MPKAlloc: Efficient Heap Meta-data Integrity Through Hardware Memory Protection Keys

William Blair[1(✉)], William Robertson[2], and Manuel Egele[1]

[1] Boston University, 111 Cummington Mall, Boston, MA 02215, USA
{wdblair,megele}@bu.edu
[2] Northeastern University, 360 Huntington Ave, Boston, MA 02115, USA
wkr@ccs.neu.edu

Abstract. Memory corruption exploits continue to plague high profile applications such as web browsers, high performance servers, and mobile devices. Modern defenses for these targets have rendered classic attack vectors that execute shellcode directly on the stack impotent and obsolete. Instead, modern exploits frequently corrupt the data structures found in a program's memory allocator in order to take control of running processes. These attacks against the heap are much harder to defend against versus classic stack-based buffer overflows because they often rely on an allocator acting on corrupted data in order to take control of a process. In this work, we introduce MPKAlloc, a memory allocator that utilizes memory protection keys (MPKs) found in recent Intel CPUs to effectively isolate heap meta-data from adversaries. We present our prototype implementation of MPKAlloc which hardens the `tcmalloc` and `PartitionAlloc` memory allocators used by the popular Chrome web browser. MPKAlloc protects each page containing heap meta-data with a key that provides an allocator exclusive access to the page. Effectively, MPKAlloc thwarts an adversary's ability to access or corrupt heap meta-data at the hardware level. We embed the MPKAlloc defense in the open-source Chromium web browser, and demonstrate MPKAlloc stopping realistic attack vectors. Furthermore, we evaluate the performance overhead of Chromium configured with MPKAlloc on the top 50 web sites contained in the Alexa site ranking. Our evaluation shows that MPKAlloc introduces a geometric mean of 1.71% performance overhead (2.44% on average) when browsing the most popular web sites, in exchange for a significant increase in security against heap meta-data exploitation.

Keywords: Memory protection keys · Hardened memory allocators · Hardware security

1 Introduction

The turn of the 21st century saw the explosive growth of the World Wide Web and with it the rapid adoption of web browsers as a means to consum-

L. Cavallaro et al. (Eds.): DIMVA 2022, LNCS 13358, pp. 136–155, 2022.
https://doi.org/10.1007/978-3-031-09484-2_8

ing entertainment, conducting business, and social networking. As the world flocked to the Web and began rapidly creating and sharing content within web browsers, the problem of securing individual computers from malicious content quickly emerged. In 2001, the classic unlink exploit became known on the popular Netscape browser using a seemingly harmless JPEG image [33]. This exploit only required a user to visit a web site hosting the hostile image in order to allow an adversary to hijack the browser process. The exploit itself took advantage of how Netscape's memory allocator failed to sufficiently validate pointers held in a memory chunk's meta-data. If an adversary could corrupt this meta-data, they could fool the allocator into writing into an arbitrary address once the allocator freed the corrupted chunk. In this case, a malicious JPEG file could trick the image processing module in the browser to corrupt a chunk of memory, free it later on, and take control of the Netscape process. In the decades since the disclosure of this exploit, heap exploitation has grown into a sophisticated craft that continues to evade the defenses found within modern memory allocators [6]. Memory allocators protect the integrity of meta-data with cookies [31], use advanced hardware features to check the integrity of pointers throughout program execution [18], or explicitly track pointers to prevent the exploitation of temporal memory errors [8]. Such defenses may come with non-trivial performance overhead in both execution time and memory usage. For example, the recent MarkUS allocator [8] can efficiently track dangling pointers throughout program execution. However, it does so at the cost of a worst-case 2X performance penalty and one-third memory overhead. Making a memory allocator more secure by adding additional checks to detect tampering with internal data structures creates tension with the allocator's intended purpose: efficiently providing memory chunks of arbitrary size to a program whilst minimizing fragmentation. Furthermore, recent work has begun automating the discovery of novel heap exploitation techniques [20,35,37] in the spirit of automated exploit generation (AEG) [10]. This automation complicates the task of reliably ensuring heap integrity. For these reasons, any proposed security improvement to a production-quality allocator requires significant evidence that the change will not negatively impact performance in the allocator's intended use cases.

In light of this tension, we propose MPKAlloc, a technique for hardening memory allocators that isolates heap meta-data by leveraging hardware memory protection keys (MPKs) available on recent CPU architectures. Unlike software-based integrity checks, MPKs immediately detect any attempt to read from or tamper with heap meta-data. In addition, MPKs incur no significant performance overhead and no memory footprint. Memory protection keys can also be found in ARM processors in the upcoming memory tagging extensions (MTE) [5] as well as IBM's AIX operating system as "storage protect keys" [3] for Power systems [21]. MPKs divide the pages that make up a process' address space into individual protection domains and allow each thread of execution to operate within one or more domains. MPKAlloc uses this functionality to confine a memory allocator to operate within its own (privileged) domain when reading from and writing to heap meta-data. MPKAlloc assigns regular (i.e., non

meta-data) heap memory chunks a label with another (unprivileged) domain. Later on, if an adversary exploits a bug in the running program, any attempt to read or corrupt heap meta-data is prevented by the hardware and, by default, the OS terminates the offending process. This simple defense effectively neutralizes the entire class of heap meta-data corruption attacks that evolved from the classic unlink attack. Isolating heap meta-data from program components using memory protection keys can be accomplished with a compact implementation given sufficient domain knowledge of the memory allocators used by an application. We present a prototype implementation of MPKAlloc based on the popular `tcmalloc` and `PartitionAlloc` memory allocators and Intel MPK. We evaluate MPKAlloc on the SPEC CPU2006 benchmarks and show that MPKs introduce only marginal overheads. To evaluate MPKAlloc's defensive capabilities on real software, we embedded MPKAlloc within multiple allocators in the Chromium web-browser. Browsers based off of Chromium enjoy 63.59% of the worldwide browser market share as of 2021. We found that in a realistic scenario, where an adversary achieves an arbitrary write primitive by corrupting meta-data, MPKAlloc intercepts the adversary before the corruption and terminates the compromised Chromium process in response. We measure the performance impact of MPKAlloc when visiting the top 50 most popular web sites contained in the Alexa ranking. Our results show that MPKAlloc can protect widely used programs like the Chromium web browser from real attacks while introducing little performance overhead on average. Prior systems have focused on developing secure intra-process isolation schemes within specialized high-performance servers [19,34] or making MPKs a virtualized resource [28]. In contrast, MPKAlloc uses MPKs to ensure the security of sensitive data within widespread consumer software. In the interest of open science, the source code for MPKAlloc can be found online[1].

