SdShield: Effectively Ensuring Heap Security via Shadow Page Table

Linong Shi, Chuanping Hu, Yan Zhuang(✉), and Yan Lu

Zhengzhou University, Zhengzhou, China
yan.zhuang@zzu.edu.cn

Abstract. Heap security has become a serious threat in recent years. To address the problem of heap vulnerabilities that are hard to detect and mitigate, this paper proposes a new heap protection scheme using shadow page tables. This scheme builds on the traditional idea of page permission and designs a novel shadow page table structure that stores the virtual address and random value of each object. This enables checking the boundaries and validity of heap objects, and effectively detects various types of heap-related attacks, such as heap overflow, use-after-free, invalid free, and double free. In addition, the scheme adopts a dynamic system call addition method, which is not dependent on specific runtime environments or kernel modifications, and has high scalability and portability. Experimental evaluation on various applications shows that our proposed scheme is effective in detecting many types of heap vulnerabilities, providing more comprehensive security with low performance overhead than comparable solutions.

Keywords: Heap vulnerabilities · Shadow page table · Page protection

1 Introduction

Heap-related memory corruption vulnerabilities have remained a serious threat and caused many catastrophic exploits in the past decades [5, 16]. The number of heap-related vulnerabilities has grown in recent years, out-of-bounds write bugs and use-after-free bugs were ranked first and seventh in the CWE top 25 of the most common and impactful issues in software [36]. Attacks that exploit these vulnerabilities not only affect the normal execution of programs, but may also lead to sensitive data leakage, control flow hijacking, arbitrary code execution, and other impacts.

So far, many studies have focused on solving heap security problems through memory allocators [1–5, 10, 17–21, 23]. Robertson et al. [20] proposed using canaries and checksums to detect possible buffer overflows. Gene et al. [2] provide probabilistic memory safety through the random layout of heap area and randomized object placement and reuse. Sam et al. [3] borrowed the idea of "freelist" from performance allocators and shadow memory technology, providing better performance than previous secure allocators. Liu et al. [4] employed an efficient fine-grained class indexing scheme and implemented a dynamic canary scheme, further increasing security. Such schemes achieve

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
S. Goel and P. R. Nunes de Souza (Eds.): ICDF2C 2023, LNICST 571, pp. 114–131, 2024.
https://doi.org/10.1007/978-3-031-56583-0_8

heap area protection by overriding the system's allocator, however, they change the system's default memory allocation strategy, which may result in overhead and compatibility problems.

On the other hand, other researchers have proposed various solutions that target a single vulnerability by combining some form of static analysis and runtime checking [6–9, 11, 28–33]. Nikiforakis et al. [7] prevent heap overflows by detecting the state of heap objects during the system calls. Tian et al. [33] monitor heap buffer overflows using multicore technology and concurrent monitoring algorithms. Lee et al. [8] prevent the use of dangling pointers by tracking the relationship between pointers and objects and invalidating the corresponding pointers when released. Dang et al. [14] and Gorter et al. [15] use the page protection scheme by creating a new virtual page (called a shadow virtual page) for each memory allocation to prevent use-after-free errors. However, most of these solutions either require a specific environment or are compiler-based and cannot handle compiled binaries.

In this paper, we introduce SdShield, an efficient, portable, and more comprehensive heap protection scheme that builds on the traditional idea of page permission. This scheme design a novel shadow page table structure that stores the canonical address and the random value of each object in the process shadow page table, which allows for higher security and quicker information access. We adds an interceptor between the target program and the memory allocator to intercept all memory allocation and deallocation requests from the program without needing the program source code. Specifically, we implement information transfer between user space and kernel space using dynamically added system calls that do not require a specific environment or kernel recompilation and can be easily adopted in desktop and server environments.

In brief, this paper makes the following contributions:

(1) A new heap protection scheme using shadow page tables, which utilizes page management to provide more comprehensive heap protection.
(2) A novel shadow page table structure design, which can conveniently store and access relevant information of heap objects.
(3) Using the cooperative scheme between user space and kernel space, dynamic management of program heap memory and page tables without program source code.
(4) Using NIST Juliet test suite and SPEC CPU 2006 benchmark, the security and performance of this scheme are studied in detail.

The rest of this paper is organized as follows. In Sect. 2 we introduce the background and present the threat model of SdShield. In Sects. 3 and 4 we respectively present the design and implementation details of SdShield, and then evaluate its security and performance consumption in Sect. 5. We finally discuss the current limitations of SdShield in Sect. 6 and conclude in Sect. 7.

2 Background

In this section, We detail the background of SdShield, including the common heap-related vulnerabilities and the page table structure. Then discuss the development of the methods that protect heap by page permissions, and point out their limitations that drive the design of SdShield. Finally, we present the threat model of SdShield.

2.1 Heap-Related Vulnerabilities

Spatial heap-based vulnerabilities include heap overflows and heap over-reads. Heap overflows occur when a program performs an out-of-binding write operation to a heap object due to an error or lack of proper boundary checks [4]. The less common heap underflow is when incorrect access to the heap buffer results in a write to memory before the buffer. Heap overflows overwrite memory space that does not belong to the object and may lead to many security issues. Similarly, if built-in boundary checks for memory accesses are lacking, over-reads will occur when accessing heap objects.

Temporal heap-based vulnerabilities include use-after-free (or dangling pointer), double free, and invalid free. Use-after-free is caused by the dereference of a heap-allocated object, because the pointer (now called a dangling pointer) to the freed object remains unchanged even though the memory location pointed to is no longer valid [15], so the freed memory can be forced to be used by the corresponding dangling pointer. Double free is considered a special case of use-after-free, which occurs when an object is freed twice [3]. Invalid free occurs when an application attempts to free a value that does not point to an object created by the memory allocator.

