ShadowHeap: Memory Safety through Efficient Heap Metadata Validation

Johannes Bouché, Lukas Atkinson, and Martin Kappes*
Frankfurt University of Applied Sciences, Frankfurt, Germany
{johannes.bouche, lukas.atkinson, kappes}@fb2.fra-uas.de

Received: April 15, 2021; Accepted: September 2, 2021; Published: December 31, 2021

Abstract

In the past, stack smashing attacks and buffer overflows were some of the most insidious data-dependent bugs leading to malicious code execution or other unwanted behavior in the targeted application. Since reliable mitigations such as fuzzing or static code analysis are readily available, attackers have shifted towards heap-based exploitation techniques. Therefore, robust methods are required which ensure application security even in the presence of such intrusions, but existing mitigations are not yet adequate in terms of convenience, reliability, and performance overhead. We present a novel method to prevent heap corruption at runtime: by maintaining a copy of heap metadata in a shadow-heap and verifying the heap integrity upon each call to the underlying allocator we can detect most heap metadata manipulation techniques. The results demonstrate that ShadowHeap is a practical mitigation approach, that our prototypical implementation only requires reasonable overhead due to a user-configurable performance–security tradeoff, and that existing programs can be protected without recompilation.

Keywords: Memory Safety, Buffer Overflow, Memory Allocator, System Integrity

1 Introduction

With billions of computing devices processing untrusted data using software of all quality grades, securing this data processing adequately is on one hand an extremely difficult but on the other hand also an extremely important task. Core aspect of this is ensuring *memory safety* [1],[2],[3]: preventing access to memory outside of currently allocated regions. Memory safety bugs can be used by attackers for remote code execution or exfiltration of sensitive data [4]. There are various strategies towards stronger memory safety: software can be written in memory-safe languages that either disallow unrestricted pointers and use runtime checks [5] or statically prove memory safety guarantees [6]. Alternatively, memory safety violations can be found through static analysis tools or through dynamic analysis tools such as Valgrind [7] and especially through targeted fuzzing [8]. Other violations can be prevented sufficiently through operating-system level techniques such as Address Space Layout Randomization (ASLR) [4]. However, a large body of existing software is already written in memory-unsafe languages such as C and C++, making further mitigations necessary.

Stack based memory safety issues and buffer overflows have already received significant attention, and many mitigations such as stack cookies [9] or shadow stacks [10] are available. Heap-based attacks are well known since 2005 [11][12][13], but the few available mitigation techniques have seen little use, also because of high overhead.

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), 12(4):4-21, Dec. 2021
DOI:*10.22667/JOWUA.2021.12.31.004*

*Corresponding author: Faculty of Computer Science and Engineering, Frankfurt University of Applied Sciences, 60318 Frankfurt am Main, Germany, Tel: +49-69-1533-2791

One aspect of heap memory safety is ensuring the integrity of allocator data structures, which manage allocated and available memory chunks in the C standard library. If an attacker is able to manipulate these data structures, they can control future allocations and cause undesired behavior. While the glibc malloc implementation provides some defenses against this, they are usually deactivated for performance reasons and partially insufficient. The design of glibc malloc makes it impossible to determine whether a given chunk was originally allocated by malloc or by an attacker.

This paper proposes lightweight solutions and describes a prototypical implementation following three major design principles:

Usability: The augmented functions are injected into a process using `LD_PRELOAD`, which avoids the need to modify the allocator itself and can protect existing binaries without having to resort to techniques such as dynamic instrumentation. The proposed technique can therefore be applied in a granular manner to harden especially important processes.

Mitigations: The malloc library functions are augmented to replicate chunk metadata on a shadow heap. This allows chunk metadata to be validated whenever it is passed to a malloc library function offering means of protection against a wide range of manipulations.

Performance: The usage of the ShadowHeap functionality introduces only moderate overhead in terms of memory consumption and CPU-Utilization. Our performance evaluation shows that our implementation can outperform comparable implementations, e.g. *DieHard*[3], *DieHarder*[14], and *FreeGuard*[15].

Our ShadowHeap approach was initially described in [16]. In this paper, we expand on that work to provide a more detailed explanation of the used mitigation techniques and to demonstrate substantially improved performance.

In the following, we give an overview of related work, describe working principles of memory allocators, provide a threat model, explain mitigation techniques, present our prototypical implementation, and evaluate the performance of these mitigations.

2 Related work

In the past several approaches have been published which deal with the mitigation of heap-based attacks in order to prevent spatial and temporal memory errors [2]. Although these kind of attacks are published and well understood for years, memory safety issues are still prevalent in real world scenarios and productive environments [17]. Because of the wide range of different attack methods, a detailed explanation and classification of each method is out of scope for this document and we therefore refer to several attack-specific publications [11] [13] [18] for a better understanding. An overview of several techniques will be given in section 4 of this document.

For the purpose of our prototypic implementation we took into account different memory allocator designs [19] [20] [21] as well as several publications dealing with the mitigation of specific attacks or attack principles [22] [23] [24]. Instrumentation tools like Valgrind [25] and AddressSanitizer (ASAN) [26] are able to detect some but not all of the relevant attacks. Memory tagging can be used to detect buffer overflows and use after free accesses with low time overhead, but requires hardware assistance (HWASAN) e.g. from AArch64, compiler assistance, and can have substantial memory overhead due to alignment requirements [27]. Other techniques include N-Versioning and replication [28], leveraging the OS-kernel [29], automated error-correction approaches [30], external processes and Client-Server architectures [31], offering codeless patching [32] or model-driven detection approaches [33] [34].

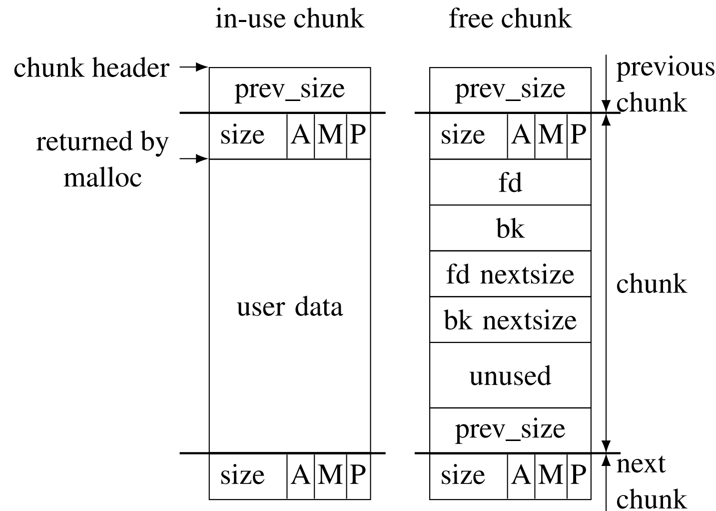


Figure 1: Chunk metadata layout

DieHard and FreeGuard are especially remarkable and are able to prevent a majority of heap-based attacks. To do this they have designed special security-focused allocators. DieHard [3] [14] provides propabilistic memory-safety by over-allocating heap space so that there are unallocated regions between in-use chunks. This minimizes the impact of illegal accesses such as buffer overflows and thus maximizes the "likelihood of correct execution", but doesn't attempt to reliably detect all memory safety violations. Furthermore, DieHard can replicate the running process under multiple randomized heap layouts in order to detect inconsistencies. Unfortunately, DieHard no longer works consistently on current Linux systems.

