.3 Conclusion

Memory corruptions are one of the basic exploitation mechanics and are a persistent problem dating back more than two decades to Aleph One's iconic work "Smashing the Stack for fun and profit"[2]. Since then the technology has improved in many ways, not only became processors faster, but also many security mechanisms were added. Nowadays memory pages can have access permissions, preventing the execution of Data or modification of code. The address space layout is randomized to make attacks that rely on guessing any address inevitable. To further protect return addresses and frame pointers randomized canary values have been widely adopted. All these improvements

work together to minimize the threat of memory corruptions, that can only occur because the languages used until today, such as C and C++, are not inherently memory safe. While it is not feasible and maybe even not possible to port all existing programs to some safe language, such languages should be avoided in new projects.

In order to mitigate the potential damage of attacks involving memory corruptions there are two basic principles that can be followed. First, attempts could be made to detect these corruptions. A primary example here is StackGuard. Buffer overflows have to overwrite a secret value in order to also overwrite the return address on the stack. This secret value is checked before the return address is used. The chances of guessing the secret value are small and thus an attacker not knowing this secret only has minute chances of succeeding with exploitation. Another way to foil attacks involving memory corruption is to place structures that are important in places that can not be reached by the attack. Here the separation of local variables and return addresses using a shadow stack are one example, CPI another.

The goal of this thesis was to analyze currently used heap implementations and to identify weaknesses in their management routines and structures. It has been shown that current popular implementations of dynamic memory allocators provide insufficient security. Storing metadata in places an attacker can predict, such as keeping free lists in the free elements themselves or having a block of meta data in front of *jemalloc*'s chunks can be used to leak information and to increase the attack surface. This thesis provides evidence that when meta data is stored along with the allocations, integrity of such meta data has to be easily verifiable and should not leak sensitive information, such as virtual memory addresses. Building on this principle, we designed and implemented a new dynamic memory allocator: the Leap. While it can not yet directly compete performance-wise with *ptmalloc* and *jemalloc* in real world scenarios, it is not too far off, and is even able to perform better then *ptmalloc* in some cases. With the current implementation of Leap only being a prototype, it is possible that it can be improved to match the performance of the other allocators more closely. Last, I want to argue that the default dynamic memory allocator in the C library should put emphasis on safety as it will be used by many different programmers with varying levels of skill. With buffer overflows still being an important threat, it can be concluded that correct memory management is not easy. The heap is one of the central elements of memory management and therefore should conform to a high standard of safety.

References

[1] Martín Abadi et al. "Control-flow integrity". In: *Proceedings of the 12th ACM conference on Computer and communications security*. ACM. 2005, pp. 340–353.

[2] One Aleph. *Smashing the stack for fun and profit*. `http://phrack.org/issues/49/14.html`, visited 2018-10-14. 1996.

[3] huku argp. *Pseudomonarchia jemallocum*. `http://www.phrack.org/issues/68/10.html`, visited 2018-10-14. 2012.

[4] Patroklos Argyroudis and Chariton Karamitas. "Exploiting the jemalloc memory allocator: Owning Firefox's heap". In: *Blackhat USA* (2012).

[5] Kumar Avijit, Prateek Gupta, and Deepak Gupta. "TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection." In: *USENIX Security Symposium*. 2004, pp. 45–56.

[6] Emery D Berger and Benjamin G Zorn. "DieHard: probabilistic memory safety for unsafe languages". In: *Acm sigplan notices*. Vol. 41. 6. ACM. 2006, pp. 158–168.

[7] Nathan Burow et al. "Control-flow integrity: Precision, security, and performance". In: *ACM Computing Surveys (CSUR)* 50.1 (2017), p. 16.

[8] Shuo Chen et al. "Non-Control-Data Attacks Are Realistic Threats." In: *USENIX Security Symposium*. Vol. 5. 2005.

[9] J. Cohen. *RELRO: RELocation Read-Only*. 2011.

[10] Crispan Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *USENIX Security Symposium*. Vol. 7. 1998.