In summary, we make the following contributions in this paper.

- We recognize that heap meta-data and regular data allocated in the heap can be completely partitioned.
- We introduce MPKAlloc, a generic technique for hardening memory allocators that leverages memory protection keys to enforce these partitions. This implements intra-process code partitioning and provides an allocator exclusive access to its meta-data (see Sect. 3.2).
- We design and implement a prototype that applies the MPKAlloc technique by hardening Google's popular `tcmalloc` and `PartitionAlloc` allocators using Intel's memory protection keys.
- Our security evaluation shows that MPKAlloc prevents exploits that corrupt heap meta-data in the Chromium web browser.
- Finally, our performance evaluation demonstrates that MPKAlloc introduces little performance overhead as measured over the SPEC CPU2006 benchmarks (0.8% on average) and on page load times in Chromium (2.44% on average).

[1] https://github.com/BUseclab/mpkalloc.

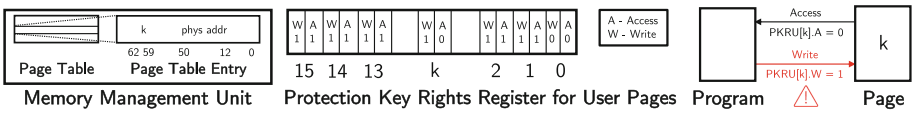


Fig. 1. Memory protection keys on intel CPUs.

2 Background and Threat Model

In this section, we provide a brief overview of memory protection keys, works related to MPKAlloc, and the threat model we assume in this work.

2.1 Memory Protection Keys

Recent Intel CPUs feature memory protection keys (MPKs) for partitioning memory pages into multiple intra-process domains. Figure 1 visualizes MPKs protecting a page associated with a key k. The memory management unit (MMU) within these CPUs maintains a page table which represents a mapping of virtual pages held by userspace processes to the physical pages located in hardware. Each page table entry (PTE) maps a single virtual page to a corresponding physical page and holds additional information, such as whether or not the page is dirty. On Intel, the PTE also includes an additional 4-bit value located at bits 59–62 that denotes the page's protection key. Intuitively, a page marked with key k can only be accessed or written to by properly configured threads. Since altering the page table requires executing privileged instructions on x86, the operating system kernel must provide a system call for programs to assign protection keys to individual pages. Note that, MPK currently only supports associating an individual page with a single key from the 16 available keys. By default, all pages are assigned the protection key $k = 0$.

During program execution, each CPU thread maintains its own protection key rights register for user pages (`PKRU`) that restricts the thread's access to individual protection key domains. Upon every memory load or store instruction, the MMU compares the protection key given in the accessed PTE to the permissions given in the current thread's `PKRU` register. If the `PKRU` register permits the operation, then the CPU carries out the instruction. Otherwise, the hardware will raise an error by notifying the operating system that a segmentation violation has occurred. An operation on a page with key k is permitted if the operation's bit is 0 at the relevant position in k's entry within the `PKRU` register. For the `PKRU` register given in Fig. 1, the access to the page with key k succeeds, while the write fails. An operator may therefore disallow either writes or all access to pages associated with k by setting the appropriate bit in the `PKRU` register. The CPU provides instructions available to userspace programs to fetch and alter an individual thread's `PKRU` register. By default, the `PKRU` register begins with a value of 0 which effectively disables MPKs. Note that, programs may not restrict access to $k = 0$ on current hardware. More details on the system calls and instructions required to use MPKs can be found in Sect. 4.

2.2 Related Work

MPKAlloc relates to prior work in two categories: Hardened Memory Allocators and Process Isolation Schemes.

Hardened Memory Allocators. Memory allocators often set aside a region of memory called the heap to store chunks needed by a program. Allocators must maintain data structures (heap meta-data) so that a program can efficiently obtain memory (heap chunks) of arbitrary size on demand. Isolating heap meta-data from adversaries is an important security goal since meta-data corruption can violate the integrity or confidentiality of a program. Numerous changes have been made over the last two decades to balance memory allocator performance requirements with the need to secure their data structures from corruption. As an example, the classic unlink [9,33] attack allows an adversary to write arbitrary data into any location in memory by overwriting pointers that refer to the next and previous chunks in the allocator's doubly-linked list. When the allocator attempts to free, or unlink, the corrupted chunk, the allocator instead writes an attacker-controlled value at an arbitrary address specified by the attacker which can have serious security consequences [11,32]. While the classic unlink attack has been mitigated for some time, this simple attack has spawned numerous variants that exploit binary programs through corrupting heap data structures [6]. For this reason, any defense that ensures the integrity of heap data structures without adversely affecting the performance of an allocator can provide significant security value, as it renders the entire class of heap meta-data corruption attacks moot [31]. MPKAlloc uses MPKs to isolate meta-data without requiring instrumentation or extensive changes to the allocator. This improves the allocator's security posture and enables easier adoption. That is, applications are typically implemented against a generic allocator interface, and so could easily be configured to use MPKAlloc instead of a default allocator. In some cases, MPKs can be easily added to an allocators like the big bag of pages (BiBoP) allocators found in some BSD derived operating systems [27] which store meta-data and allocated chunks on separate pages. Contrast this approach to one that requires explicitly changing the layout of memory within an allocator to accommodate MPKs [24]. Recent allocator fuzzing techniques like the `Uninitialized` module found in the HardsHeap [36] framework permit efficiently testing for the co-location of meta-data and allocated chunks without requiring extensive manual analysis. Furthermore, more offensive analysis could automatically discover novel exploits against allocators that fail to properly separate meta-data from allocated chunks [35]. In order to harden a given memory allocator using MPKs, developers could combine these two approaches in order to detect critical security flaws during development. This task may be difficult for glibc's `ptmalloc` [2], where meta-data immediately precedes allocated chunks in memory, but simpler for Firefox's `jemalloc` [1], where meta-data and allocated chunks are stored separately but still share pages in memory.