FreeGuard [15] combines BiBOP and free-list heap designs in order to maximize performance in highly concurrent environments. Its multiple sub-heaps can be used to serve allocation requests in parallel. As security features, FreeGuard separates metadata from chunks, uses randomized memory locations, and injects guard pages. However these features imply substantial memory overhead compared to newer glibc versions.

3 Memory allocators

Memory allocators are standardized components that provide dynamic memory management to a process. This saves developers from having to write error-prone low-level code, but also implies that many software components become vulnerable if the underlying allocator implementation has weaknesses. As background for understanding heap-based attacks we first discuss core aspects of the glibc memory allocator ($\geq$ 2.25), which was based on *ptmalloc2* [19] and is used as default on most Linux systems. Similar principles apply to the Windows allocators, but are out of scope here.

At a very high level, the allocator is accessed through the functions *malloc()* and *calloc()* which request a memory chunk of a particular size, or *realloc()* which changes the size of a chunk. To serve such requests, the allocator splits off chunks from an unallocated memory region (the *topchunk*). After use, the application calls *free()* to return a chunk to the allocator which stores free chunks in *bins* in order to optimize performance for later reuse.

Every allocated chunk contains a metadata header which stores the size of the chunk, and possibly other data (see Figure 1). This depends on the state of the chunk: the chunk is *in use* if it is held by the application, or *freed* if held by the allocator. Chunk state and other flags are encoded into unused bits of the size field. A freed chunk can store additional metadata in the memory otherwise occupied by

Table 1: Bins in ptmalloc (as of glibc 2.26)

category	count	linkage	depth	description
fast	8	single	∞	each bin contains a fixed size, very low book-keeping
small	54	double	∞	storage for medium chunk sizes
large	64	double	∞	each bin contains a range of sizes, extra links between sizes
unsorted	1	double	∞	cache of chunks before consolidation
tcache	64	single	7	per-thread cache with bins for fixed sizes

application data. The header is extended to include forward and potentially backward pointers so that the chunk can be enqueued in linked lists. The last word in the chunk repeats the size field. Viewed alternatively, the next bordering chunk's header starts with a *prev-size* field.

An *arena* data structure manages the bins, the topchunk, and additional metadata. There is usually one arena per thread. Every chunk in freed state is inserted into exactly one bin. When two neighboring chunks are freed, they may be *consolidated* into a larger chunk. The topchunk is always in freed state.

Upon each call into the allocator, it decides whether a request can be satisfied via a *first-fit* approach or whether chunks have to be split or consolidated. These decisions involve complex logic, so that attack methods have to set up a suitable allocator state that forces subsequent requests to use carefully chosen code paths that avoid certain security checks. To minimize memory fragmentation, memory usage, and CPU overhead, the allocator uses several bins for different sizes and purposes, including various caching layers as shown in Table 1. Very large chunks are managed with the *mmap()* system call and are never stored in bins. Tcache was introduced in glibc 2.26.

Since this allocator is a general purpose allocator its main focus is not to be the fastest or most secure allocator. Therefore appropriate mitigation mechanisms are often lacking or insufficient as described in the next section. A more in depth description of the design and its components is given by [19].

4 Threat Model

In this section we provide an overview of required conditions and techniques that allow an attacker to compromise systems via heap-based exploitation methods. We assume that ptmalloc2 itself is free of vulnerabilities, so that an attacker is required to exploit weaknesses in the user application, as discussed in this section. To successfully execute an attack, it is further necessary to set up a specific heap state (a.k.a. *heap feng shui*). While this requires access to the malloc API, this can often be achieved indirectly through legitimate inputs. A detailed description of all exploit methods as collected in the how2heap repository [18] would be out of scope, but we present a classification of their salient characteristics.

The *unsorted bin into stack* attack can serve as a representative example where a fake chunk is injected into the allocator. Given use-after-free access to a chunk on the unsorted bin, the attacker can use a buffer overflow to overwrite the forward pointer to point to a fake chunk. Now that the fake chunk has been inserted into the unsorted bin, a future allocation request will return this attacker-controlled memory, allowing application control flow to be hijacked.

In this example, a buffer overflow was used to inject a fake buffer into the heap. Other possible outcomes include overlapping, duplicate, or completely attacker-controlled chunks, which can ultimately lead to arbitrary memory reads and writes or even arbitrary code execution. Other vulnerabilities include:

Double-Free (DF): A legitimately allocated chunk is freed twice, which might corrupt the heap state in a way such that the chunk occurs multiple times in the allocator's bins. Subsequent calls to the allocator will then eventually return the same buffer to more than one caller, which can lead to a race

condition. Because the same chunk will be used by multiple parts of the software, this can also be used as a method to cause reads or writes to certain data, e.g. to read data that another component has written to this memory location.

Invalid-Free (IF): A pointer to a forged memory chunk is freed. Such vulnerabilities can be used to inject attacker controlled chunks into the allocator. Since the allocator relies on metadata within the chunk, it has no means to distinguish between chunks obtained by legitimate API invocation and chunks that are placed arbitrarily within user memory, for example on the stack.

Buffer Overflow (BO): An out-of-bounds write corrupting the data in the chunk physically located right after the chunk which was overflown. This could corrupt user data in that chunk, but also chunk metadata. Overflows that target the size field an its encoded bits are especially important for several attack techniques.

Off-by-One Overflow (OB1): A special case of a buffer overflow that writes just one item beyond the valid memory range. E.g. this could manipulate the size field of the following chunk with the aim to enforce consolidation of chunks based on invalid metadata. Under certain conditions, this can cause the allocator's consolidation phase to create overlapping chunks that would later be returned to the user.

Use-After-Free (UAF): An already freed chunk is used to either corrupt heap metadata (UAF-Write) or to read otherwise unavailable data (UAF-Read). Such UAF-Reads might be used to extract sensitive data from locations which enable the attacker to circumvent other mitigation techniques such as ASLR. By utilizing UAF-Writes an attacker might inject own fake memory chunks in order to trick the allocator into returning controlled memory locations to the requesting application code.

In Table 2 we classify attacks depending on the involved vulnerabilities. These vulnerabilities have common patterns and properties as follows:

Category A (1–9): Double-free or invalid-free vulnerabilities which allow the injection of fake or duplicate chunks into bins of the allocator. By utilizing such methods the memory offsets of particular allocations might be controlled deterministically.