[11] DJ Delorie. *malloc per-thread cache: benchmarks*. `https://sourceware.org/ml/libc-alpha/2017-01/msg00452.html`, visited 2018-09-06. 2017.

[12] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. ISSN: 0018-9448. DOI: `10.1109/TIT.1983.1056650`.

[13] Isaac Evans et al. "Control jujutsu: On the weaknesses of fine-grained control flow integrity". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 901–913.

[14] Jason Evans. *A scalable concurrent malloc (3) implementation for FreeBSD*. 2006.

[15] *glibc 2.27*. `https://www.gnu.org/software/libc/`, visited 2018-09-20. 2018.

[16] *How2heap*. `https://github.com/shellphish/how2heap`, visited 2018-10-01. 2018.

[17] Bradley C Kuszmaul. "SuperMalloc: a super fast multithreaded malloc for 64-bit machines". In: *ACM SIGPLAN Notices*. Vol. 50. 11. ACM. 2015, pp. 41–55.

[18] Volodymyr Kuznetsov et al. "Code-Pointer Integrity." In: *OSDI*. Vol. 14. 2014, p. 00000.

[19] Per-Åke Larson and Murali Krishnan. "Memory allocation for long-running server applications". In: *ACM SIGPLAN Notices*. Vol. 34. 3. ACM. 1998, pp. 176–185.

[20] Kyung-suk Lhee and Steve J Chapin. "Type-Assisted Dynamic Buffer Overflow Detection." In: *USENIX Security Symposium*. 2002, pp. 81–88.

[21] Ali José Mashtizadeh et al. "Cryptographically enforced control flow integrity". In: *arXiv preprint arXiv:1408.1451* (2014).

[22] Vishwath Mohan et al. "Opaque Control-Flow Integrity." In: *NDSS*. Vol. 26. 2015, pp. 27–30.

[23] D. Moore et al. "Inside the Slammer worm". In: *IEEE Security Privacy* 99.4 (2003), pp. 33–39. ISSN: 1540-7993. DOI: 10.1109/MSECP.2003.1219056.

[24] Gene Novark and Emery D Berger. "DieHarder: securing the heap". In: *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 2010, pp. 573–584.

[25] Phantsmal Phantasmagoria. "The malloc maleficarum". In: *Bugtraq mailinglist* (2005).

[26] *Pwning (sometimes) with style*. https://j00ru.vexillium.org/slides/2015/insomnihack.pdf, visited 2018-10-14. 2015.

[27] Ganesan Ramalingam. "The undecidability of aliasing". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), pp. 1467–1471.

[28] William K Robertson et al. "Run-time Detection of Heap-based Overflows." In: *LISA*. Vol. 3. 2003, pp. 51–60.

[29] Olatunji Ruwase and Monica S Lam. "A Practical Dynamic Buffer Overflow Detector." In: *NDSS*. Vol. 2004. 2004, pp. 159–169.

[30] Hovav Shacham et al. *On the Effectiveness of Address-Space Randomization*.

[31] Caroline Tice et al. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM." In: *USENIX Security Symposium*. 2014, pp. 941–955.

[32] Wei Xu, Daniel C DuVarney, and R Sekar. "An efficient and backwards-compatible transformation to ensure memory safety of C programs". In: *ACM SIGSOFT Software Engineering Notes* 29.6 (2004), pp. 117–126.

[33] Chao Zhang et al. "Practical control flow integrity and randomization for binary executables". In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 559–573.

[34] Mingwei Zhang and R Sekar. "Control Flow Integrity for COTS Binaries." In: *USENIX Security Symposium*. 2013, pp. 337–352.

