Shadow-Heap: Preventing Heap-based Memory Corruptions by Metadata Validation

Johannes Bouché
Frankfurt University of Applied Sciences
Frankfurt a. M., Germany
johannes.bouche@fb2.fra-uas.de

Lukas Atkinson
Frankfurt University of Applied Sciences
Frankfurt a. M., Germany
lukas.atkinson@fb2.fra-uas.de

Martin Kappes
Frankfurt University of Applied Sciences
Frankfurt a. M., Germany
kappes@fb2.fra-uas.de

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 Shadow-Heap 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.

ACM Reference Format:
Johannes Bouché, Lukas Atkinson, and Martin Kappes. 2020. Shadow-Heap: Preventing Heap-based Memory Corruptions by Metadata Validation. In *European Interdisciplinary Cybersecurity Conference (EICC 2020), November 18, 2020, Rennes, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3424954.3424956

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* [26],[1],[17]: 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 [31]. 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 [28] or statically prove memory safety guarantees [27]. Alternatively, memory safety violations can be found through static analysis tools or through dynamic analysis tools such as Valgrind [29] and especially through targeted fuzzing [30]. Other violations can be prevented sufficiently through operating-system level techniques such as Address Space Layout Randomization (ASLR) [31]. 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[32] or shadow stacks[33] are available. Heap-based attacks are well known since 2005 [3][2][4], but the few available mitigation techniques have seen little use, also because of high overhead.

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 Shadow-Heap 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*[17], *DieHarder*[18], and *FreeGuard*[19].

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

EICC 2020, November 18, 2020, Rennes, France

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 [1]. Although these kind of attacks are published and well understood for years they are still prevalent in real world scenarios and productive environments [5]. In cause 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 [3] [4] [24] 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 [6] [7] [8] as well as several publications dealing with the mitigation of specific attacks or attack principles [13] [14] [16]. Forensic tools like Purify [12], Valgrind [9] and AddressSanitizer [10] are able to detect some but not all of the relevant attacks. Other techniques include N-Versioning and replication [20], leveraging the OS-kernel [21], automated error-correction approaches [15], external processes and Client-Server architectures [22], offering codeless patching [23] or model-driven detection approaches [11].

DieHard and FreeGuard are especially remarkable and are able to prevent a majority of heap-based attacks. In contrast to our lightweight mitigation approach, The design of DieHard aims to increase the "likelihood of correct execution" instead of detecting errors reliably [17][18], and no longer works consistently on current Linux systems. Freeguard implements a BiBOP heap, allowing the chunk metadata to be separated [19]. However, our approach does not make it necessary to re-implement the entire allocator. Both also have relatively high performance or memory overhead, especially compared to newer GNU libc 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* [25] 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 optimze 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 application data. The header is extended to include

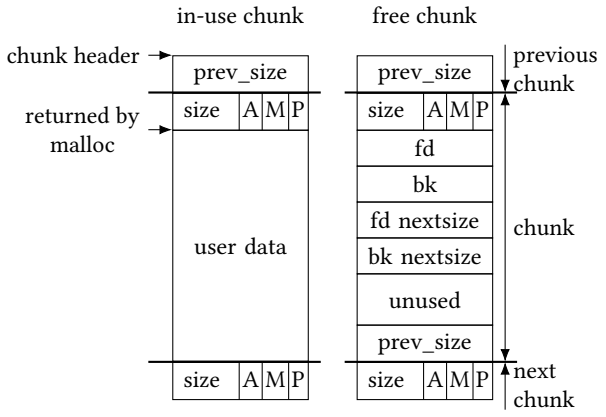


Figure 1: Chunk metadata layout

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

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 [25].

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 [24] 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 corrupts heap metadata or leads to other unwanted behavior. E.g. the allocator could later return this same chunk once as a pointer that the attacker can already read and again for other data that the attacker wants to access.

Invalid-Free (IF): A pointer to a forged memory chunk is freed, corrupting heap metadata or leading to other unwanted behavior.