Category B (10–15): Methods utilizing buffer overflows against freed chunks, allocated chunks or while they are being stored on the unsorted bin. In such scenarios the allocator uses manipulated metadata during regularly occurring consolidation phases to inject fake buffers, overlap other buffers or achieve attacker controlled memory writes.

Category C (16): Methods such as buffer overflows or UAF-Writes which are based on the insertion of fake chunks particularly into the tcache bin introduced in glibc version 2.26. The technique is used to inject arbitrary buffers into the allocator.

Category D (17): Unique method in which the topchunk size stored in the main arena can be manipulated by using a buffer overflow. At the time of writing the method *house of force* was the only published example [11]. The attack manipulates the topchunk metadata and uses an integer overflow in order to trick the allocator into returning an arbitrary location.

Category E (18–21): All methods utilizing UAF-Write vulnerabilities in order to alter metadata of freed chunks that are stored in bins other than the unsorted bin or the tcache. These techniques are used in various manners to inject fake or attacker controlled buffers or to achieve arbitrary memory writes.

Category F (22–23): Methods using buffer overflows or off-by-one overflows which specifically target the *prev in use* flag (*P*-flag) or small parts of the size field which can force the allocator to overlap buffers or to achieve arbitrary memory writes.

Table 2: Attack methods in the how2heap repository and their used vulnerabilities

ID		name	weakness	glibc			outcome	mitigated
				2.25	2.26	2.28		
A	1	fastbin dup	DF	✓	✓	✓	Duplication	✓
	2	fastbin dup consolidate	DF	✓			Duplication	✓
	3	fastbin dup into stack	DF	✓			Arbitrary buffer	✓
	4	house of spirit	IF	✓			Fake buffer	✓
	5	tcache dup	DF		✓		Duplication	✓
	6	tcache house of spirit	IF		✓	✓	Fake buffer	✓
	7	house of botcake	DF		✓	✓	Arbitrary buffer	✓
	8	house of mind fastbin	BO	✓	✓	✓	Memory write	✓
	9	mmap overlapping chunks	BO	✓	✓	✓	Overlap	✓
B	10	overlapping chunks	BO	✓	✓	✓	Overlap	✓
	11	overlapping chunks 2	BO	✓			Overlap	✓
	12	poison null byte	BO	✓			Overlap	✓
	13	unsorted bin attack	BO,UAF	✓	✓	✓	Memory write	✓
	14	unsorted bin into stack	BO,UAF	✓	✓	✓	Fake buffer	✓
	15	house of roman	BO,UAF	✓		✓	Fake buffer	✓
C	16	tcache poisoning	BO,UAF		✓	✓	Arbitrary buffer	✓
D	17	house of force	BO	✓	✓	✓	Arbitrary buffer	✓
E	18	house of lore	BO,UAF	✓	✓	✓	Arbitrary buffer	
	19	large bin attack	BO,UAF	✓	✓	✓	Memory write	
	20	fastbin reverse into tcache	BO,UAF		✓	✓	Fake buffer	
	21	tcache stashing unlink attack	BO,UAF		✓	✓	Fake buffer	
F	22	unsafe unlink	BO,UAF	✓	✓	✓	Arbitrary write	
	23	house of einherjar	OB1	✓	✓	✓	Overlap	

5 Mitigation Techniques

In this section, we discuss mitigation techniques used for our ShadowHeap approach. Together, these methods achieve reliable mitigation against all attacks in categories A–D. Since our approach protects the integrity of underlying data structures rather than detecting individual attacks, these mitigations are also able to protect against future attacks.

Verification. We verified the mitigations using exploits from the how2heap repository [18]. How2heap has adapted attacks for different glibc versions between 2.23 to 2.31. The main concepts of an attack can usually be ported to later versions as well, but might require certain build configurations or slightly different allocation patterns. Glibc 2.26 introduced the tcache as a major performance improvement, but without sufficient security checks [33]. The resulting highly predictable behavior enabled novel heap-based exploitation methods and allowed existing attack methods to be adapted to the tcache.

To verify successful mitigation, we executed the various how2heap attack examples under the applicable glibc versions from 2.25 to 2.28. It can be assumed that there are several undisclosed attack methods, or adaptations of known attacks against other glibc versions. Over the last year of our work,

five previously unknown techniques were added to how2heap.[1][2] Of these, ShadowHeap was able to mitigate three attacks reliably and without any modification.

Overview of mitigation techniques. Generally spoken, the suggested mitigations rely on storing a snapshot of chunk metadata in a ShadowHeap. In the *free protection*, we store metadata of in-use chunks and check consistency of it's metadata when the chunk is freed later on. For the *bin protection* including the *tcache protection*, we store metadata of freed chunks that are stored in particular bins which where chosen for performance reasons and mitigation purposes. Similarly, the *topchunk protection* refers to metadata of a specific unallocated chunk into which unused chunks might be consolidated eventually. The provided mechanisms were designed with the aim to maximize mitigation capabilities with acceptable performance overhead and a reasonable level of additional memory consumption.

This section therefore contains a more thorough description of the already mentioned protection schemes, their requirements and performance implications. Finally, we discuss limitations of each concept and evaluate our distinct design goals.

Metadata validation. All described approaches involve the storage and validation of metadata copied from the related memory chunk to a separate shadow location. This requires knowing which chunk metadata fields are currently valid which is non-trivial. As discussed in section 3, the glibc allocator stores metadata in headers for each chunk. However, glibc uses low-level programming techniques to save space. Some information is encoded into unused bits of the size field, some fields relate to adjacent chunks, and some fields overlap with user data and are only active on freed chunks and on certain bins. Thus, storing and checking this information requires an accurate understanding of the state of the chunk and its current position in the arena data structures. This leads to the following consequences for a validation approach.

Glibc stores chunks in various bins for different sizes and purposes. These are implemented as singly- and doubly-linked lists. While such a design allows for efficient insertion and removal of chunks, validation requires traversal of the entire linked list.

The practice of storing metadata in adjacent chunk headers means that it is not sufficient to store and check just the main header of a chunk, but that metadata of the neighboring chunks also have to be updated and verified. In reverse, this also means that the metadata header of a given chunk can be legitimately modified by interposed API invocations relating to adjacent chunks. Without accounting for such changes, validation will fail. In particular, the consolidation procedure can change metadata in difficult to track ways.

This also has consequences for validating the A/M/P flags in the size field, which cannot be validated with equal performance investment. Whereas the *mmap* and *main arena* flag are fixed for the lifetime of a chunk, the *P*-flag depends on the state of the preceding chunk. Its correct state cannot be obtained from the metadata itself, but can only be predicted by tracking the dynamic state of the entire heap, as discussed in the following paragraphs.

Metadata tracking and the *P*-flag. Due to the overhead in storing and verifying metadata, it is necessary to make some compromises which will limit which vulnerabilities can be mitigated. As already explained, the glibc data structures are not particularly well suited to efficiently determine the current heap state. In particular, verification of the *P*-flag poses special challenges that will be discussed in the following.