Appendix

The program used to simulate an attacker with different capabilities, but full access to the heap API and full control over an objects content

```c
#define _POSIX_C_SOURCE 200112L
#define _ISOC11_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdint.h>
#include <sys/types.h>

#define NUMBUCKETS 100

typedef struct note_s {
    char *text;
    size_t len;
    struct note_s *next;
} note;

note notes[NUMBUCKETS];
note *free_notes = notes;
note *back = &notes[NUMBUCKETS];

void init() {
    // setting stdout and stdin to be unbuffered
    setvbuf(stdout,0,_IONBF,0);
    setvbuf(stdin,0,_IONBF,0);
}

unsigned long readlong() {
    char buf[40];
    char *tmp;
    unsigned long ret;
    ssize_t len;
    len = read(0, buf, sizeof(buf) - 1);
    if (len <= 0)
        exit(0);
    buf[len] = 0;
    ret = strtoul(buf, &tmp, 0);
    if (tmp == buf) {
        puts("invalid");
        exit(1);
    }
    return ret;
}

void print_menu() {
    puts("1. malloc");
    puts("2. realloc");
    puts("3. free");
    puts("4. calloc");
    puts("5. aligned_alloc");
    puts("6. posix_memalign");
    puts("7. read");
    puts("8. write");
    puts("9. exit");
}

unsigned long long getull(char *text) {
    char buf[0x20];
    size_t len;
    printf("%s", text);
    len = read(0, buf, sizeof(buf) - 1);
```

```
63      buf[len] = 0;
64      return strtoull(buf, NULL, 0);
65  }
66
67  void perform_malloc() {
68      size_t index, len;
69      if (free_notes == back) {
70          puts("error, no free bucket left");
71          return;
72      }
73      index = free_notes - notes;
74
75      len = getull("size: ");
76      // TODO replace if needed
77      free_notes->text = malloc(len);
78      free_notes->len = len;
79
80      fprintf(stderr, "malloc(%#zx) := %p\n", len, free_notes->text);
81      printf("%zu\n", index);
82
83      if (free_notes->next == NULL)
84          free_notes++;
85      else
86          free_notes = free_notes->next;
87  }
88
89  void perform_realloc() {
90      char *old;
91      size_t len, index;
92
93      index = getull("index: ");
94      if (index >= NUMBUCKETS || notes[index].text == NULL) {
95          puts("not a valid bucket");
96          return;
97      }
98
99      len = getull("size: ");
100     // TODO replace if needed
101     notes[index].text = realloc((old = notes[index].text), len);
102     notes[index].len = len;
103
104     fprintf(stderr, "realloc(%p, %#zx) := %p\n", old, len, notes[index].text);
105     puts("done.");
106 }
107
108 void perform_free() {
109     size_t index;
110
111     index = getull("index: ");
112     if (index >= NUMBUCKETS || notes[index].text == NULL) {
113         puts("not a valid bucket");
114         return;
115     }
116
117     // TODO replace if needed
118     free(notes[index].text);
119     fprintf(stderr, "free(%p)\n", notes[index].text);
120     puts("done.");
121
122     notes[index].text = NULL;
123     notes[index].next = free_notes;
124     free_notes = &notes[index];
125 }
126
127 void perform_calloc() {
128     size_t index, len, nmemb;
129     if (free_notes == back) {
130         puts("error, no free bucket left");
```

```
131         return;
132     }
133     index = free_notes - notes;
134 
135     nmemb = getull("nmemb: ");
136     len = getull("size: ");
137     // TODO replace if needed
138     free_notes->text = calloc(nmemb, len);
139     if (free_notes->text != NULL)
140     free_notes->len = len * nmemb;
141 
142     fprintf(stderr, "calloc(%#zx, %#zx) := %p\n", nmemb, len, free_notes->text);
143     printf("%zu\n", index);
144 
145     if (free_notes->next == NULL)
146         free_notes++;
147     else
148         free_notes = free_notes->next;
149 
150 }
151 
152 void perform_aligned_alloc() {
153     size_t index, alignment, len;
154     if (free_notes == back) {
155         puts("error, no free bucket left");
156         return;
157     }
158     index = free_notes - notes;
159 
160     len = getull("size: ");
161     alignment = getull("alignment: ");
162     len = getull("size: ");
163     // TODO replace if needed