2.2 Page Table

Most modern operating systems use the paging mechanism, under which both virtual and physical memory is divided into fixed-size chunks called Pages. The mapping between virtual and physical pages is established through the Page Table, which are data structures maintained by the system and stored in the kernel space. The memory management unit (MMU) of the CPU uses them to obtain the mapping relationship between virtual memory and physical memory. Each process has a separate page table that describes the process address space, and the kernel has its own set of page tables that manage the kernel space. Some modern CPUs also have 5-level page tables, but most 64-bit architectures use 4-level page tables. In the remaining part of this paper, we assume a 4-level page table structure, which is commonly referred to as the Page Global Directory (PGD), the Page Upper Directory (PUD), the Page Middle Directory (PMD), and the page table (PT).

2.3 Page Permission Schemes

In early work, Perens et al. [23] proposed a debugging tool that uses the page protection mechanism to detect dangling pointer errors. The tool allocates a new virtual and physical page for each allocation of the program. However, this scheme means that even small allocations would occupy a whole page of actual physical memory. This would cause

a drastic increase in the memory consumption of the application, making it impractical for real production environments. Following this idea, Dhurjati et al. [12] proposed a key improvement, where they mapped each allocation using a new virtual page to the same physical page as the original allocator to solve the physical memory consumption problem. And they used a compiler transformation called automatic pool allocation to mitigate the problem of virtual address space exhaustion.

In recent years, Dang et al. [14] demonstrated the theoretical basis for such schemes and argued that page permissions should be the ideal method in principle. Unlike Dhurjati's automatic pool allocation, Dang's design can unmap shadows immediately after object release. They also discussed compatibility with fork, which seems to be an unknown limitation in Dhurjati's scheme. Then Gorter et al. [15] provided a more efficient implementation that relies on direct page table access in ring 0. In addition, Gorter introduced a new garbage collection style recycling module, which can safely reuse the freed areas and address the key scalability issues that plague such solutions.

However, all existing solutions based on page permissions lack portability, need to run in a specific environment (such as KML), or require kernel editing. Furthermore, all such solutions currently focus only on mitigating the use-after-free bugs while ignoring the dangers of other vulnerabilities. In this paper, we show that these limitations are not fundamental and that an efficient, portable, and more comprehensive heap protection solution can be unlocked by using shadow virtual page tables.

2.4 Threat Model

It is well-known that security by obscurity is not a good practice, so we hypothesize that it is possible for the attacker to access the sources of SdShield. We assume a standard threat model [15], in which the host operating system (such as Linux) is trusted; and the attacker can perform any arbitrary operation on the heap area of the victim program (including allocation, free, read and write, such as), and try to achieve information leakage and privilege escalation through the vulnerabilities. We consider all vulnerabilities in the heap area, whether buffer overflow, use-after-free, or some other types of attack technique, and assume that the program has no other vulnerabilities (such as stack).

3 Overview

SdShield is a scheme based on page protection policies designed to detect and prevent heap vulnerabilities through the cooperation between user space and kernel space. The structure of SdShield is divided into two parts: the interceptor running in user space and the kernel module running in kernel space, and the interaction process is mainly divided into two phases: allocate and release. Figure 1 describes the main components of SdShield and their interaction process.

When the application requests memory allocation, the interceptor intercepts all such requests and sends them to the system allocator. The system allocator requests virtual and physical memory from the operating system and then returns the allocated heap area address, which we call the canonical virtual address of the object, to the interceptor. After receiving the allocated address, the interceptor passes the relevant data to the kernel

module using a system call. The kernel module creates several shadow page table entries in the process page table, calculates an unused address in the shadow virtual space, and returns it to the interceptor. Finally, the interceptor transparently returns the corresponding shadow address to the user program, so that the user program can only use this address to reference the allocated object, without knowing that it is related to the shadow page table.

In the application memory release phase, when the interceptor intercepts a memory release request, it first uses a system call to pass the relevant data to the kernel module, and the kernel module modifies the valid bit in the process page table to invalidate its mapping. It then obtains the canonical virtual address of the object from the page table entry and returns it to the interceptor. Finally, the interceptor passes the canonical virtual address to the default allocator, so that its physical memory is released by the system allocator and can be reused.

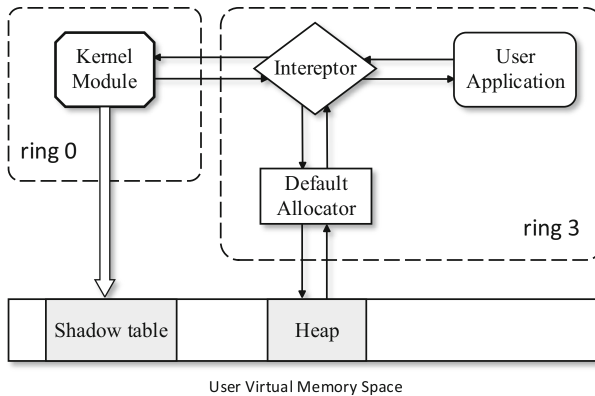


Fig. 1. Structure of SdShield's components

We provide SdShield's interceptor as a shared library for applications, so that SdShield can be used with only existing binaries, without scanning the program source code or modifying the default allocator, and is theoretically compatible with all allocators of modern operating systems as well as those described in the literature.