For the purposes of our wrapper-based approach, we can distinguish between *static* and *dynamic* metadata. Static metadata is guaranteed to stay the same for a given time. For example, the *chunk size*, *mmap* flag, and *main arena* flag will be static between allocation and deallocation of a chunk. In contrast,

[1] Mitigated new attacks: mmap overlapping chunk, house of roman, house of mind fastbin.
[2] Unmitigated new attacks: fastbin reverse into tcache, tcache stashing unlink attack.

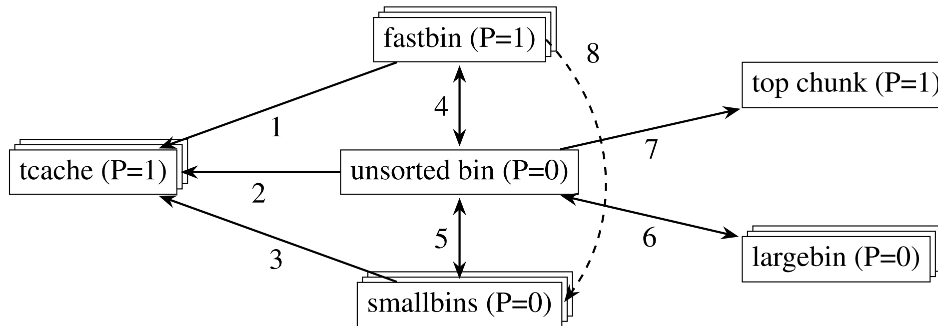


Figure 2: Location changes of unallocated chunks within the allocator

dynamic metadata such as the *P*-flag and *prev size* field can change at any time, depending on the state of the previous chunk.

Thus, certain metadata can not be obtained directly but only indirectly. This leads to a chicken-and-egg problem: to know the correct state of the *P*-flag, we need to know the state of the previous chunk, which requires the previous chunk's address. The address can be determined from the *prev size* field, except that this field can only be accessed if the *P*-flag is set – which we want to verify.

The alternative would be to store additional location information outside of the chunk, but such data about neighboring chunks must be updated and verified whenever it changes. This requires an accurate understanding of all potential state transitions for this chunk. A protection of this field must consider the following facts:

- The *prev size* field at the front of the chunk header is only valid if the *P*-flag is unset.
- The *P*-flag indicates the previous chunk's allocation state, but remains set on some bins.
- Chunks can change their state at any time, flipping the *P*-flag in the next chunk's header.
- Freed chunks might move between bins, flipping the *P*-flag in the next chunk's header.
- Freed chunks can merge or split, which will change its size, including the *prevsize* field.

While some of this information can be obtained with moderate overhead, other aspects can only be determined by storing copies of whole bin contents. By maintaining a shadow copy of all bins, all relevant state changes can be predicted and applied to the shadow copy.

These state transitions of freed chunks are illustrated in Figure 2. Numbers 1–8 label all relevant state transitions. For each bin, the corresponding value of the *P*-flag for chunks on that bin is shown. The transitions 1–3 in figure 2 refer to transitions in which the allocator *stashes* unused chunks from certain bins to the tcache for performance reasons (introduced in glibc 2.26). The transitions 4–6 are regular cleanup mechanisms which are applied to every chunk in the *unsorted bin* during a *first-fit* search. Transition 7 occurs when a chunk is merged back into the topchunk. Transition 8 represents an exceptional transition referred to as *malloc_consolidate()* which cannot be observed from allocator data structures alone. Internally, consolidation is performed by putting all fastbin elements to the unsorted bin and then merging and moving them into their respective bin. Since a wrapper-based approach can only observe the heap state before and after the allocation, it is not possible to recognize the intermediate step to store chunks on the *unsorted bin*, which is what changes the *P*-flag. While it is possible to track how chunks move between bins by comparing before-and-after snapshots, this is inefficient since the fast, small, and large bins can contain a very large number of chunks.

Given the glibc allocator design and the limitations of a wrapper-based approach, our mitigation and performance goals conflict. We have to conclude that full protection of the *prev size* and *prev in use*

fields would require a tracking effort that is unreasonable for performance reasons. By relinquishing full *prev in use* state tracking, we lose protection against category F attack methods, but gain considerable performance.

Instead, we propose metadata verification for two specific bin types. The tcache bins have fixed size and are thus cheap to verify. The unsorted bin typically contains only a small number of elements since chunks are quickly moved to other bins, though its size ultimately depends on the application's allocation patterns. Thus, these bin contents can be protected with a reasonable amount of overhead. In general, bin protection could be extended to all other bin types (fast, small, large) in order to be able to recognize all heap-based UAF-Writes and all state changes performed by the allocator which would enable to mitigate all current and potentially future exploitation methods from category A–F. In the following paragraphs our three chosen techniques are described with respect to their performance aspects as well as their limitations.

Free Protection. For each allocating function call such as *malloc()*, the returned chunk pointer and its metadata is copied into a lightweight data structure. Upon each releasing function call such as *free()*, the provided chunk metadata is verified against the shadowed version in the utilized data structure and removed from the data collection. If pointer validation fails, the process is terminated in order to reliably protect against all vulnerabilities from category A.

The limitations of this approach are that the removal of data is essential upon every releasing call to *free()* in order to prevent usage of large and unjustified amounts of unused data. Therefore metadata changes of deallocated chunks are not recognized. Furthermore unmitigated attacks are still possible involving the manipulation of dynamic metadata parts such as the *P*-flag or by manipulating metadata of already *freed* chunks.

Bin Protection. Vulnerabilities from categories B and C target the unsorted bin and the tcache, which serve as caches before the main malloc data structures in order to satisfy requests with the *first-fit* behavior. These vulnerabilities can be easily mitigated because the bins have limited size: tcache has fixed bounds (64 bins of at most 7 elements) and the unsorted bin is usually kept short by frequent consolidation events.

For the unsorted bin, 32 bytes have to be stored per entry, including forward and backward links. While the unsorted bin has unbounded length it is typically short, and we store up to 128 entries (up to 4 KByte total). For the tcache, we store 24 bytes per entry (up to 10.5 KByte total), including the forward link. For each allocating or releasing call the elements from these bins have to be persisted into the ShadowHeap. Upon any subsequent call into the allocator, our mitigation compares all elements in the bins against the shadowed version and terminates the process if any manipulation is detected. Time overhead is $O(n)$ for n the number of elements in the bins.

The limitations of that concept are that a certain amount of memory has to be reserved up front, but often for only a small amount of shadowed metadata. Further the verification of such data is depending on the actual size of each bin and is therefore drastically usecase-specific. Additionally only methods are mitigated which do not solely focus on the manipulation of the *P*-flag as this flag is omitted in verification. Generally spoken, a moderate amount of performance and memory utilization has to be invested in order to mitigate a quite large percentage of published attack methods. For current and future methods relying on the manipulation of metadata of chunks being stored on such bins, the proposed solution provides reliable mitigation as the change of such data can by definition not occur in legitimate usecases.