Process Isolation. Memory can also be partitioned using secure memory views (SMV) which permit operating system threads and their children access to memory associated with individual domains. Recent systems like ERIM [34] and Hodor [19] employ MPKs to isolate both individual workloads [34] and shared libraries [25] within the same process. More restrictive sandboxes can also be defined by embedding instruction monitoring directly into the hardware [16]. These works demonstrate MPKs preventing a compromised thread running within one protection domain from executing code in another domain. This builds upon widely used defenses such as data execution prevention (DEP) and $W \oplus X$. In contrast to these approaches, code domain partitioning uses MPKs to ensure the security of a library's sensitive memory pages, as opposed to developing general purpose intra-process sandboxes. Therefore, our defense complements prior work on intra-process sandboxes. The interface that allows programs to obtain and use MPKs can easily cause security problems and improvements have been proposed to make MPKs a proper virtualized resource [28]. NOJITSU [29] uses MPKs to ensure a Just in Time (JIT) compiler in web browsers has exclusive access to internal data structures and employs dynamic and static analysis that limits the performance overhead introduced by MPKs. In contrast, MPKAlloc establishes a memory allocator as a trusted component for managing sensitive meta-data. In both NOJITSU and MPKAlloc, MPKs enforce a security policy that provides a principal exclusive access to private resources. Prior works have also demonstrated the benefit of using hardware protection mechanisms to protect allocator meta-data [12,17,23]. However, these works primarily demonstrate the performance benefits of using hardware features. They do not investigate the security benefits of using MPKs beyond preventing simple heap overflows. In this work, we introduce an indirect meta-data attack vector (see Sect. 3.1) and show how MPKs prevent such attacks in a commodity web browser.

2.3 Design Assumptions and Threat Model

In this work, we assume that the protected program utilizes a memory allocator to dynamically allocate memory chunks. Meta-data documents the structure of the heap and allows an allocator to quickly meet the running program's memory demands. A memory chunk is a contiguous span of memory held by an allocator. In this work, we consider an allocator as a trusted component that stores heap meta-data and chunks on separate memory pages. Requiring meta-data and chunks to reside on separate pages is an inherent requirement when using existing Intel MPK technologies. If the program were to access memory chunks lying on the allocator's pages, MPKAlloc would trigger a segmentation fault upon every access. This implies that allocators that co-locate meta-data with allocated chunks, such as `ptmalloc` and `jemalloc`, cannot protect meta-data using Intel MPKs by default. The high performance allocators found within the Google Chromium web browser store meta-data and allocated chunks on separate pages, and therefore provide natural targets for prototyping MPKAlloc (see Sect. 4).

In this work, we assume the following threat model.

- The adversary's primary goal is to corrupt heap meta-data in order to achieve a malicious goal. One such goal would be to obtain a write primitive in a victim program. With the ability to write data anywhere in the process, an adversary can achieve more powerful capabilities such as escalating privileges or executing arbitrary code. We stress this represents just one example of what an adversary may achieve by corrupting meta-data.
- To this end, the adversary can inspect the protected program and all of its library dependencies.
- The adversary can further influence the input to the program (e.g., standard input, a remote socket, a file read by the program, or through specific command line arguments).
- Finally, the adversary has access to a vulnerability in the program or any of its libraries (with the exception of the memory allocator itself) that allow him to access the heap's meta-data and achieve his primary goal.

In this work, we assume an adversary does not begin with the ability to execute arbitrary code either explicitly or through code reuse attacks. Such an adversary has no motivation to corrupt heap meta-data since he can trivially take over a process or disclose any information reachable from the process. For this reason, we also assume the protected program does not utilize control flow integrity (CFI) [7] since the adversary specified by our threat model has no ability to alter the program's flow of execution or alter threads' protection domains. This is just an assumption made for our threat model, and in a deployed setting CFI could easily run alongside MPKAlloc to further protect a workload. The principal goal of MPKAlloc is to prevent adversaries from breaking the security of a running process by corrupting internal memory allocator data structures, not to design a general purpose intra-process isolation scheme that is resilient to compromised threads. At the time of writing, a secure intra-process isolation scheme using MPKs is still an active research topic [15]. As a concrete step under this broad objective, MPKAlloc uses memory protection keys to prevent adversaries from corrupting internal memory allocator data structures. At the same time, we acknowledge that MPKAlloc does not prevent issues caused directly by memory allocators, such as double-free corruptions.

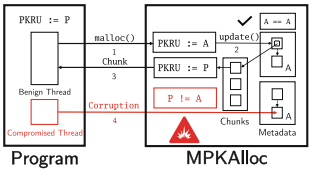


Fig. 2. Architectural overview of MPKAlloc.

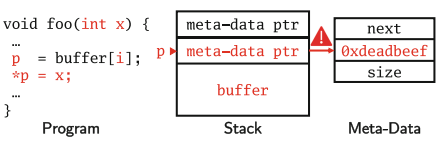


Fig. 3. MPKAlloc blocking a corruption performed by writing to a meta-data pointer obtained from an out-of-bounds read from the stack.

3 System Overview

In this section, we provide a high level overview of MPKAlloc. We show how MPKAlloc successfully detects and thwarts an adversarial attempt to transform a simple heap corruption into a powerful exploit. An adversary could easily use corrupted meta-data to place a ROP chain somewhere in the process, cause the program to jump to the chain, and connect to a remote command and control (C&C) server and install malware. We describe code domain partitioning, a technique that isolates memory allocator meta-data from the rest of the program, as a mechanism for isolating heap meta-data. Finally, we show that memory protection keys available in recent Intel CPUs provide an efficient implementation of code domain partitioning. Figure 2 shows the architectural overview of MPKAlloc and how it protects a memory allocator's meta-data from a vulnerable program. The key invariant MPKAlloc preserves is that any access made to a meta-data page must come from a privileged domain. By default, all threads in a running program run in an unprivileged domain. Upon invoking the memory allocator, MPKAlloc switches the thread to the privileged domain so that the allocator may access its internal meta-data. Note that this happens completely transparently to the running program. This allows MPKAlloc to be deployed by exclusively modifying the program's memory allocator. Later on, when a malicious or compromised thread attempts to access this internal data from an unprivileged domain, the hardware observes that the thread's protection key register disallows access to the domain's data. Upon receiving an exception from the hardware, the OS responds by terminating the program before any information is disclosed or corrupted. This prevents the malicious thread from successfully exploiting the heap. This is the default response, and, depending on the use-case, less drastic measures may be taken, such as alerting an administrator or raising a visual warning within the browser to alert a user of a possible exploit.