164     free_notes->text = aligned_alloc(alignment, len);
165     free_notes->len = len;
166 
167     fprintf(stderr, "aligned_alloc(%#zx, %#zx) := %p\n", alignment, len, free_notes->
            text);
168     printf("%zu\n", index);
169 
170     if (free_notes->next == NULL)
171         free_notes++;
172     else
173         free_notes = free_notes->next;
174 }
175 
176 void perform_posix_memalign() {
177     int ret;
178     size_t index, len, alignment;
179     if (free_notes == back) {
180         puts("error, no free bucket left");
181         return;
182     }
183     index = free_notes - notes;
184 
185     alignment = getull("alignment: ");
186     len = getull("size: ");
187     // TODO replace if needed
188     ret = posix_memalign((void **) &(notes[index].text), alignment, len);
189     fprintf(stderr, "posix_memalign(%p, %#zx, %#zx) := %u\n", (void *) &notes[index].
            text, alignment, len, ret);
190     if (free_notes->next == NULL)
191         free_notes++;
192     else
193         free_notes = free_notes->next;
194     puts("done");
195 }
196 
```

```
197 void perform_read() {
198     char *buf;
199     size_t len, ret, tmp, index;
200 
201     index = getull("index: ");
202     if (index >= NUMBUCKETS || notes[index].text == NULL) {
203         puts("not a valid bucket");
204         return;
205     }
206 
207     buf = notes[index].text;
208 #ifdef OVOFFSET
209     buf += getull("offset: ");
210 #endif
211 
212     len = getull("length: ");
213 #ifdef OVLEN
214     if (notes[index].len + OVLEN < len) {
215         puts("you can not write that much");
216         return;
217     }
218 #endif
219 
220     ret = 0;
221     do {
222         tmp = read(0, buf + ret, len - ret);
223         if (tmp == 0) {
224             perror("read");
225             exit(1);
226         }
227         ret += tmp;
228     } while (ret < len);
229 
230 #ifdef ADDNULL
231     buf[len] = 0;
232     puts("null byte added");
233 #endif
234 
235     fprintf(stderr, "read(%p, %#zx) := \"", buf, len);
236     for (size_t i = 0; i < len; i++) {
237         fprintf(stderr, "%02hhx", buf[i]);
238     }
239     fprintf(stderr, "\"\n");
240     puts("done");
241 }
242 
243 void perform_write() {
244     char *buf;
245     size_t len, index;
246 
247     index = getull("index: ");
248     if (index >= NUMBUCKETS || notes[index].text == NULL) {
249         puts("not a valid bucket");
250         return;
251     }
252 
253     buf = notes[index].text;
254 #ifdef OVOFFSET
255     buf += getull("offset: ");
256 #endif
257 
258     len = getull("length: ");
259 #ifdef RDLEN
260     if (notes[index].len + RDLEN < len) {
261         puts("you can not read that much");
262         return;
263     }
264 #endif
```

```
265     write(0, buf, len);
266
267     fprintf(stderr, "write(%p, %#zx) := \"", buf, len);
268     for (size_t i = 0; i < len; i++) {
269         fprintf(stderr, "%02hhx", buf[i]);
270     }
271     fprintf(stderr, "\"\n");
272 }
273
274 void perform_exit() __attribute__((noreturn));
275 void perform_exit() {
276     exit(0);
277 }
278
279 int main () {
280     unsigned long long choice;
281
282     setbuf(stdout, NULL);
283     setbuf(stdin, NULL);
284
285     print_menu(1);
286     while (1) {
287         choice = getull("> ");
288         switch(choice) {
289         case 1:
290             perform_malloc();
291             break;
292         case 2:
293             perform_realloc();
294             break;
295         case 3:
296             perform_free();
297             break;
298         case 4:
299             perform_calloc();
300             break;
301         case 5:
302             perform_aligned_alloc();
303             break;
304         case 6:
305             perform_posix_memalign();
306             break;
307         case 7:
308             perform_read();
309             break;
310         case 8:
311             perform_write();
312             break;
313         case 9:
314             perform_exit();
315             break;
316         default:
317             print_menu();
318             break;
319         }
320     }
321 }
```