Topchunk Protection. To protect against the *house of force* vulnerability (category D), the topchunk is saved in a shadow copy, and validated upon subsequent calls into the allocator. This mitigation is very cheap, since only 32 bytes have to be saved and checked ($O(1)$ time and space). Similar mitigations could be extended to other metadata in the malloc arena and therefore also acts as a representative example for securable metadata located in the main arena. Currently the mitigation concept provides reliable

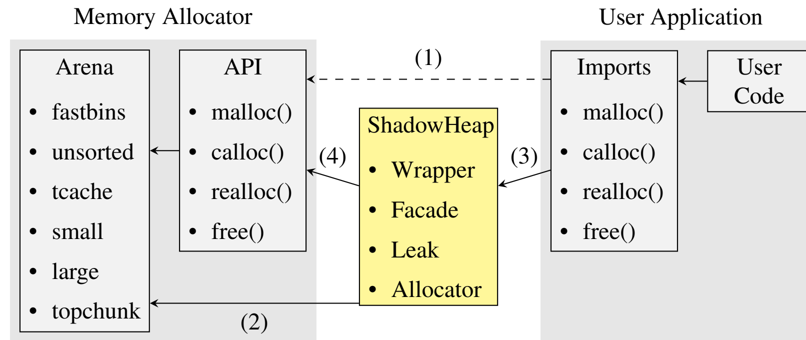


Figure 3: ShadowHeap working principle

mitigation only against one published attack.

The limitations of this concept are that in order to obtain the location of sensitive data an additional method for their leakage is required. Such data is usually kept local to the allocator and has to be extracted into separate memory locations. To extend this mitigation approach to a multi-threaded setting, it would be necessary to leak and check the topchunk metadata for all active arenas, not just the main arena.

6 Prototypical Implementation

In order to verify the mitigation approach, we created a prototypical implementation of the ShadowHeap technique. In this section, we provide an overview of the architecture and explain specific implementation choices.

Architecture. The ShadowHeap mitigation layer is injected between the user application and the libc functions that implement the heap allocator. Our implementation consists of wrapper, facade and leak components as well as an internal allocator. Figure 3 provides an overview of our architecture and data flows. (1) The imported allocator functions are hooked so that all calls are redirected through the ShadowHeap. (2) We leak the allocator's *main arena*. (3) Allocation requests from the user application are intercepted by the ShadowHeap, which allows for security checks to be applied. (4) ShadowHeap ultimately satisfies the request by using the original allocator.

Wrapper component. The Wrapper component provides an entrypoint into the allocator API. In general, this just consists of invoking metadata validation checks provided by the Facade component, immediately before and after invoking the original allocator APIs. An exception is the *realloc()* function, which will usually attempt to grow the current chunk if the adjacent memory is free. Since this makes the heap state difficult to track, the Wrapper will instead allocate a new chunk, copy the data over, and free the old chunk.

The Wrapper also manages the ShadowHeap lifecycle, in particular ensuring correct initialization. The order in which the *Executable and Linkable Format* (ELF) loader invokes static initialization functions is not deterministic, so the Wrapper's functions might have been linked to other libraries before initialization has been completed. In case there are allocation requests before initialization, they are satisfied via the internal allocator, without further checks. Full checks will be available by the time the application's *main()* is called.

Leak component. During initialization, the Leak component provides access to the internal allocator data structures, in particular the main arena. The original *malloc()* functions can be acquired from the ELF loader via the *dlsym()* function. We then perform a user-after-free attack to reveal the arena location and to acquire the tcache. The component accounts for layout differences between libc versions.

However, a patched libc is required to leak the tcache prior to glibc 2.28.

Facade component. The Facade is the main ShadowHeap component. It provides storage for shadow-copies of heap metadata, and provides checks for various points in the allocation lifecycle as outlined in section 5. These checks save and validate heap metadata. Individual checks can be toggled via feature flags.

For example, a check is invoked just before the original *malloc()* is called in order to verify the integrity of relevant heap metadata. Another checks is invoked just after *malloc()* returns in order to update any changed metadata, and to register the returned pointer for the free-protection. Care has been taken to minimize how much metadata is touched in order to minimize performance overhead. In particular, tcache bins are separated by size, so that it is sufficient to only check the bins corresponding to the size of the allocation request.

Copies of metadata are stored in fixed-sized arrays for the unsorted-bin and tcache protection. Whereas these are usually processed in bulk, the free protection requires random access to the metadata indexed by the chunk pointer. Feasibility of implementing this protection depends crucially on an efficient data structure to store metadata. Practical experiments indicated that C++ standard library containers were too inefficient for this use case, so that a custom storage layer was designed. While the optimal data structure depends on the specific allocation patterns, hash tables are a good general choice due to attractive worst-case behavior. Our custom hash cache uses a SplitMix64-derived hash function[35], features very compact buckets that fit into one CPU cache line, and can be driven to a load factor of almost 1. In case multiple collisions occur in the cache, old entries are evicted to a full hash table implementation. While hash-based data structured generally exhibit poor cache locality, this technique minimizes memory accesses in the typical case. This approach makes it possible to perform insertions and removals/retrievals including validation in (amortized) $O(1)$ time, with only 16 bytes used per entry.

Internal allocator component. During the checks in the Facade component, it is important to not disturb the heap state in any way. Nevertheless, heap memory is required for ShadowHeap data structures and for buffered I/O. Therefore, a simple internal allocator component was implemented. This allocator uses *mmap()* to acquire memory directly from the operating system. Since all internal allocations are trustworthy by definition, this allocator largely avoids safety checks of its own.

Consequences and limitations. The presented design is able to satisfy the usability, mitigation, and performance goals defined in section 1. Due to the wrapper approach, it is possible to inject the ShadowHeap protections without having to recompile libc or the user application itself. Furthermore, it is not necessary to provide a different heap implementation. The different mitigation techniques can be applied separately, thus allowing for an arbitrary security–performance tradeoff. Since there are no pointers from chunks to the ShadowHeap, this method's low susceptibility to heap based attacks contributes to overall system security.

However, the approach suffers from certain limitations: Category E attacks are prohibitively expensive to mitigate because the targeted bins have unbounded size. Category F attacks manipulate the *prev in use* field, but it is difficult for a wrapper approach to determine the correct status of this field since it depends on the in-use status of the *previous* chunk. Furthermore, the proof of concept implementation assumes single-threaded use. Only the main arena and the main thread's tcache are leaked, and metadata stores are not protected against concurrent modification.

7 Performance Evaluation

To evaluate the overhead of applying the ShadowHeap mitigation, a number of real-world examples and benchmark workloads were executed in mitigated and non-mitigated configurations.