In addition, SdShield requires system calls for communication between the interceptor and the kernel module. However, modifying the operating system source code contradicts SdShield's philosophy of portability, so we dynamically add system calls by modifying the syscall_table array in the kernel without recompiling the kernel. If this method of modifying kernel instructions adversely affects some production environments or workloads, user program access to kernel space can be achieved by recompiling the kernel or using Kernel Module Linux [14, 15, 34].

3.1 Shadow Address for Object

The core idea of SdShield is the shadow virtual address [12, 14, 15]. Typically, virtual pages and physical pages are in one-to-one correspondence, while using shadow

address allows mapping the same physical page frame to multiple virtual pages. Figure 2 illustrates the relationship between the shadow address, canonical address, and physical address of each object. We refer to the virtual address generated by SdShield as the shadow virtual address, and the address allocated by the system default allocator as the canonical virtual address of the object. SdShield places each object allocated by the program in a different shadow virtual page, but these shadow pages are all mapped to the same physical pages as the original allocator, and their offsets remain the same as the offsets of the actual physical pages. This allows each object to have an independent shadow page, to achieve the purpose of managing each object individually through page permissions. While multiple objects can be located on the same physical page, making the program physical memory consumption almost the same as the original.

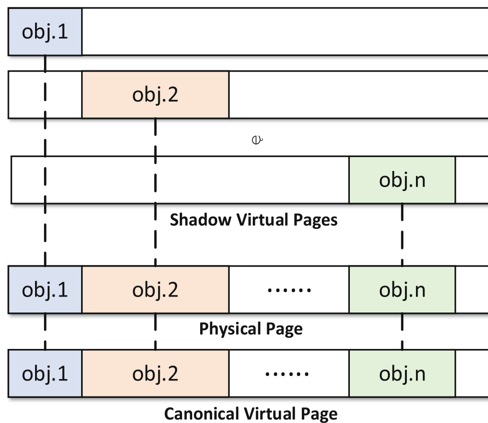


Fig. 2. The object in the virtual pages and physical pages

Generally, multiple small memory objects share a system page, but it is possible in process that there are large objects which occupy multiple pages. If an object spans multiple pages, SdShield will allocate the same number of shadow pages as the physical pages and keeps its in-page offset consistent with the original. When the application frees an object, we need the corresponding canonical virtual address to initiate the release process from the default allocator. For this purpose, we add a shadow page table entry for each object that stores the canonical virtual address. Note that placing a canonical virtual address as an entry into the page table requires a separate flag bit to locate it and that the page table entry should always be in the page-fault status.

3.2 Security Features

The security features implemented in SdShield are as follows: Add a random value at the end of each allocated heap object and check it at each malloc and free to prevent overflow. Create different shadow virtual pages for each heap object and prevent use-after-free by invalidating the mapping of its shadow page to the actual physical page as soon as the object is freed. Use unmapped protection pages at the end of each object,

to prevent overflow, over-read and spray attacks, and free checks to prevent double free and invalid free, are detailed below.

Random Canary. SdShield borrows a common mechanism from the existing secure allocators to prevent potential buffer overflow errors: it adds an extra random value at the end of each allocated heap object, called canary [3, 4, 17]. SdShield generates a random value from the kernel entropy and places it at the end of each object—meaning that each malloc(n) call is changed to malloc(n + sizeof(void*)). Unlike the secure allocators, SdShield stores the random values in the kernel space, so it is not affected by the processes and attackers. Figure 3 shows the canary in the heap area and the shadow page table. Similar to the canonical virtual address, SdShield also saves the random value as a separate page table entry in the shadow page table, and keeps the entry always in a page-fault status.

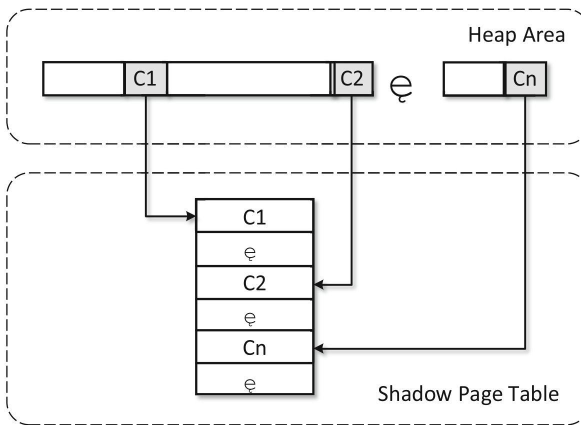


Fig. 3. Canary in heap area and shadow page table

For the most part, using canary to prevent data overflow of applications is effective. Although the timeliness is limited by the frequency of canary checks, we can always detect whether its canary has been modified when an object is freed. For each memory request or release in the program, SdShield checks all the random values in the same page table of that object, so the number of objects checked each time may be between 1 and 173 (the case where only one object in a single shadow page table and the case where a single shadow page table is full). If any canary is modified, SdShield reports information about the potential buffer overflow location and optionally stops the execution of the current program.

Shadow Page Permissions. The memory management unit (MMU) in most modern processors perform a runtime check for each memory access. One idea is to use the operating system page access and protection policy to detect and prevent use-after-free, but MMU can only manage access permissions at the granularity of page (typically 4096 bytes). Placing each object on a separate physical page would result in a significant memory over-head caused by fragmentation.

In SdShield, due to the shadow virtual pages, each object can have a separate page mapping, so the object granularity of mapping can be disabled through page permission management. SdShield prevents the use-after-free by deleting the application's access to the freed heap object. When the corresponding object is released, SdShield sets the valid bit of the shadow page to zero, and all subsequent accesses to this page will generate invalid access information and cause MMU to trigger an exception.