3.1 Indirect Meta-data Corruption

In allocators like `tcmalloc` and `PartitionAlloc`, chunk meta-data resides on separate pages which makes it difficult to successfully achieve the classic unlink attack given in Sect. 2. Fortunately, browsers like Chromium also place guard pages beneath meta-data pages which makes it impossible to corrupt pages via a direct overflow. Figure 3 shows an alternative attack vector where an adversary indirectly corrupts meta-data by writing to a pointer stored on the stack. At the beginning of the function `foo`, the adversary can influence the value of `x` and the index `i` used to read an element from `buffer`. In this attack, an adversary uses this capability in order to read past the boundary of `buffer` and obtain a pointer `p` that refers to meta-data. When the program writes to `p`, an adversary can then influence the contents of meta-data. In this case, if the target meta-data page contains a free list entry, then an adversary can replace a `prev` pointer with an address of their choice. Later, when the allocator frees the corrupted chunk, the allocator will inadvertently write to an attacker controlled address (`0xdeadbeef`). Targeting allocator meta-data on the stack has the advantage of corrupting a

heavily used data-structure without requiring intricate knowledge of a victim application. That is, if an adversary is familiar with the memory allocator used by a victim application, and the application heavily relies on heap memory, then the adversary could attempt to spray attacker controlled values using meta-data pointers located on the stack. MPKAlloc prevents such an attack from achieving an arbitrary write primitive by detecting any successful meta-data corruption. Note that this is a two-stage attack, an adversary must first obtain a buffer over read to obtain a meta-data pointer, and then successfully dereference the obtained pointer. We also assume that no attacker controlled values are reachable on the stack, since an adversary could trivially obtain a write primitive by obtaining those values as opposed to meta-data. In the past, remote attackers have been able to achieve the first stage of our attack by exploiting an out-of-bounds read on the stack within `libevent`, a library used by the Chromium web browser [4]. In Sect. 5, we demonstrate our hypothetical attack vector in Chromium.

3.2 Code Domain Partitioning

Code domain partitioning allows a developer to grant regions of a program's code segment exclusive access to memory pages associated with a specific domain. MPKAlloc utilizes two domains and divides the heap into the allocator domain A and the program domain P. Pages in A belong exclusively to the memory allocator and pages in P belong to the program. To ensure the memory allocator has exclusive access to A, we alter the memory allocator to assign `PKRU` the appropriate domain A before entering every allocator routine. When the allocator obtains new pages for meta-data from the operating system, the allocator associates each page with the domain A using a variant of the `mprotect` system call that assigns pages to protection key domains. We do not need to explicitly assign other pages to the domain P since by default every page is assigned the MPK domain 0 by the hardware. In this case, any thread in the process is allowed to access pages in domain P subject to the "classic" read, write, and execute permission bits. Upon exiting any allocator routine, we remove the domain A from the `PKRU` register so that any attempt to alter a page in A yields a segmentation violation.

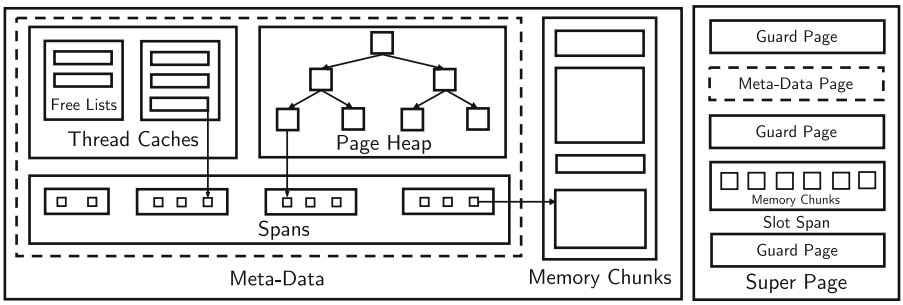
3.3 Detecting Domain Violations

Code domain partitioning delegates the task of detecting accesses made across domain boundaries to the CPU. Once an allocator's internal source has been altered to switch domains at the appropriate points, the hardware will detect any unprivileged accesses or writes made outside the allocator. For example, an adversary may attempt to corrupt a meta-data page by indirectly writing to a meta-data pointer as shown in Fig. 3. At this point, an adversary can take advantage of an indirect write to corrupt heap meta-data (e.g., overwrite pointers contained therein) in an attempt to exploit the program. The code causing the corruption is outside the memory allocator itself and the protection domain

of the current thread belongs to P which differs from the domain A assigned to the victim meta-data page. As soon as the adversary in domain P issues a store instruction into a meta-data page associated with domain A, the hardware observes that the thread P cannot access the meta-data in domain A. This causes the hardware to notify the OS of the violation, and the OS terminates the process in response. This is the default response after detecting an MPK violation, and therefore we utilize it in MPKAlloc. This simple example shows how assigning meta-data pages their own protection domain A can prevent adversaries from corrupting heap meta-data. Section 4 provides an overview of the various data structures that `tcmalloc` and `PartitionAlloc` use as meta-data in order to implement a performant heap. Furthermore, the trend of automated heap exploitation suggests that anticipating all possible attack patterns facing a given memory allocator may become impractical. Storing meta-data within its own protection domain A successfully stops any attempt to access or alter meta-data. MPKAlloc thus defangs all attacks that rely on corrupting heap meta-data.

4 Implementation

In this section we describe our prototype implementation of MPKAlloc on the Intel x86-64 architecture. This allows us to apply code domain partitioning using the memory protection keys (MPKs) available on recent Intel CPUs. We choose to embed the MPKAlloc technique within the `tcmalloc` and `PartitionAlloc` allocators in our prototype. This implementation consists of 160 SLOC. This includes the implementation embedded within both `tcmalloc` contained in gperftools-v2.7 and `PartitionAlloc` included in Chrome version 84.0.4108.0. While the amount of code needed to implement MPKAlloc is small, understanding the structure and layout of two different high performance memory allocators and carefully adjusting their design to protect meta-data requires both significant domain knowledge and careful experimentation. The latest Intel CPUs provide a thread-specific register, `PKRU`, that defines each thread of execution's permissions for each memory protection domain. MPKAlloc uses the new instructions that alter this register to implement a security policy for an allocator's meta-data; accesses made by the program are forbidden, but those made by the allocator are allowed. We use functionality provided by the GNU/Linux kernel and glibc to label every page allocated for meta-data the domain A. Every time control transitions the boundary between the allocator and the rest of the program, MPKAlloc must switch between the two domains. The boundary between the domain of the program P and the domain of the allocator A consists of the Standard C Library functions that allocate and deallocate memory (e.g., `malloc`, `realloc`, `free`, etc.), and the C++ operators that allow programs to create and destroy objects (e.g., `new` and `delete`). In our implementation of MPKAlloc, we initialize the `PKRU` register with a specific domain assigned to the program P. As the pages backing the allocator's meta-data belong to domain A, any access made by P to these pages will generate a segmentation violation



(a) `tcmalloc` (b) `PartitionAlloc`

Fig. 4. Memory allocators protected by MPKAlloc.

before any meta-data can be disclosed or corrupted. Since both `tcmalloc` and `PartitionAlloc` store meta-data and allocated chunks on separate pages, our defense allows a program to read, write, or even execute allocated memory in P without requiring any modifications or instrumentation to the program itself.