Table 3: Median performance measurements for the malloc experiment, relative to baseline.

mitigation	time	mem	mitigation	time	mem	mitigation	time	mem
L1	1.42	1.22	ptr	1.58	1.22	none	1.00	1.00
L2	1.45	1.22	top	1.17	1.02	hooked	1.04	1.02
L3	3.62	1.22	usb	3.25	1.02	hash-ptr	2.04	1.38
L4	4.08	1.22	tca	1.62	1.02	tree-ptr	3.31	1.62
freeguard	1.96	5.10	diehard	2.35	1.60	dieharder	5.18	3.25

Testing methodology. The tests were executed in a containerized environment using the Linux 5.8.0 kernel, Debian Buster operating system in the container, and glibc 2.28. The system used a Ryzen 3900X CPU with 32GB DDR-3600 RAM. Due to the topology of this CPU, each experiment was pinned to a randomly chosen core complex, thus providing 3 cores, 6 threads, and 16MB L3-cache. While all tests involve single-threaded programs, the GCC-test will run concurrent processes.

Overhead was measured with the *time* utility which reports elapsed time, CPU time, and kernel time used by the child processes, as well as peak memory consumption. First, a baseline without any mitigations is measured. Then, ShadowHeap mitigations are successively activated in levels L1–L4 (free protection, topchunk protections, unsorted bin protection, and tcache protection) using optimized executables that only include the checks for those levels. Additional factors were tested in the malloc test. In order to compare alternative mitigation approaches, all experiments were measured under FreeGuard (FG). Diehard and Dieharder were also used, but did not perform reliably.

Overhead is normalized to the baseline and thus reported as a dimension-less factor. For quantitative comparisons, Welch's t-test is used.

Malloc test. In this experiment, we performed various allocation/deallocation patterns in a single thread, such as allocating thousands of chunks and then freeing them in bulk, or interspersing allocations and deallocations in a random order. This is expected to stress our free-protection and unsorted bin protection, and thus present a worst-case scenario. As additional factors, different data structures for the free protection and individual protection checks were benchmarked. For those scenarios, unused checks were disabled at runtime. Furthermore, Diehard and Dieharder were measured in this test. Each configuration was sampled 50 times.

Our findings are summarized in Table 3. We observe that the runtime overhead increases from L1 (42%) to L4 (308%) as more mitigations are enabled. Since memory use by the free protection dominates, there are significant but only negligible differences in memory overhead between levels (L1 22% vs L4 22%, $p = .006$). Whereas the ShadowHeap-L1 (free protection) is significantly faster than FreeGuard at 96% overhead ($p < .005$), FreeGuard outperforms the full mitigation suite ShadowHeap-L4 ($p < .005$) but also consumes significantly more memory (410% vs 22%, $p < .005$). ShadowHeap-L4 is both significantly faster than Dieharder (308% vs 418%, $p < .005$) and consumes significantly less memory (22% vs 225%, $p < .005$).

The table also contains measurements for individual ShadowHeap features. Entries *ptr*, *top*, *usb*, and *tca* refer to the free protection, topchunk protection, unsorted bin protection, and tcache protection features respectively. Whereas L1 and *ptr* provide identical mitigations, selecting mitigations at runtime comes at a significant time overhead (42% vs 58%, $p < .005$). In this particular experiment, the majority of ShadowHeap overhead can clearly be attributed to the unsorted bin protection, whereas there is a small but significant difference between *ptr* and *tca* (58% vs 62%, $p < .005$). Figure 4 compares different data structures for the free protection. The *ptr* configuration uses our optimized hash-cache data structure and is both significantly faster ($p < .005$) and more memory-efficient ($p < .005$) than C++ standard library

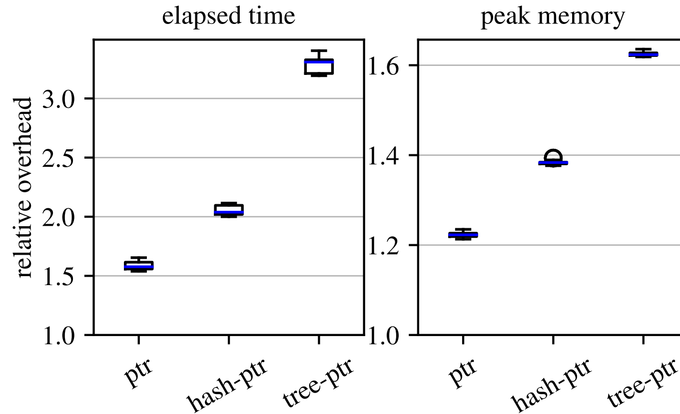


Figure 4: Comparison of overhead for different data structures for the free protection (malloc experiment, 50 repetitions).

Table 4: Median performance measurements for the GCC experiment, relative to baseline.

mitigation	elapsed	usr	sys	usr+sys	mem
L1	1.11	1.06	1.21	1.08	1.04
L2	1.11	1.07	1.20	1.08	1.04
L3	1.19	1.15	1.24	1.15	1.04
L4	1.22	1.19	1.24	1.20	1.04
FG	3.21	1.10	17.37	2.55	1.45

data structures (*tree-ptr* uses *std::map*, *hash-ptr* uses *std::unordered_map* as provided by GCC 10.2). The *none* configuration represents the baseline with the native allocator, and *hooked* a ShadowHeap configuration with all mitigations disabled at compile time. The resulting indirection already comes at a small but significant overhead (0% vs 4%, $p < .005$).

GCC compilation test. As a demonstration of a practical workload, we compiled GCC 9.2 under different mitigation levels, using GCC 8.3 as the host compiler. The compilation process involves both frequent invocations of shell utilities that stress heap initialization performance, as well as complex allocation patterns in the compiler itself.

Results are shown in Table 4. The levels L1–L4 have between 11% to 22% time overhead, compared to 221% overhead for FreeGuard. However, this measures elapsed wall time which is not fully informative due to concurrency. Measuring CPU time (in user and kernel mode) indicates 8% to 20% overhead for ShadowHeap, compared to 155% overhead for FreeGuard which primarily stems from its substantially increased time spent in syscalls. With regards to peak memory usage, ShadowHeap has about 4% overhead compared to 45% for FreeGuard. All these differences between ShadowHeap-L4 and FreeGuard are significant ($n_1 = n_2 = 10, p < .005$).

Perlbench. The Perlbench suite[3] collects various microbenchmarks for the Perl interpreter. It is no longer useful as a benchmark suite, but is merely used here to generate load. While Perl performs its own memory management with refcounting, it uses the system malloc to manage its arenas and to store string contents. We used Perl 2.28 and configured a fixed number of iterations (cpu-factor 2029794).