Guard Pages. Guard page is an unmapped virtual memory page that is placed before or after an allocated heap object. Guard pages can prevent heap overflow, over-read, and heap spraying attacks, as any access involving guard pages will immediately trigger a page fault. Since guard pages are not mapped to actual physical pages, they only consume the size of a page table entry (8B) instead of a whole page (4 KB). Normally they can only be placed with page granularity, so it is not possible to place guard pages before and after every allocated heap object.

In some secure allocator schemes [3, 4, 17], guard pages need to be explicitly placed by using the mprotect system call. While in SdShield the canonical virtual address entries and random value entries placed before and after each object in the shadow page table just serve the purpose of guard pages, so that protection pages can be easily set at object granularity and all accesses to them will result in exceptions.

Free Check. When the application performs free operations, SdShield prevents double and invalid releases by checking the status marker bits of each object's shaded page table entry. SdShield can easily detect the following exceptions: The pointer of the free operation points to a range outside the heap area; The shadow page table entry corresponding to the free pointer is in the unallocated status; The shadow page table entry corre-sponding to the free pointer is already in the freed status. For each allocation, SdShield marks its shadow page table entry as in use. When deallocating, SdShield first judges its status bits and confirms whether this release is valid.

4 Implementation

In the previous section, we introduced the design ideas of SdShield and the security features SdShield uses. In this section, we continue to cover the implementation details of SdShield, including the selection of the shadow area, the structure of the shadow page table, and the dynamic addition of system calls. In addition, this section discusses some optimizations to reduce the overhead of SdShield and to solve address space exhaustion and compatibility issues.

4.1 Shadow Page Table

To support the use of multiple processes, SdShield stores the shadow page mapping in the process page table. Each user process has an independent virtual address space, and its virtual pages to physical pages mapping are stored in its process page table, which is managed by the operating system kernel. Although the kernel module has direct access to system memory, such as page table data structures, the operating system is unaware of the module and may overwrite the mappings that SdShield changes in the user process

page tables. Therefore, SdShield chooses an unused area in the user virtual address space as the shadow page table, and directly writes into the page table entries corresponding to that area via the kernel module, thus avoiding interference from the user process and the system kernel.

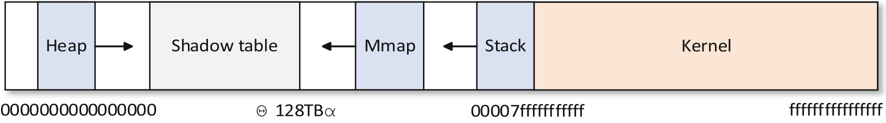
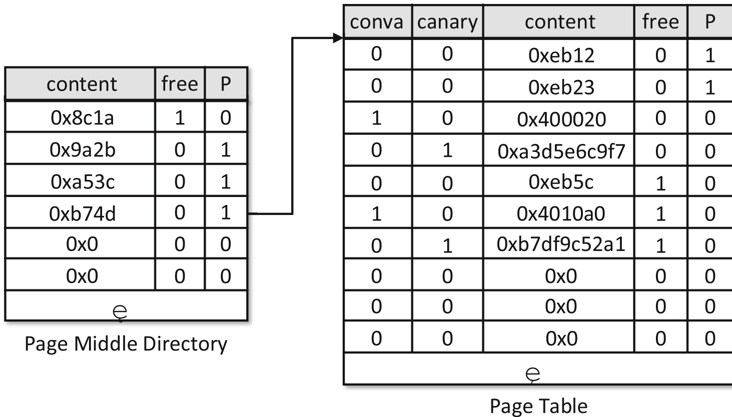


Fig. 4. Virtual memory space layout

Figure 4 shows a simplified layout of virtual memory space. Each user process has 2^47 bits of virtual address space, while the actual physical space size of most machines is less than 1 TB, so the vast majority of the process's address area is unused. SdShield utilizes a part of these free address areas as the range of the shadow page table. Considering the VMA structure of the process and methods such as ASLR, SdShield chooses between entries 96 and 160 of the Page Global Directory (PGD) as the area of the shadow page table. This area can accommodate up to 20 TB of virtual alias space, corresponding to up to two billion concurrent 4K pages.



Page Middle Directory

content	free	P
0x8c1a	1	0
0x9a2b	0	1
0xa53c	0	1
0xb74d	0	1
0x0	0	0
0x0	0	0
...		

Page Table

conva	canary	content	free	P
0	0	0xeb12	0	1
0	0	0xeb23	0	1
1	0	0x400020	0	0
0	1	0xa3d5e6c9f7	0	0
0	0	0xeb5c	1	0
1	0	0x4010a0	1	0
0	1	0xb7df9c52a1	1	0
0	0	0x0	0	0
0	0	0x0	0	0
0	0	0x0	0	0
...				

Fig. 5. PMD and PT in shadow page table

SdShield stores the status and attributes of the page table entries in the ignored bits of the shadow page table entries, thus avoiding extra memory overhead. Figure 5 shows a Page Middle Directory structure and a Page Table page structure, where the content column shows the actual content saved of each page table entry, the P flag bit indicates the presence bit, and the free flag bit indicates the release information of the page table entry. In addition, in the lowest-level page table, there are two additional flag bits (called conva and canary) used to distinguish canonical virtual address entries and random value entries from the page table. The right side of the figure describes a mixed situation of

in-use objects (PT entries 1–4), freed objects (PT entries 5–7), and available entries (PT entries 8–10).