4.1 Meta-data in `tcmalloc` and `PartitionAlloc`

The `tcmalloc` and `PartitionAlloc` allocators are high performance memory allocators found within the Chromium web browser. In `tcmalloc`, every allocated chunk is backed by one or more span objects that may extend across multiple pages in virtual memory. A single span is simply a collection of one or more pages in virtual memory. Figure 4a shows the layout of the various data structures that define `tcmalloc`'s meta-data. Within `tcmalloc`'s implementation, a special MetaDataAllocator class is responsible for allocating the pages that hold these internal data structures such as thread caches, free lists, a central page heap, and individual spans. Thread caches provide an efficient way for threads to allocate memory using free lists without having to consult the larger page heap that is shared across all threads. The page heap is a trie data structure that allows `tcmalloc` to quickly obtain the span object associated with a given pointer. In `PartitionAlloc`, the unit of allocation is a super page that records the set of memory chunks that reside across a series of allocated pages. Figure 4b visualizes the data structures that make up each super page. Each allocated page is referred to by an individual slot span located in the super page. At the top of the super page lies a meta-data page surrounded by guard pages that prevent linear overflows from corrupting the data structures that document the super page's contents. Upon allocation of every super page, MPKAlloc assigns the meta-data page with the domain A to ensure that the allocator retains exclusive access to its meta-data during run-time. In this design, we assign all the instructions found within the `tcmalloc` and `PartitionAlloc` allocators the domain A, and the rest of the process' code to the domain P.

4.2 Code Domain Partitioning

Both `tcmalloc` and `PartitionAlloc` rely on the `mmap` system call to obtain pages from the operating system upon which they build a heap. In `tcmalloc` we implement code domain partitioning by altering the MetaDataAllocator class to tag every page allocated by `mmap` with the appropriate protection key and permissions. Before the program allocates any memory, we must allocate a protection key for the domain A within `tcmalloc`'s initialization routine. This involves simply calling the `pkey_alloc` function with our desired permissions (e.g., to disable access and write operations). Whenever a MetaDataAllocator obtains new pages, MPKAlloc calls `pkey_mprotect` on a pointer to the new pages with the allocated protection key. This assigns the pages to the protection domain A and prevents threads outside of A from accessing the pages. Likewise, every time `PartitionAlloc` allocates a super page, MPKAlloc associates the corresponding meta-data page with the domain A. This implements a form of code domain partitioning that divides the pages in the virtual address space into two domains, the domain A reserved for memory allocators, and the domain P for memory allocated in the heap. Furthermore, this approach allows us to initialize MPKAlloc's context without modifying the running program. Later on, if a thread does not hold the proper permission in the `PKRU` register while reading from or writing to one of the key's pages, the hardware will raise an exception and prevent the memory access from succeeding. The OS will then send a SIGSEGV signal to the offending process and, by default, terminate it. Since both allocators reside in Chromium's source tree, they can easily share A's protection key. Once we obtain a protection key from the operating system, we store the key in a static variable which makes the key available throughout both `tcmalloc` and `PartitionAlloc`. Under our threat model, disclosing the protection key does not help an adversary corrupt meta-data, but in a production deployment it would be wise to obscure the protection key's location by storing it outside the allocator's structures.

In order for an allocator to access internal meta-data pages, MPKAlloc must perform a domain switch whenever the program manipulates the heap using one of the allocator's public APIs. This is accomplished by simply writing the appropriate value to the `PKRU` register as specified by the Intel Architecture Software Developer Manual [22]. That is, MPKAlloc zeros the bits reserved for the allocated key in the `PKRU` register which permits full access to A's pages. The `wrpkru` instruction allows MPKAlloc to alter the contents of the `PKRU` register in order to perform this privilege escalation. When the memory allocator runs in domain A, the hardware will see A's entry in `PKRU` is zero, which permits full access to all pages associated with A. Once an allocation or deallocation routine returns, the allocator no longer needs to alter heap meta-data. At this point, MPKAlloc switches the thread's domain by setting the write and access bits at A's offset in the `PKRU` register, which causes any future attempt to read from or write to A's pages to fail. After the context switch completes, the function returns to the code that invoked the allocator. The high performance memory allocators given in our evaluation often do not have a clear boundary between

entering and leaving the memory allocator. This can lead to redundant writes to the `PKRU` register when performing a context switch. Writing to the `PKRU` register incurs more clock cycles than reading its contents. For this reason, we follow the example set by prior work [29] by checking the contents of the `PKRU` register before altering it whenever we enable or disable our defense. In our evaluation, we saw this straightforward optimization reduce the average performance overhead from 5.4% to 2.4% when loading the top 50 websites in the Alexa ranking (see Sect. 5.6).

4.3 Detecting Corruptions

Protecting meta-data from adversaries can be accomplished by switching a thread's protection domain to A whenever a program enters an allocator's public function. This ensures that whenever an allocator alters its internal data structures, the calling thread has the proper protection domain to pass the protection key check performed by the CPU on all pages holding heap meta-data. Furthermore, once a thread leaves an allocator's functions, MPKAlloc transitions to the protection domain reserved for the program P. Any attempt made by an adversary to access or alter the allocator's meta-data while the program is in this unprivileged domain will fail. Note that each thread has its own value for the `PKRU` register which obviates any synchronization between different threads and similarly forestalls race conditions that could occur while switching domains.

5 Evaluation

In this section, we evaluate both the performance overhead and security benefits provided by MPKAlloc. First, we embed MPKAlloc within the stock `tcmalloc` allocator, and measure the performance overhead induced by MPKAlloc over the SPEC CPU2006 benchmarks. We compare this overhead to the runtime we observe when using a stock `tcmalloc` allocator in the benchmarks (see Sect. 5.2). Next, we evaluate the feasibility of embedding MPKAlloc into two memory allocators found within the Chromium web browser (see Sect. 5.3). Once we confirmed that Chromium can benefit from MPKAlloc, we evaluate the feasibility of our PoC attack that indirectly corrupts meta-data. This involves writing directly through pointers that refer to meta-data. In Sect. 5.4 we confirm that meta-data pointers occur frequently in Chromium's process address space. We use this information to construct a PoC that corrupts meta-data by writing through such a pointer, and show how MPKAlloc prevents the corruption from occurring (see Sect. 5.5). Finally, we measure the performance impact a user may experience when using Chromium hardened with MPKAlloc (see Sect. 5.6).