[3] `https://github.com/gisle/perlbench` at commit d4033ad (2012)

Table 5: Median run time and median peak memory usage at various ShadowHeap mitigation levels, relative to baseline.

	run time					peak mem	
	L1	L2	L3	L4	FG	L4	FG
malloc	1.42	1.45	3.62	4.08	1.96	1.22	5.10
gcc	1.11	1.11	1.19	1.22	3.21	1.04	1.45
perlbench	1.11	1.11	1.10	1.15	3.97	1.13	2.60
octane	1.00	1.00	1.01	1.01	1.00	1.00	1.05
tar	1.00	0.98	0.98	1.01	1.01	1.02	1.05

The results are included in Table 5 and are in line with the findings for the GCC test. The ShadowHeap levels have 11% to 15% time overhead and 13% peak memory overhead, compared to 297% time and 160% peak memory overhead for FreeGuard. These differences between ShadowHeap-L4 and FreeGuard are significant ($n_1 = n_2 = 10, p < .005$).

Octane and TAR extraction. The previous experiments involved complex allocation patterns. However, many programs do not perform substantial heap allocations. As an example, we ran the Octane benchmark[4] for JavaScript engines under SpiderMonkey 60, which can be used in a single-threaded mode. While Octane includes both CPU- and memory-heavy benchmarks, SpiderMonkey uses garbage collection and is thus not dependent on the libc heap. As another example of a CPU-limited workload, we extracted a large archive with GNU tar 1.30.

Results are included in Table 5. The runtime of any ShadowHeap configuration does not significantly differ from FreeGuard in these experiments (Octane: $n_1 = 40, n_2 = 10, p = .29$, Tar: $n_1 = 80, n_2 = 20, p > .05$). However, ShadowHeap-L4 uses slightly less peak memory than FreeGuard (Octane: $n_1 = n_2 = 10, p < .005$, Tar: $n_1 = n_2 = 20, p < .005$).

Combined Results. Results for different experiments are compared in Table 5 and Figure 5. As discussed above, ShadowHeap overhead increases as mitigations are enabled, but this overhead depends on the specific use case. Overhead is highest in the synthetic *malloc* experiment, where FreeGuard achieves better throughput. The situation is reversed in the GCC and Perlbench experiments, where ShadowHeap overhead is much lower than FreeGuard.

8 Discussion

In the introduction we introduced the three major design goals of usability, mitigations and performance. ShadowHeap largely achieves these goals, but the details require further discussion.

Our results show that ShadowHeap can offer an attractive performance profile, but this depends on the exact allocation patterns. The *malloc* benchmark is limited by the speed of individual allocations, whereas the real-world experiments involve many short-lived processes so that initialization performance has a greater weight. ShadowHeap has to duplicate some of the work of the glibc allocator and negates performance optimizations such as linked lists, so that allocators with integrated protections like FreeGuard are faster on some scenarios. In particular, ShadowHeap cannot defend against attacks from categories E + F within a reasonable performance budget. However, ShadowHeap uses consistently less memory (see Fig. 5).

[4] `https://github.com/chromium/octane` at commit 30b1d8d (2017)

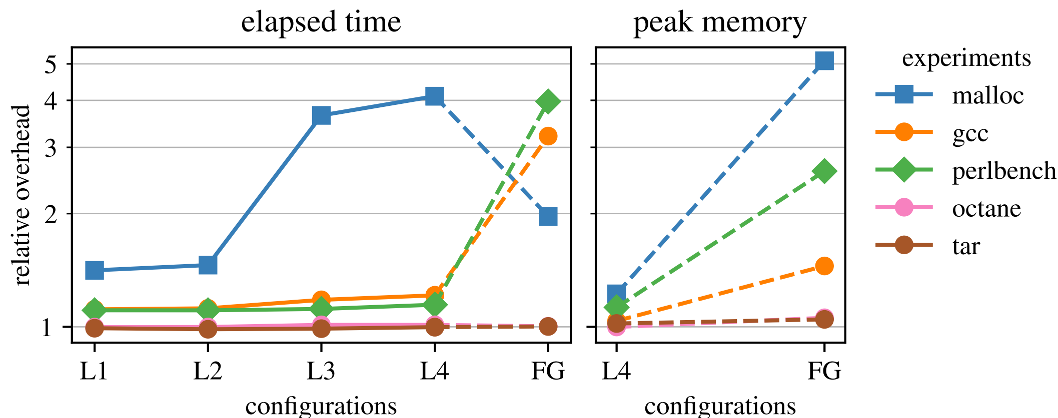


Figure 5: Comparison of overhead across experiments and mitigation levels (logarithmic scale).

As previously discussed, ShadowHeap is able to mitigate against most but not all of the analyzed attacks. Defending against attacks from categories E + F is especially challenging due to performance constraints (see sections 5 and 6). It is worth considering that the ShadowHeap implementation might introduce vulnerabilities of its own, and that its presence is easily detectable by attackers. Specific defenses are out of scope for our prototypical implementation. We therefore assume that the ShadowHeap initialization including the `LD_PRELOAD` mechanism completes correctly. Glibc heap metadata is only protected from the end of initialization until the termination of the process.

Utilizing the `LD_PRELOAD` mechanism enables an easily usable mitigation of the stated attacks without recompilation or dynamic instrumentation. Despite the mentioned limitations, ShadowHeap allows for attractive security vs performance tradeoffs.

9 Conclusion

In this paper, we presented a technique that is able to mitigate a wide range of heap-based attacks and allows the user to flexibly select mitigation components depending on individual security and performance requirements. ShadowHeap is available as Open Source[5] and is able to protect nearly all single-threaded POSIX-compliant programs. Depending on the application, the performance impact of ShadowHeap can range from unnoticeable to moderate (0%–22%, excluding the malloc test), but in practice the memory overhead seems to be reasonably low (0%–22%). Thus, ShadowHeap could be especially useful in resource-constrained systems with high integrity requirements, such as IoT devices.

Reflecting on these mitigations, we find that the core issue with ptmalloc is its inherent predictability, combined with the unchecked assumption that the heap will be secure if the APIs are only used correctly. Given available computing power, an allocator that is purely tuned for performance might no longer be appropriate.

As currently written, ShadowHeap has a number of limitations. The overhead from the wrapper approach could be avoided if the security features were directly integrated into the allocator. The prevsize flag can not be reasonably protected without such integration, which might be exploitable by novel attacks. Wrapping overhead is highest with the unsorted bin protection which has to traverse all linked list elements. Protection for the large and small bin has not been implemented as their size is unbounded, unlike the tcache and the unsorted bin.

[5] `https://github.com/fg-netzwerksicherheit/ShadowHeap`

Future work includes implementing mitigations against category E–F vulnerabilities, and integrating mitigations directly into a malloc implementation instead of depending on a wrapper approach. This would also make it possible to reasonably perform these mitigations in multithreaded programs. The metadata records could also be protected further with process isolation techniques as already shown by [24]. Aside from protecting glibc's ptmalloc implementation, similar mitigations could be implemented with other allocators such as jemalloc, Windows Heap. Finally, further performance analysis e.g. on web servers and representative benchmark suites is necessary in order to provide a wider view on real-world performance.