4.2 Free and Reuse

The main limitation of using shadow virtual pages is that the used shadow address can never be reused throughout the execution of the program, because it is impossible to determine whether the program still retains pointers to the freed address. SdShield is designed with as large as 20 TB of shadow space, but due to the non-reusability of shadow address, they will still be exhausted after excessive consumption. In fact, the following two problems were found in practice: (1) Each non-reusable shadow page table entry takes up a small amount of operating system resources (page table entries). (2) Long-running programs will eventually run out of shadow address space.

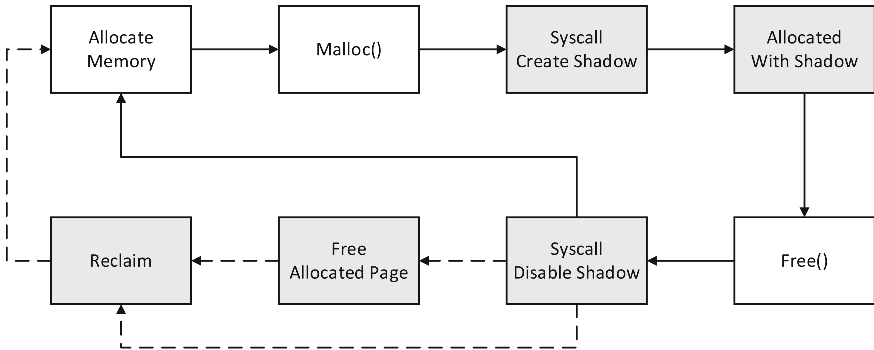


Fig. 6. The process of allocate and free memory

In order to solve the above problem, SdShield has designed the page table release and reclaim components. The main goal of the release component is to allow the safe release of page tables that are completely invalid. And the main goal of the reclaim component is to allow the safe reuse of shadow address that have already been freed. The whole process of memory allocation and release is shown in Fig. 6. Each time SdShield performs a release operation (when deleting a shadow), it determines whether the page on which the page table entry is located is releasable or not. If any one of the 512 entries of a shadow page table is in the valid status, the entire page table needs to be retained because SdShield may still need to access physical pages or obtain relevant information through it. However, when all entries in a shadow page table are free (unused or freed), retaining such page tables will waste a lot of memory space, so that the release component will run to set the release flag bit and free this page table.

In addition, in order to solve the problem that the limited shadow address space may be exhausted, SdShield's reclaim component starts when the shadow address space utilization reaches a critical point and reclaims all reusable shadow address. The reclaim is divided into two main phases: marking and scanning. In the marking phase, SdShield scans all memory of the program. For any data pointing to the shadow area, we will set a

flag bit in the corresponding shadow page table entry that represents a still referenced one. If a piece of used shadow address does not find a pointer associated with it in memory, this area is considered safe for reuse. During the scan phase, SdShield traverses all shadow page tables and adds all freed and unmarked shadow address ranges to a reusable linked list. To improve memory utilization, SdShield reclaims only completely free shadow page tables, avoiding the situation where only a few active objects exist in some page tables.

4.3 Shadow Page Fork

When a process creates a child process using the fork system call, the parent and child process share some pages, and copies of these pages are only created using copy-on-write (CoW) by the operating system when modifying them. Unfortunately, since SdShield bypasses the kernel and modifies the process page table directly, the operating system is unaware of our shadow page table, so the shadow areas created by SdShield are not copied to the child process page table. Even if we record these areas in a kernel data structure such as VMA, the child process page table entries will incorrectly point to the physical pages in the parent process.

Therefore, SdShield solves this problem by wrapping the function call of the fork and simulating the memory semantics expected by the program. Specifically, after calling the original fork function, we force triggering copy-on-write for all heap pages in the child process, so that the kernel allocates new physical page frames for these pages of the child process. Next, we traverse all shadow page table entries in the parent process, copy those page table entries that are still in use to the child process shadow page table, and remap them to the new physical page frames of the child process.

The final effect is that both the shadow and canonical virtual address of the child process remains unchanged and valid, while their mapped physical pages are separated from the parent process. Throughout the algorithm, the parent process will remain in the waiting status until all shadow page table entries in child processes are copied and mapped to new physical page frames. Thus achieving a consistent memory status for the parent and child processes.

5 Evaluation

In this section, we evaluated the security features and performance characteristics of SdShield using different benchmarks. For the security performance evaluation of SdShield, we used the Juliet Test Suite [24], as well as some real-world bugs. For the performance evaluation of SdShield, we used the SPEC CPU2006 benchmark tests. For all reported values, the average of 3 runs was used to reduce the effect of noise.

5.1 Security Evaluation

In the previous sections, we introduced SdShield and provided descriptive arguments about the attacks it covers. In this section, we quantify the security of heap protection provided by SdShield using the NIST Juliet test suite. The test suite contains hundreds

of test cases and is categorized by vulnerability type (CWE). We experimentally confirm that by running the NIST Juliet test suite v1.3 [24], SdShield can accurately detect and mitigate vulnerabilities in the heap. For heap buffer overflow errors (CWE-122) in the test suite, SdShield is always able to detect overflows to other objects' behavior when object free. For heap over-read errors (CWE-126), SdShield can detect those that cross pages. In addition, SdShield successfully detects all use-after-free (CWE-416), double free (CWE-415), and invalid free errors (CWE-590) in the test suite.

To verify the effectiveness of SdShield, we tested it on several different real-world vulnerabilities, as shown in Table 1. And, these vulnerabilities were also evaluated on other schemes, including the latest secure memory allocator (SlimGuard [4]) and the latest schemes using shadow virtual address (Dangzero [15]). We verified whether SdShield can prevent or detect potential errors in these applications. As in Guarder [17] and slimguard [4], we moved the target buffers of some of these programs from the stack to the heap. In Table 1, "reported" means that the error can be detected immediately, "delay-reported" means that the occurrence of the error cannot be detected in time, but it can be reported with a delay, "probable" means that the error is detected probabilistically, and "unreported" means that no such error is reported.