Buffer Overflow (BO): An out-of-bounds write corrupts the heap metadata or the application data in that chunk.

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. Under certain conditions this can cause the memory allocator to return new chunks that overlap with currently in-use chunks.

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).

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

Category A (1–7): Injection of fake or duplicate chunks via double-free or invalid-free.

Category B (8–12): Use of buffer overflows against freed chunks while they are stored on the unsorted bin, so that the allocator uses manipulated metadata during consolidation.

Category C (13): Insertion of fake chunks into the tcache via a use-after-free buffer overflow.

Category D (14): Manipulation of the topchunk size via a buffer overflow.

Category E (15–16): Use-after-free buffer overflows targeting chunks that are not on the unsorted bin.

Category F (17–18): Buffer overflow to overwrite the *prev in use* flag.

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

ID	name	weakness	glibc 2.25	glibc 2.26	outcome
1	fastbin_dup	DF	yes	yes	Duplication
2	fastbin_dup_consolidate	DF	yes		Duplication
3	fastbin_dup_into_stack	DF	yes		Arbitrary buffer
4	house_of_spirit	IF	yes		Fake buffer
5	tcache_dup	DF		yes	Duplication
6	tcache_house_of_spirit	IF		yes	Fake buffer
7	house_of_botcake	DF		yes	Arbitrary buffer
8	overlapping_chunks	BO	yes		Overlap
9	overlapping_chunks_2	BO	yes		Overlap
10	poison_null_byte	BO	yes		Overlap
11	unsorted_bin_attack	BO,UAF	yes		Memory write
12	unsorted_bin_into_stack	BO,UAF	yes		Fake buffer
13	tcache_poisoning	BO,UAF		yes	Arbitrary buffer
14	house_of_force	BO	yes		Arbitrary buffer
15	house_of_lore	BO,UAF	yes		Arbitrary buffer
16	large_bin_attack	BO,UAF	yes	yes	Memory write
17	unsafe_unlink	BO,UAF	yes		Arbitrary write
18	house_of_einherjar	OB1	yes		Overlap

5 PROTOTYPICAL IMPLEMENTATION

In this section we discuss our prototypical Shadow-Heap implementation. This approach achieves mitigation against all attacks in category A–D. Since our approach protects the underlying data structures rather than detecting individual attacks, these mitigations might also be able to protect against future attacks. We verified the mitigations using the exploits in how2heap [24]. Although how2heap assumes glibc 2.25/2.26, our approach is also known to work under 2.30. In the following, we describe our mitigation approaches, core components of our implementation, and the used data structures. Finally, we discuss limitations and evaluate our design goals.

Our mitigations rely on storing a snapshot of chunk metadata in a Shadow-Heap. In the free protection, we store metadata of in-use chunks. For the bin protection, we store metadata of freed chunks that are stored in a particular bin. Similarly, the topchunk protection refers to metadata of an unallocated chunk. In the following, the protection schemes are described in more detail.

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

The performance of the free protection scheme depends drastically on the backing data structure. By using a compact hash table, it is possible to perform insertions and removals/retrievals including validation in (amortized) $O(1)$ time, with only 16 bytes used per entry.

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. In general, bin protection could be extended to other bin types (fast, small, large). While their bin sizes ultimately depend on the application's allocation patterns, they are unbounded in practice so that the performance–security tradeoff is disadvantageous.

For each allocating or releasing call the elements from these bins have to be persisted into the Shadow-Heap. 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. 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.

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.

Our implementation consists of wrapper, facade, and leak components. Figure 2 provides an overview of our architecture and data flows. (1) The imported allocator functions are hooked so that all calls are redirected through the Shadow-Heap. (2) We leak the allocator's *main arena*. (3) Allocation requests from the user application are intercepted by the Shadow-Heap. (4) Shadow-Heap ultimately satisfies the request by using the original allocator.

Facade: Main component of Shadow-Heap. Manages initialization, metadata storage, and feature flags. Provides hooks for the allocation lifecycle which save and validate metadata.