5.1 Experimental Set Up

We evaluated MPKAlloc on an Ubuntu 20.04 LTS server with 96 Intel Xeon Platinum 8000 CPUs and 192 GB of RAM. The initial experiments for evaluating MPKAlloc over the SPEC CPU2006 benchmarks used `tcmalloc` available in

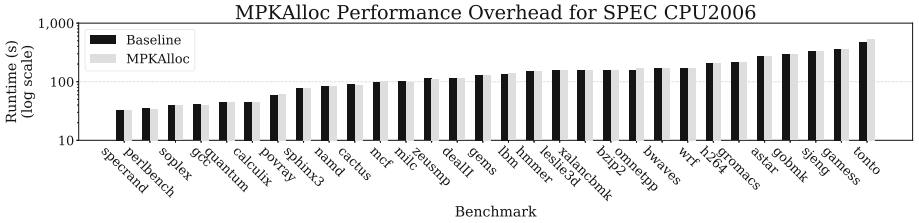


Fig. 5. Measuring MPKAlloc's performance overhead (+0.8% on average) using an unmodified `tcmalloc` as a baseline.

gperftools version 2.7. The browser experiments were conducted using `tcmalloc` and `PartitionAlloc` found in version 84.0.4108.0 of Chromium. We built and evaluated Chromium within a container running the Ubuntu 16.04 LTS distribution as recommended by the browser's documentation for developers.

5.2 SPEC CPU2006 Benchmarks

Figure 5 summarizes the performance overhead of running the SPEC CPU2006 benchmarks with MPKAlloc compared to using an unmodified `tcmalloc` allocator as a baseline. In this evaluation, each benchmark is ran three times with each allocator. The main source of performance overhead introduced by MPKAlloc comes from issuing the relatively expensive `wrpkru` instruction when switching between protection domains. On average, we observed low performance overhead across all the SPEC CPU2006 benchmarks (+0.8%). Based on this result, we argue that MPKAlloc could be applied to production programs without substantially affecting their performance.

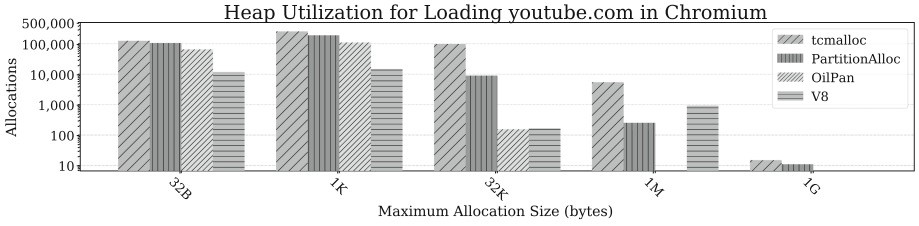


Fig. 6. Measuring each memory allocator's activity inside Chromium while loading youtube.com. Each allocation fits within a bin given on the x-axis.

5.3 Hardening Chromium with MPKAlloc

The positive result obtained from the SPEC CPU2006 benchmarks makes MPKAlloc a natural candidate to protect the Chromium web browser. Chromium is an open source browser upon which the Google Chrome browser

is based and heavily relies on the `tcmalloc` and `PartitionAlloc` allocators. The PartitionAlloc-Everywhere initiative tracks the eventual transition to using `PartitionAlloc` across the entire Chromium codebase [14]. Two garbage collectors, OilPan and V8, allocate memory for the Blink CSS renderer and Javascript interpreter, respectively. In this work, we restrict MPKAlloc to protect meta-data within `tcmalloc` and `PartitionAlloc`. We leave protecting garbage collectors' meta-data in Chromium to future work. Figure 6 summarizes the pressure placed on each memory allocator while Chromium renders youtube.com, a widely viewed media heavy web site. For every chunk allocated by a heap, we place the chunk in the smallest bin that will hold it. For example, the largest bin of size 1 GB holds all chunks that cannot fit in the bin of size 1 MB. Even though all allocators are utilized while rendering youtube.com, we observed that Chromium utilized `tcmalloc` and `PartitionAlloc` the most, measured by the total number of chunks allocated. To ensure that different web sites stress MPKAlloc, we counted the number of protection domain context switches performed by MPKAlloc while rendering the top 50 web sites contained in the Alexa ranking. Figure 7 visualizes the number of protection domain context switches observed and shows popular web sites utilize MPKAlloc by placing significant pressure on the protected allocators.

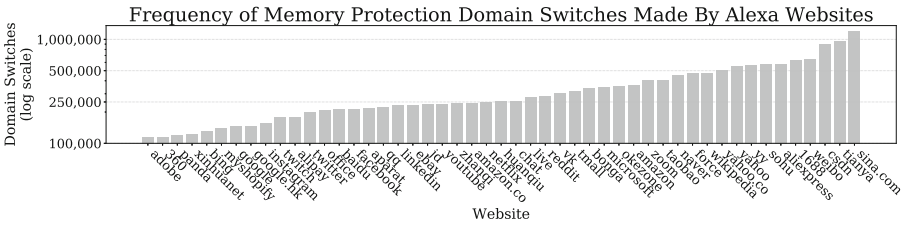


Fig. 7. Protection domain switches performed by MPKAlloc while loading Alexa websites.

5.4 Detecting Heap Meta-data in Chromium

Recall that the attack vector discussed in Sect. 3 achieves an arbitrary write primitive by writing to a meta-data pointer stored on the stack. While MPKAlloc is oblivious to an adversary's goals beyond compromising heap meta-data, an adversary can easily use a write primitive to corrupt arbitrary data and escalate privileges, disclose sensitive information, or execute arbitrary code. In order to understand the practicality of this attack vector within the Chromium web browser, we altered `tcmalloc` and `PartitionAlloc` to record every memory region allocated for heap meta-data while loading Google's home page. As the most popular website on Alexa's ranking, Google's home page can provide a good reference as to whether an adversary can reach heap meta-data from the stack through a buffer over read.