Table 1. Effectiveness evaluation on known vulnerabilities.

Application	Vulnerability	Reference	SlimGuard	Dangzero	SdShield
gzip-1.2.4	Overflow	Bugbench [25]	delay-reported	unreported	delay-reported
Libtiff-4.0.1	Overflow	CVE-2013-4243	delay-reported	unreported	delay-reported
Heartbleed	Over-read	CVE-2014-0160	probable	unreported	probable
PHP-5.3.6	Use-After-Free	CVE-2016-6290	probable	reported	reported
PHP 7.0.7	Use-After-Free	CVE-2016-5773	probable	reported	reported
Python 2.7	Use-After-Free	Issue-24613 [26]	probable	reported	reported
PHP-5.3.6	Double Free	CVE-2016-5772	reported	reported	reported
ed-1.14.1	Invalid Free	CVE-2017-5357	reported	unreported	reported

From the table, SlimGuard reports overflow, double free, and invalid free types of errors, but for use-after-free, SlimGuard can only probabilistically mitigate them through its delayed reuse and random allocation methods [4]. On the other hand, Dangzero only targets the use-after-free error but cannot detect buffer overflow and invalid free. On the contrary, SdShield can both eliminate the occurrence of use-after-free through the shadow pages and detect potential overflows through dynamic canary as well as guard pages.

5.2 Performance Evaluation

To quantify the performance overhead of SdShield in a real-world scenario, we ran the SPEC CPU2006 benchmark suite (for CPU and memory intensive real-world program

mix) on the Intel (R) Core (TM) i5–7400 CPU @ 3.00 GHz configured machine using the Ubuntu 18.04 operating system with the Linux v4.04 kernel. We measured performance by comparing the overhead of each program between the protected version and the base version without any modifications. SdShield benchmark results in SPEC CPU2006 are shown below and compared to previous related work. To avoid interference, we also disable all optional CPU mitigation.

Runtime Overhead. Similar to SdShield, there are also schemes that use allocators to secure the comprehensive heap such as freeguard [3], guarder [17], and silmguard [4], but they do not provide evaluation results based on the SPEC CPU2006 benchmark, and we were not able to rerun them on SPEC CPU2006. Therefore, we compare SdShield with Oscar [14] and DangZero [15], which use the same core principle (shadow virtual page scheme) as SdShield.

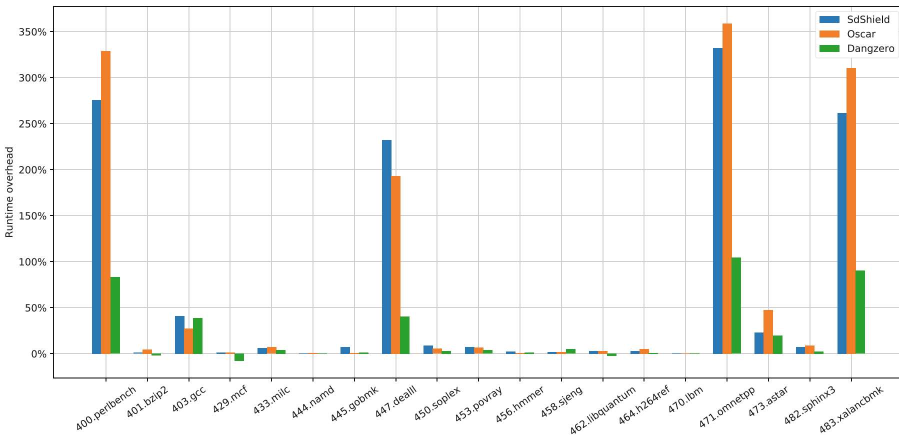


Fig. 7. SPEC CPU2006 runtime overhead for SdShield

Figure 7 compares the runtime overhead of SdShield with Oscar and Dangzero on the binary files of SPEC CPU2006. The results show that four of the 19 benchmarks incur exponential overhead due to the guard scheme. The Highly affected benchmarks, perlbench, dealII, omnetpp, and xalancbmk, are highly allocation-intensive programs, resulting in a large number of shadow page table creation and deletion operations. In addition, for some benchmark programs with few heap objects, such as bzip2, namd, hmmer, and lbm, the runtime overhead of SdShield, Oscar, and DangZero are both less than 1%, or even negligible. For the complete SPEC CPU2006 benchmark suite, the average SdShield runtime overhead was 64%. It should be noted that none of the test suites ran SdShield's reclaim component, because SPEC CPU2006 only contains short-term applications, and the virtual address space usage in the system did not reach the configuration limit of the reclaim component. Therefore, if we forced the reclaim component to run once at the end of each program it would increase the runtime overhead by approximately 2%.

Memory Overhead. SdShield requires additional space to store relevant information when applying for memory. For each memory allocation, the user-space interceptor allocates an object that is 8 bytes larger than the original request, which is used to store a random value for overflow detection, the kernel module needs at least three page table entries (typi-cally 24 bytes) to store information, including the canonical virtual address, the ran-dom value at the end of the object, and the physical page address that is actually mapped. Table 2 shows the relationship between the memory usage of each space and the number of heap objects in the process when SdShield does not use the release component. However, the system's memory overhead hardly reaches this upper val-ue in practice, because most objects will be freed after use, and the information stored by SdShield will also be cleaned up when the release component runs.