Acknowledgments

This work was supported by the German Federal Ministry for Economic Affairs and Energy grant no ZF4131805MS9.

References

[1] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proc. of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09), Dublin, Ireland*, pages 245–258. ACM, June 2009.

[2] M. S. Simpson and R. K. Barua. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Journal of Software: Practice and Experience*, 43(1):93–128, 2013.

[3] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, 41(6):158–168, June 2006.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Proc. of the 2013 IEEE Symposium on Security and Privacy (S&P'13), Berkeley, California, USA*, pages 48–62. IEEE, May 2013.

[5] P. H. Hartel and L. Moreau. Formalizing the safety of Java, the Java Virtual Machine, and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.

[6] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proc. of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03), San Diego, California, USA*, pages 69–80. ACM, June 2003.

[7] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of the 28th ACM SIGPLAN Conference on Programming Design and Implementation (PLDI'07), San Diego, California, USA*, pages 89–100. ACM, June 2007.

[8] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Proc. of the 2018 Network and Distributed Systems Security Symposium (NDSS'18), San Diego, California, USA*. NDSS, February 2018.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symposium, San Antonio, Texas, USA*, pages 63–78. USENIX Association, January 1998.

[10] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1):1–40, November 2009.

[11] P. Phantasmagoria. The Malloc Maleficarum. `https://seclists.org/bugtraq/2005/Oct/118` [Online; accessed on October 4, 2021], 2005.

[12] J. N. Ferguson. Understanding the heap by breaking it. `https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf` [Online; accessed on October 4, 2021], 2007.

[13] M. F. Rørvik. Investigation of x64 glibc heap exploitation techniques on Linux. Master's thesis, University of Oslo, 2019.

[14] G. Novark and E. D. Berger. DieHarder: securing the heap. In *Proc. of the 17th ACM conference on Computer and Communications Security (CCS'10), Chicago, Illinois, USA*, pages 573–584. ACM, October 2010.

[15] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu. Freeguard: A faster secure heap allocator. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17), Dallas, Texas, USA*, pages 2389–2403. ACM, October 2017.

[16] J. Bouche, L. Atkinson, and M. Kappes. Shadow-Heap: Preventing heap-based memory corruptions by metadata validation. In *Proc. of the 2020 European Interdisciplinary Cybersecurity Conference (EICC'20), Rennes, France*, pages 1–6. ACM, November 2020.

[17] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey. 2011 CWE/SANS top 25 most dangerous software errors. `https://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.html` [Online; accessed on October 4, 2021], 2011.

[18] Shellphish Contributors. How2heap. `https://github.com/shellphish/how2heap` [Online; accessed on October 4, 2021], 2017.

[19] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html` [Online; accessed on October 4, 2021], 1996.

[20] G. L. Steele Jr. Data representations in PDP-10 MacLISP. In *Proc. of the 1977 MACSYMA Users' Conference, Berkeley, California, USA*, pages 203–224. NASA, July 1977.

[21] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices*, 35(11):117–128, November 2000.

[22] G. J. Duck and R. H. Yap. Heap bounds protection with low fat pointers. In *Proc. of the 25th International Conference on Compiler Construction, Barcelona, Spain*, pages 132–142. ACM, 2016.

[23] Y. Younan, W. Joosen, and F. Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proc. of the 2006 International Conference on Information and Communications Security (ICIS'06), Raleigh, North Carolina, USA*, volume 4307 of *Lecture Notes in Computer Science*, pages 379–398. Springer, December 2006.

[24] S. Zonouz, M. Zhang, P. Sun, L. Garcia, and X. Liu. Dynamic memory protection via Intel SGX-supported heap allocation. In *Proc. of the 16th IEEE International Conference on Dependable, Autonomic and Secure Computing, the 16th IEEE International Conference on Pervasive Intelligence and Computing, the 4th IEEE International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech'18), Athens, Greece*, pages 608–617. IEEE, August 2018.

[25] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the 2005 USENIX Annual Technical Conference (USENIX ATC'05), Anaheim, California, USA*, pages 17–30. USENIX Association, April 2005.

[26] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12), Boston, Massachusetts, USA*, pages 309–318. USENIX Association, June 2012.

[27] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov. Memory tagging and how it improves C/C++ memory safety. `https://arxiv.org/pdf/1802.09517.pdf` [Online; accessed on December 10, 2021], 2018.

[28] E. D. Berger. HeapShield: Library-based heap overflow protection for free. Technical Report UM-CS-2006-028, University of Massachusetts, 2006.

[29] N. Nikiforakis, F. Piessens, and W. Joosen. HeapSentry: kernel-assisted protection against heap overflows. In *Proc. of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'13), Berlin, Germany*, volume 7967 of *Lecture Notes in Computer Science*, pages 177–196. Springer, July 2013.

[30] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: protecting critical data in unsafe languages. *ACM SIGOPS Operating Systems Review*, 42(4):219–232, May 2008.

[31] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and efficiently protecting the heap. *ACM SIGOPS Operating Systems Review*, 40(5):207–218, October 2006.

[32] Q. Zeng, G. Kayas, E. Mohammed, L. Luo, X. Du, and J. Rhee. HeapTherapy+: Efficient handling of (almost) all heap vulnerabilities using targeted calling-context encoding. In *Proc. of the 49th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks (DSN'19), Portland, Oregon, USA, pages 530–542. IEEE, June 2019.

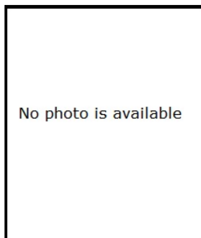
[33] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. HeapHopper: Bringing bounded model checking to heap implementation security. In *Proc. of the 27th USENIX Security Symposium (USENIX Security'18), Baltimore, Maryland, USA*, pages 99–116. USENIX Association, August 2018.

[34] I. Yun, D. Kapil, and T. Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *Proc. of the 29th USENIX Security Symposium (USENIX Security'20)*, pages 1111–1128. USENIX Association, August 2020.

[35] C. Wellons. Prospecting for hash functions. `https://nullprogram.com/blog/2018/07/31/` [Online; accessed on April 13, 2021], 2018.

Author Biography

Johannes Bouché completed his Master of Science in High Integrity Systems at the Frankfurt University of Applied Sciences in 2020 and is since pursuing a PhD there. His research focuses on the security of embedded systems.



Lukas Atkinson completed their Master degree in Computer Science at the Frankfurt University of Applied Sciences and is since pursuing a PhD there. Research interests cover systems programming and privacy-preserving technologies.

Martin Kappes is Professor at the Frankfurt University of Applied Sciences and heads the research group for network security, information security, and data privacy. His research interests include evolutionary computing techniques for network security, anomaly detection, and applied information security in enterprise networks. He holds degrees in Computer Science and Economics and a PhD in Computer Science from the University of Frankfurt. He has published more than 70 articles in scientific journals and conference proceedings.