Wrapper: Provides entry points into the allocator API. Calls into the facade's handlers to validate metadata upon entry into the API, and save the changed metadata in the shadowed copy before returning. To account for internal allocations, recursive calls are satisfied without metadata validation.

Leak: Provides access to internal allocator data structures (main arena), which is required for bin and topchunk protection. Uses a use-after-free attack to reveal the arena location during Shadow-Heap initialization, and accounts for layout differences between glibc versions. However, a patched libc is required to leak tcache prior to glibc 2.28.

Feasibility of performing free-protection depends crucially on an efficient data structure to store metadata. Practical experiments indicated that standard library data structures were too inefficient for our use case, so that a custom storage layer was designed. While the optimal data structure ultimately depends on the specific allocation patterns, hash tables are a good general choice due to attractive worst-case behavior. Our custom hash cache 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, entries are evicted to a full hash table.

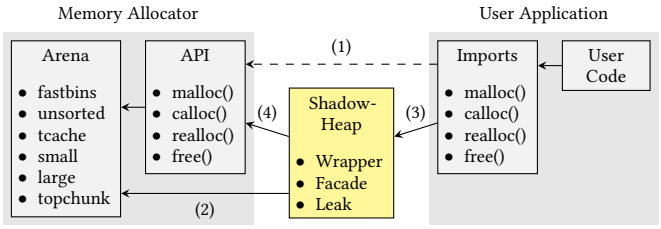


Figure 2: Shadow-Heap working principle

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 Shadow-Heap 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 Shadow-Heap, 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.

6 PERFORMANCE EVALUATION

To evaluate the overhead of applying the Shadow-Heap mitigation, a number of real-world examples and benchmark workloads were executed in mitigated and non-mitigated configurations. The test system used Linux 5.3.0, a i7-8565U CPU, 8G DDR4-2400 RAM, and native glibc 2.30. The prototypical Shadow-Heap implementation can protect nearly all single-threaded POSIX-compliant programs.

The *malloc* experiment performs synthetic allocation/deallocation patterns within a single process. The *perlbench*[1] and *Octane*[2] benchmark suites measure the performance of Perl 5.28 and JavaScript runtimes, respectively. As a JavaScript engine, SpiderMonkey 60.8 is used in single-threaded mode. Whereas perlbench focuses on micro benchmarks, Octane includes a mix of complete programs. While both Perl5 and SpiderMonkey use malloc, they also overlay their own memory management techniques in order to implement garbage collection. The *GCC* benchmark builds the GCC-9.2 compiler suite. The build process involves a large array of command line tools. The compilation process itself uses malloc heavily. In the *tar* experiment, a large file archive is extracted with GNU tar 1.30 as a CPU-limited workload.

Shadow-Heap is used in various configurations in order to provide insight into the performance effect of these mitigations. First, a baseline without any mitigations is measured. Then, Shadow-Heap

[1] https://github.com/gisle/perlbench at commit d4033ad (2012)
[2] https://github.com/chromium/octane at commit 30b1d8d (2017)

Table 3: Median run time and median peak memory usage at various Shadow-Heap mitigation levels, relative to baseline.

	run time					peak mem	
	L1	L2	L3	L4	FG	L1	FG
malloc	1.64	1.75	4.70	15.24	1.90	1.26	5.09
gcc	1.13	1.13	1.19	3.71	2.50	1.03	1.44
octane	1.01	1.01	1.02	1.07	1.00	1.00	1.05
perlbench	1.07	1.06	1.09	1.57	3.25	1.17	2.21
tar	1.01	1.01	1.01	1.04	1.00	1.03	1.05

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

mitigation	time	mem	mitigation	time	mem
none	1.00	1.00	L1	1.64	1.26
hooked	1.10	1.08	L2	1.75	1.26
diehard	2.59	1.60	L3	4.70	1.26
dieharder	8.33	3.25	L4	15.24	1.26
freeguard	1.90	5.09	ptr	1.79	1.26
hash-ptr	2.90	1.44	top	1.31	1.04
tree-ptr	4.66	1.51	usb	4.29	1.04
			tca	9.72	1.04

mitigations are successively activated in levels L1–L4 (free protection, topchunk protection, unsorted bin protection, and tcache protection). For the malloc experiment, these are also tested individually (abbreviated as ptr, top, usb, and tca, respectively). The malloc test additionally measures the overhead of hooking into the malloc library functions without performing mitigations, and the impact of different data structures for the free protection (hash: `std::unsorted_map`, tree: `std::map`). Alternative mitigation approaches include Diehard, Dieharder, and FreeGuard (FG). They were loaded into the experiments, but only FreeGuard could be used successfully in all experiments.