Table 2. Memory usage of SdShield without releasing components

Heap Objects	User Space	Kernel Space	Total
1	8 B	4120 B	4128 B
1000	8 KB	27 KB	35 KB
100,000	781 KB	2361 KB	3142 KB
10,000,000	76 MB	239 MB	315 MB

Because SdShield's shadow page table is created in the kernel space by the kernel module, we cannot represent all of SdShield's memory overhead by just the application's memory usage. Instead, we calculate the overall memory overhead by the sum of application's resident set size (RSS) and the maximum number of page tables in the application's shadow range. Figure 8 compares the maximum memory overhead of SdShield with Oscar and Dangzero on the binary files of SPEC CPU2006. Similar to the runtime statistics, all three generate multiple overheads in those highly allocation-intensive programs. Although SdShield needs to store additional information for heap overflow detection compared to Oscar and Dangzero, due to the unique release feature, SdShield consumes less system memory than Oscar and is close to Dangzero, which uses a compression method. For the complete SPEC CPU2006 benchmark suite, the average SdShield memory overhead is 59%.

5.3 Comparison to Other Systems

The previous article compared the performance overhead of SdShield with Oscar and Dangzero, but due to the different actual effects of the schemes, it is limited to evaluate the quality of the schemes based solely on performance. Therefore, we next conducted a comprehensive comparison of the time overhead, memory overhead, and defense capabilities of each scheme, as shown in Table 3. Among this table, "√" means that such vulnerabilities can be detected, "×" means that they cannot be detected, and we indicate the average running overhead impact of the tool on the program by percentage.

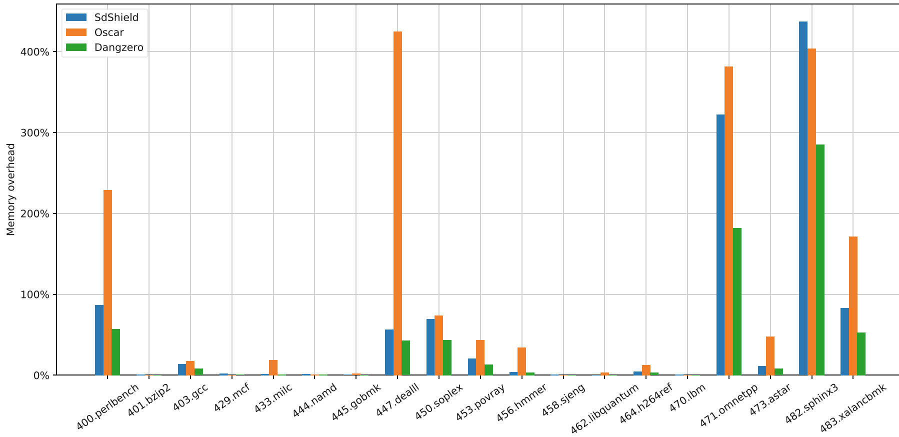


Fig. 8. SPEC CPU2006 memory overhead for SdShield

Table 3. General comparison between SdShield and Oscar and Dangzero

Feature	SdShield	Oscar	Dangzero
Runtime Overhead	64%	69%	20%
Memory Overhead	59%	98%	37%
Overflow	√	×	×
Use-After-Free	√	√	√
Double Free	√	√	√
Invalid Free	√	×	×
Over-read	√	×	×

Overall, on the complete CPU2006 benchmark suite, Oscar reported a geometric mean runtime overhead of 69% and memory overhead of 98%, Dangzero reported a geometric mean runtime and memory overhead of 20% and 37%, respectively, while SdShield's values were 64% and 59%. However, Dangzero requires running in a specific Kernel Mode Linux environment, and Oscar requires modifying the kernel source code. The generally shorter runtimes reported in Dangzero also seem to be a result of the Kernel Mode Linux environment. In contrast, although the overhead of SdShield is similar to the other two, SdShield demonstrates stronger portability and defense capabilities—it has no running environment restrictions, does not require compiling the kernel, and can defend against other types of heap vulnerabilities including heap overflows rather than just use-after-free, which proves the effectiveness of our design.

6 Limitations and Future Work

SdShield, as a solution that uses shadow virtual address, has similar limitations to other designs with the same principle [12, 14, 15]. All such designs currently target the system's default memory allocator, but applications may also rely on custom allocators [27]. In principle, our scheme can be applied to all applications by implement a separate interceptor for all custom allocators. Second, the current implementation of SdShield is not yet thread-safe, as competition between threads may lead to a confusing mapping of shadow pages. Future work could address the issue of thread competition through page table locking or by dividing a separate shadow area for each thread.

Besides that, while SdShield can prevent almost all use-after-free errors and report them as soon as it happens, for heap overflow we can only detect them when the program does an allocation or release operation. Although we can always check the canary of all objects in its page table when an object is free, we may not be able to detect it immediately when an overflow occurs. Future work could investigate checking the relevant canary when the process performs certain specific operations, such as dangerous system calls or memory write operations, rather than just memory allocation and release, as a way to improve the timeliness of heap overflow detection.

7 Conclusion

Using a page protection mechanism for process security is an old idea that was originally used for debugging. Although researchers have made optimizations based on shadow pages and solved the critical scalability costs in recent years, the state-of-the-art solutions still only support the detection of use-after-free errors and face portability issues. In this paper, we propose a new scheme based on shadow pages and page table permissions. We design a novel page table structure to secure the overall security of the heap area and not just for use-after-free only. We use a method that dynamically adds system calls to access and modify the process page table, enabling it to run on any Linux operating system without requiring a specific environment. Finally, our experimental evaluation demonstrates that our design significantly improves the security of the current page permission-based solutions.