Detecting Meta-data Pointers. In order to determine whether such a scenario is possible, we implemented a meta-data memory scanner that attaches itself to each process in Chromium's process tree using the `ptrace` API. The scanner starts with the set of memory pages allocated as meta-data by Chromium. The scanner then examines every readable memory region mapped into the process. The scanner enumerates each region 8 bytes at a time in order to detect any pointers that refer to a meta-data page. This initial scan revealed that heap meta-data pointers frequently appear on the stack. In order to be useful to an adversary, the pointers themselves must be located at a memory address higher than a given function's stack frame so an adversary may reach the pointer through a buffer over read. The current stack can be obtained through the `RSP` register which represents the top of the stack in the x86-64 architecture. To measure how often heap meta-data appears at memory addresses higher than a given stack frame, we implemented a PINtool using the Intel PIN framework [26] that scans a fixed amount of memory located higher than `RSP` (256 bytes) upon every function return. This allows the PINtool to scan memory on the stack for pointers to meta-data pages throughout Chromium's execution. We limited our scan to 256 bytes in order to quickly see whether meta-data was located close to the stack pointer. In addition to trapping on every function return, the PINtool traps on every call to functions that allocate heap meta-data which identifies the location of meta-data pages in virtual memory. The previous scanner provided evidence that heap meta-data occurs at the high addresses reserved for the stack. This second scanner confirms all the opportunities an adversary may have to reach meta-data during execution. We observed that the browser appears to start with a fixed set of meta-data pages which are referred to throughout browsing. This suggests the meta-data pointers we observed remain valid throughout our scan. Our second scan which tracks meta-data in real-time revealed that simply loading `google.com` causes heap meta-data to be within reach in over 52,000 distinct methods in the Chromium source tree. This implies that if an adversary can write to a pointer stored somewhere on the stack in any one of these methods, they can successfully corrupt a heap meta-data pointer, and achieve an arbitrary write primitive. For example, functions in the `blink` module may be of interest since the Blink renderer parses cascading style sheet (CSS) files downloaded from the Internet. In this setting, adversaries may craft CSS files that exploit a bug in the renderer's implementation. This more detailed scan also revealed meta-data is reachable in modules that parse URLs, handle network traffic, and interact with the domain object model (DOM). All of these modules are viable targets for an attacker looking to exploit a web browser. One explanation for meta-data pointers' frequent appearance on the stack is Chromium methods reusing stack frames belonging to internal allocator routines that store meta-data pointers on the stack while accessing data structures. For example, suppose a routine that parses CSS files calls a routine to allocate a chunk of memory. After these functions return, a CSS rendering routine is called, and a meta-data pointer may remain on the stack in place of an uninitialized variable at an address higher than the stack frame of a method vulnerable to the hypothetical

attack described in Sect. 3.1. An adversary could take advantage of this layout by obtaining the meta-data pointer, corrupting meta-data, and achieving an arbitrary write primitive.

5.5 Corrupting Meta-data in Chromium

In order to demonstrate the security benefit of MPKAlloc we follow the approach taken by recent work that protects components in Mozilla's SpiderMonkey Javascript JIT compiler [29] using MPKs. Instead of developing a full exploit to test the defense, the authors introduced a bug into the code base and showed how their defense stopped the bug's exploitation. In this work, we select one of the Chromium functions that can reach heap meta-data located on the stack. We then show how MPKAlloc prevents an adversary from corrupting meta-data by exploiting this bug. Note that, while our presented attack is artificial, it shows the security consequences of an out-of-bounds stack read which has been seen in the past in Chromium's library dependencies [4] when an additional dereference is present. Within the Blink renderer a function called `ConsumeShorthandGreedilyViaLonghands` populates a shorthand representation of multiple CSS properties using longhand declarations. This Blink function holds an array of pointers within its stack frame in order to update CSS data structures. This function is interesting for two reasons. First, our PIN-tool observed heap meta-data pointers lying at addresses located higher than a function's stack pointer. Second, the function writes to pointers stored on the stack. This implies that if an adversary could trick the function into fetching an element beyond the boundary of the array located on the stack, the function would fetch and write to a heap meta-data pointer. This could give an adversary the opportunity to corrupt meta-data by authoring malicious CSS snippets. To demonstrate MPKAlloc stopping this attack, we altered the vulnerable method to search upwards in memory starting from the stack buffer containing pointers until it detected a meta-data pointer. After writing to the meta-data pointer, we observed MPKAlloc terminate the browser with a segmentation violation.

5.6 Impact on Page Load Times

In order to evaluate the performance impact MPKAlloc has on web browsing, we compared the page load times, defined as the amount of time needed to render a web page and all its dependencies, with an unmodified Chromium browser and with one protected by MPKAlloc. We evaluate MPKAlloc in this way for the top 50 websites given in the Alexa ranking. In order to prevent network latency from skewing our measurements, we utilize the Web Page Replay tool provided by prior work [30] in the Catapult framework [13]. The Web Page Replay tool records the traffic generated when visiting a website, and allows us to replay this traffic while recording measurements in our evaluation. For every website included in our evaluation, we load the website one hundred times and record the amount of time required to load the website and all its external dependencies, such as CSS, Javascript, and image data. We repeat this both with

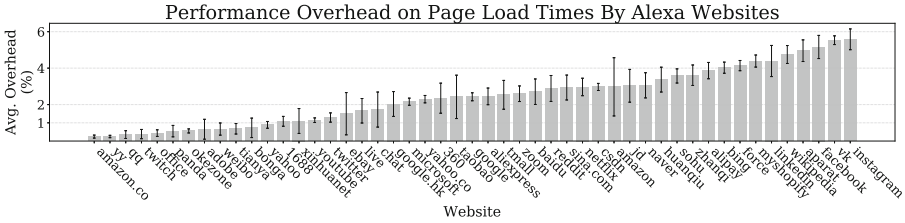


Fig. 8. Average performance overhead incurred by MPKAlloc with standard error while loading Alexa websites.

MPKAlloc enabled in Chromium and with an unmodified Chromium browser with identical versions. After running this experiment, we have one hundred load times from an unmodified Chromium instance and one hundred produced by Chromium protected with MPKAlloc. Overall, the overhead for all websites in our evaluation stayed below 5.58%. While this is the worst case overhead we observed for a single website, MPKAlloc caused 2.44% overhead on average with a geometric mean of 1.71%. Figure 8 summarizes the average performance impact MPKAlloc has on page load times for the top 50 websites given in the Alexa ranking. The main source of overhead introduced by MPKAlloc comes from switching between protection domains which requires issuing the `wrpkru` instruction. It has been observed that the worst case execution time for this instruction is 260 cycles [34]. In our evaluation, we observed that loading a website can incur at most 1.5 million domain switches, which puts a theoretical maximum overhead of a naive implementation of our defense at 130 ms when running on a 3 GHz processor. Many of the web pages in our evaluation take multiple seconds to complete loading, and so this theoretical worst case may not hinder MPKAlloc's use. In addition to page load times, we evaluated MPKAlloc on several Javascript benchmarks in order to understand the impact of MPKs on interpreting Javascript programs, which utilize the browser's general purpose allocators in addition to the V8 garbage collector. We observed MPKAlloc introduce a small amount of overhead on three Javascript benchmarks, Speedometer (0.95%), Octane (0.57%), and Sunspider (0.73%).