To allow for easier comparability, measurements are normalized on the median baseline measurement. Figure 3 and Table 3 show the impact of the mitigation levels on the total run time (wall time). Octane and tar see negligible overhead, even at L4. Under GCC, Shadow-Heap runs faster than FreeGuard up to L3, and similarly under perlbench up to L4. For the malloc experiment, we outperform FreeGuard up to L2. In all scenarios, Shadow-Heap consumes less memory than FreeGuard (e.g. 3% vs. 44% for GCC). Figure 4 and Table 4 show detailed results for the malloc experiment. Individual feature measurements indicate that memory overhead is largely due to free-protection, but that compared to standard library equivalents, the data structure is both smaller (44%, 51% vs. 26%) and faster (190%, 366% vs. 79%). While FreeGuard and DieHard run faster than L3 and DieHarder runs faster than L4, their memory overhead is substantially higher than Shadow-Heap (409%, 60%, and 225% vs. 26%). Over all experiments other than perlbench, L4 is the slowest mitigation.

7 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[3]. Depending on the application, the performance impact of Shadow-Heap can range from unnoticeable to substantial, but in practice the memory overhead seems to be reasonably low (0%–17%, excluding the malloc test). Thus, Shadow-Heap 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, Shadow-Heap has a number of limitations. The overhead is largely due to the wrapper approach, and would be avoidable if these security features were directly integrated into the malloc allocator. It is also not possible to reasonably protect the prevsize flag in the chunk metadata without such integration, which might be exploitable by novel attacks. Wrapping overhead is especially evident with the tcache protection which has to completely traverse malloc's linked lists in order to check all elements. Protection for large bins and small bins has not been implemented as their size is unbounded, unlike the tcache and the unsorted bin.

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 [16]. Aside from protecting glibc's ptmalloc implementation, similar mitigations could be implemented with other allocators such as jemalloc, Windows Heap, and BIBOP. 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.

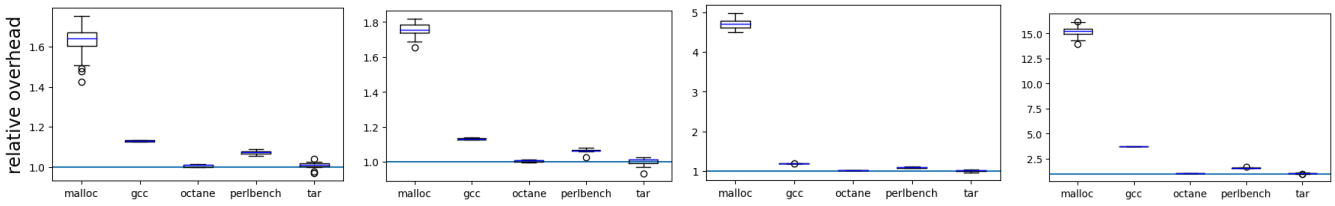
ACKNOWLEDGMENTS

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

8 REFERENCES

[1] Matthew S Simpson and Rajeev K Barua. Memsafe: ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.
[2] Justin N Ferguson. Understanding the heap by breaking it. *black Hat USA*, pages 1–39, 2007.
[3] Phantsmal Phantasmagoria. The malloc maleficarum. *Bugtraq mailinglist*, 2005.
[4] Mathias Frits Rørvik. Investigation of x64 glibc heap exploitation techniques on linux. Master's thesis, 2019.
[5] Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011 cwe/sans top 25 most dangerous software errors. *Common Weakness Enumeration*, 7515, 2011.
[6] Doug Lea and Wolfram Gloger. A memory allocator, 1996.
[7] Guy Lewis Steele Jr. Data representations in pdp-10 maclisp. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1977.
[8] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 117–128. ACM, 2000.