References

1. Akritidis, P.: Cling: a memory allocator to mitigate dangling pointers. In: USENIX Security (2010)
2. Novark, G., Berger, E.D.: DieHarder: securing the heap. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 573–584 (2010)
3. Silvestro, S., Liu, H., Crosser, C., Lin, Z., Liu, T.: FreeGuard: a faster secure heap allocator. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2389–2403 (2017)
4. Liu, B., Olivier, P., Ravindran, B.: SlimGuard: a secure and memory-efficient heap allocator. In: Proceedings of the 20th International Middleware Conference (2019)
5. Ainsworth, S., Jones, T.M.: MarkUs: drop-in use-after-free prevention for low-level languages. In: IEEE Symposium on Security and Privacy, pp. 578–591 (2020)

6. Nagarakatte, S., Zhao, J., Martin, M.M.K., Zdancewic, S.: CETS: compiler enforced temporal safety for C. In: ISMM (2010)
7. Nikiforakis, N., Piessens, F., Joosen, W.: HeapSentry: kernel-assisted protection against heap overflows. In: Rieck, K., Stewin, P., Seifert, J.-P. (eds.) DIMVA 2013. LNCS, vol. 7967, pp. 177–196. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39235-1_11
8. Lee, B., et al.: Preventing use-after-free with dangling pointers nullification. In: NDSS, pp. 1–15 (2015)
9. Kouwe, E.V.D., Nigade, V., Giuffrida, C.: DangSan: scalable use-after-free detection. In: EuroSys, pp. 405–419 (2017)
10. Erdős, M., Ainsworth, S., Jones ,T.M.: MineSweeper: a clean sweep for drop-in use-after-free prevention. In: ASPLOS, pp. 212–225 (2022)
11. He, L., Hu, H., Su, P., Cai, Y., Liang, Z.: FREEWILL: automatically diagnosing use-after-free bugs via reference miscounting detection on binaries. In: 31st USENIX Security Symposium (USENIX Security 2022), pp. 2497–2512 (2022)
12. Dhurjati, D., Adve, V.: Efficiently detecting all dangling pointer uses in production servers. In: DSN (2006)
13. Younan, Y.: FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In: NDSS, pp. 1–15 (2015)
14. Dang, T.H.Y., Maniatis, P., David Wagner, D.: Oscar: a practical page-permissions-based scheme for thwarting dangling pointers. In: USENIX Security, pp. 1–18 (2017)
15. Gorter, F., Koning, K., Bos, H., Giuffrida, C.: DangZero: efficient use-after-free detection via direct page table access. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 1–15 (2022)
16. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: eternal war in memory. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy (2013)
17. Silvestro, S., Liu, H., Liu, T., Lin, Z., Liu, T.: Guarder: a tunable secure allocator. In: 27th USENIX Security Symposium, pp. 117–133 (2018)
18. Yun, I., Song, S.W., Min, S., Kim, T.: HardsHeap: a universal and extensible framework for evaluating secure allocators. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (2021)
19. Wickman, B., et al.: Preventing use-after-free attacks with fast forward allocation. In: USENIX Security (2021)
20. Robertson, W., Kruegel, C., Mutz, D., Valeur, F.: Run-time detection of heap-based overflows. In: Proceedings of the 17th USENIX Conference on System Administration (2003)
21. Younan, Y., Joosen, W., Piessens, F., Eynden, H.V.D.: Security of memory allocators for C and C++. Technical report (2005)
22. Shin, J., Kwon, D., Seo, J., Cho, Y., Paek, Y.: CRCount: pointer invalidation with reference counting to mitigate use-after-free in legacy C/C++. In: NDSS (2019)
23. Perens, B.: Electric fence malloc debugger. http://perens.com/FreeSoftware/ElectricFence/. Accessed 10 Apr 2023
24. Boland, F., Black, P.: The juliet 1.1 C/C++ and Java test suite. IEEE Comput. **45**(10), 88–90 (2012)
25. Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P.: BugBench: benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools (2005)
26. Leitch, J.: Issue 24613. array.fromstring use after free. https://bugs.python.org/issue24613. Accessed 10 Apr 2023
27. Berger, E.D., Zorn, B.G., McKinley, K.S.: Reconsidering custom memory allocation. In: OOPSLA (2002)
28. Bernhard, L., Rodler, M., Holz, T., Davi, L.: xTag: mitigating use-after-free vulnerabilities via software-based pointer tagging on Intel x86-64. In: IEEE EuroS&P (2022)

29. Burow, N., McKee, D., Carr, S.A., Payer, M.: CUP: comprehensive user-space protection for C/C++. In: AsiaCCS, pp. 381–392 (2018)
30. Farkhani, R.M., Ahmadi, M., Lu, L.: PTAuth: temporal memory safety via robust points-to authentication. In: USENIX Security (2018)
31. Gui, B., Song, W., Huang, J.: UAFSan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. In: ISSTA (2021)
32. Microsoft: GFlags and PageHeap. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap. Accessed 10 Mar 2023
33. Tian, D., Li, X., Chen, M., Hu, C.: ICruiser: an improved approach for concurrent heap buffer overflow monitoring. IEICE Trans. Inf. Syst. **97**(3), 601–605 (2014)
34. Maeda, T., Yonezawa, A.: Kernel Mode Linux: toward an operating system protected by a type theory. In: Saraswat, V.A. (ed.) ASIAN 2003. LNCS, vol. 2896, pp. 3–17. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40965-6_2
35. Boehm, H.J., Demers, A.J., Shenker, S.: Mostly parallel garbage collection. In: PLDI (1991)