6 Conclusion

In this paper, we presented MPKAlloc, a defense that isolates heap meta-data via in-process memory protection domains made possible by recent Intel CPUs. Our prototype implementation of MPKAlloc protects heap meta-data used by the `tcmalloc` and `PartitionAlloc` memory allocators. We evaluated MPKAlloc on the SPEC CPU2006 benchmarks and showed that it induces merely 0.8% performance overhead on average. Furthermore, when used in Chromium, MPKAlloc detects and prevents potential exploits. Finally, MPKAlloc protects Chromium while introducing a geometric mean of 1.71% performance overhead (2.44% on average) on page load times.

References

1. jemalloc. http://jemalloc.net/. Accessed 31 Mar 2021
2. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html. Accessed 31 Mar 2021
3. Storage protect keys. https://www.ibm.com/docs/en/aix/7.2?topic=concepts-storage-protect-keys. Accessed 16 Aug 2021
4. CVE-2016-10195 (2016). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10195. Accessed 04 May 2022
5. Memory tagging extension: Enhancing memory safety through architecture (2019). https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety. Accessed 27 Feb 2022
6. Educational heap exploitation (2021). https://github.com/shellphish/how2heap. Accessed 31 Mar 2021
7. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**(1), 1–40 (2009)
8. Ainsworth, S., Jones, T.M.: MarkUs: drop-in use-after-free prevention for low-level languages. In: IEEE Symposium on Security and Privacy (2020)
9. Anonymous: Once upon a free(). http://phrack.org/issues/57/9.html. Accessed 14 Mar 2021
10. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Commun. ACM **57**(2), 74–84 (2014)
11. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: ACM ASIA Conference on Computer and Communications Security (2011)
12. Cha, M.H., Lee, S.M., An, B.S., Kim, H.Y., Kim, K.H.: Fast and secure global-heap for memory-centric computing. J. Supercomputing **77**, 13262–13291 (2021). https://doi.org/10.1007/s11227-021-03806-4
13. Chromium authors: Catapult. https://chromium.googlesource.com/catapult. Accessed 12 Oct 2021
14. Chromium Authors: Deploy PartitionAlloc-Everywhere. https://bugs.chromium.org/p/chromium/issues/detail?id=1121427. Accessed 12 Oct 2021
15. Connor, R.J., McDaniel, T., Smith, J.M., Schuchard, M.: PKU pitfalls: attacks on PKU-based memory isolation systems. In: USENIX Security Symposium (2020)
16. Delshadtehrani, L., Canakci, S., Blair, W., Egele, M., Joshi, A.: FlexFilt: towards flexible instruction filtering for security. In: Annual Computer Security Applications Conference (2021)
17. Demeri, A., Kim, W.H., Krishnan, R.M., Kim, J., Ismail, M., Min, C.: POSEIDON: safe, fast and scalable persistent memory allocator. In: International Middleware Conference (2020)
18. Farkhani, R.M., Ahmadi, M., Lu, L.: PTAuth: temporal memory safety via robust points-to authentication. In: USENIX Security Symposium (2021)
19. Hedayati, M., et al.: Hodor: intra-process isolation for high-throughput data plane libraries. In: USENIX Security Symposium (2019)
20. Heelan, S., Melham, T., Kroening, D.: Automatic heap layout manipulation for exploitation. In: USENIX Security Symposium (2018)
21. IBM Corporation: Power ISA version 3.0b (2017)
22. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (2016)

23. Kim, Y., Lee, J., Kim, H.: Hardware-based always-on heap memory safety. In: IEEE/ACM International Symposium on Microarchitecture (2020)
24. Kirth, P., et al.: PKRU-safe: automatically locking down the heap between safe and unsafe languages. In: European Conference on Computer Systems (2022)
25. Koning, K., Chen, X., Bos, H., Giuffrida, C., Athanasopoulos, E.: No need to hide: protecting safe regions on commodity hardware. In: European Conference on Computer systems (2017)
26. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not. **40**(6), 190–200 (2005)
27. Otto Moerbeek: A new malloc(3) for openBSD. http://www.openbsd.nl/papers/eurobsdcon2009/otto-malloc.pdf. Accessed 23 Mar 2021
28. Park, S., Lee, S., Xu, W., Moon, H., Kim, T.: libmpk: Software abstraction for intel memory protection keys (intel MPK). In: USENIX Annual Technical Conference (2019)
29. Park, T., Dhondt, K., Gens, D., Na, Y., Volckaert, S., Franz, M.: NOJITSU: locking down javascript engines. In: Network and Distributed System Security Symposium (2020)
30. Reis, C., Moshchuk, A., Oskov, N.: Site isolation: process separation for web sites within the browser. In: USENIX Security Symposium (2019)
31. Robertson, W.K., Kruegel, C., Mutz, D., Valeur, F.: Run-time detection of heap-based overflows. In: Conference on Systems Administration (2003)
32. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the ×86). In: ACM Conference on Computer and Communications Security (2007)
33. Solar Designer: JPEG COM Marker Processing Vulnerability. https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability. Accessed 23 Mar 2021
34. Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N.O., Sammler, M., Druschel, P., Garg, D.: ERIM: secure, efficient in-process isolation with protection keys (MPK). In: USENIX Security Symposium (2019)
35. Yun, I., Kapil, D., Kim, T.: Automatic techniques to systematically discover new heap exploitation primitives. In: USENIX Security Symposium (2020)
36. Yun, I., Song, W., Min, S., Kim, T.: HardsHeap: a universal and extensible framework for evaluating secure allocators. In: ACM Conference on Computer and Communications Security (2021)
37. Zhao, Z., Wang, Y., Gong, X.: HAEPG: an automatic multi-hop exploitation generation framework. In: Maurice, C., Bilge, L., Stringhini, G., Neves, N. (eds.) DIMVA 2020. LNCS, vol. 12223, pp. 89–109. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-52683-2_5