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



(a) L1 mitigation level (b) L2 mitigation level (c) L3 mitigation level (d) L4 mitigation level

Figure 3: Run time at different mitigation levels relative to baseline.

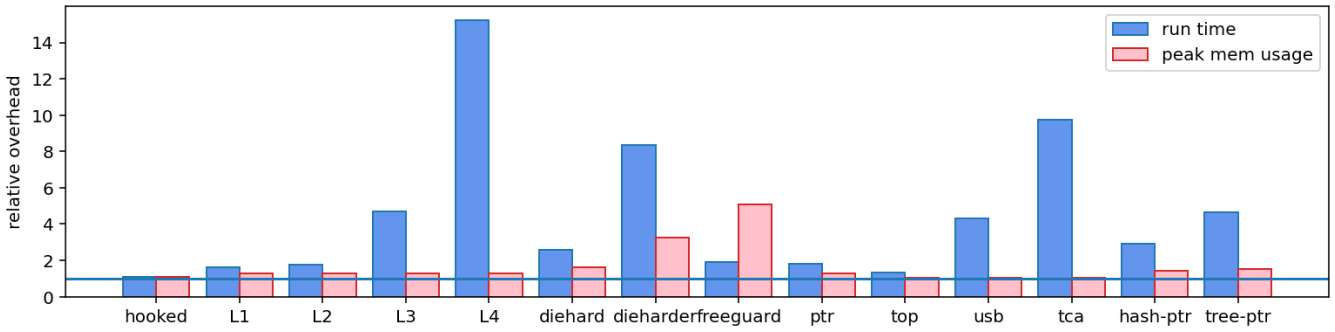


Figure 4: Overhead of different mitigations for the malloc experiment (median over 100 repetitions) relative to baseline.

[9] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, pages 17–30, 2005.

[10] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[11] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, 2018.

[12] Todd M Austin, Scott E Breach, and Gurindar S Sohi. *Efficient detection of all pointer and array access errors*, volume 29. ACM, 1994.

[13] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142. ACM, 2016.

[14] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *International Conference on Information and Communications Security*, pages 379–398. Springer, 2006.

[15] Karthik Pattabiraman, Vinod Grover, and Benjamin G Zorn. Samurai: protecting critical data in unsafe languages. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 219–232. ACM, 2008.

[16] Saman Zonouz, Mingbo Zhang, Pengfei Sun, Luis Garcia, and Xiruo Liu. Dynamic memory protection via intel sgx-supported heap allocation. pages 608–617, 08 2018.

[17] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

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

[19] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403. ACM, 2017.

[20] Emery D Berger. Heapshield: Library-based heap overflow protection for free. *UMass CS TR*, pages 06–28, 2006.

[21] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: kernel-assisted protection against heap overflows. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 177–196. Springer, 2013.

[22] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. Comprehensively and efficiently protecting the heap. *ACM SIGOPS Operating Systems Review*, 40(5):207–218, 2006.

[23] Qiang Zeng, Golam Kayas, Emil Mohammed, Lannan Luo, Xiaojiang Du, and Junghwan Rhee. Heaptherapy+: Efficient handling of (almost) all heap vulnerabilities using targeted calling-context encoding. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 530–542. IEEE, 2019.

[24] Team Shellphish. How2heap. https://github.com/shellphish/how2heap, 2017.

[25] Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1996.

[26] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[27] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. *ACM SIGPLAN Notices*, 38(7):69–80, 2003.

[28] Pieter H Hartel and Luc Moreau. Formalizing the safety of java, the java virtual machine, and java card. *ACM Computing Surveys (CSUR)*, 33(4):517–558, 2001.

[29] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[30] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Rhongai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[31] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[32] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[33] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.