ISLAB: Immutable Memory Management Metadata for Commodity Operating System Kernels

Marius Momeu*
TU Munich
Munich, Germany
marius.momeu@tum.de

Fabian Kilger
TU Munich
Munich, Germany
f.kilger@tum.de

Christopher Roemheld
TU Munich
Munich, Germany
chris.roemheld@tum.de

Simon Schnückel
TU Munich
Munich, Germany
simon.schnueckel@tum.de

Sergej Proskurin
BedRock Systems
Munich, Germany
sergej@bedrocksystems.com

Michalis Polychronakis
Stony Brook University
Stony Brook, NY, USA
mikepo@cs.stonybrook.edu

Vasileios P. Kemerlis
Brown University
Providence, RI, USA
vpk@cs.brown.edu

ABSTRACT

Kernel memory allocators maintain several metadata structures optimized for efficiently managing system memory. However, existing implementations adopt either weak or no protection at all to ensure the integrity of said metadata in the presence of memory errors. In this paper, we first demonstrate how existing memory hardening schemes fall short against several in-kernel memory corruption scenarios. We then present ISLAB: a set of novel (slab-based) heap hardening techniques that aim to ensure the integrity of the memory managed by the kernel, and minimize the incurred runtime, and memory, overhead. ISLAB prevents memory corruption exploits by segregating metadata from within corruptible memory objects into shadow memory. It also relies on a novel SMAP-assisted memory isolation framework, called kSMAP, to protect allocator metadata against adversaries with stronger memory access capabilities. We implemented and evaluated ISLAB atop SLUB, the default slab allocator in Linux, and equipped it with kSMAP to protect process credentials, a popular target in kernel exploitation. Our experiments show that ISLAB incurs no runtime overhead in realistic benchmarks, and moderate overhead in stress tests. Lastly, we show how ISLAB's approach can be generalized to protect the integrity of other kernel subsystems that use corruptible metadata for memory management, such as linked lists.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Software security engineering**.

KEYWORDS

kernel hardening, heap protection, memory-metadata isolation

*Also with Brown University.

ASIA CCS '24, July 1–5, 2024, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Marius Momeu, Fabian Kilger, Christopher Roemheld, Simon Schnückel, Sergej Proskurin, Michalis Polychronakis, and Vasileios P. Kemerlis. 2024. ISLAB: Immutable Memory Management Metadata for Commodity Operating System Kernels. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Leveraging the way dynamic-memory allocators manage memory objects has become an increasingly useful attack vector in developing exploits based on memory safety vulnerabilities. Specifically, in the presence of a memory error, attackers can take control of an object's *metadata*, and use the allocator to acquire arbitrary *read* and *write* primitives within the respective address space [80]. In response, a plethora of works have been proposed to mitigate the issue above, but they have been primarily focusing on hardening the dynamic-memory allocators used by user-space applications [2, 3, 11, 27, 28, 30, 65, 71, 89, 90, 92, 102, 112]. Surprisingly, their kernel-space counterparts have received less attention, and remain popular targets for adversaries [4, 5, 32, 34, 37, 39, 68, 70, 84] and offensive research [15, 16, 20, 86, 87]. Thus, addressing the security weaknesses of dynamic-memory allocators in kernel space remains a key challenge, and is the focus of this work.

SLUB, the default heap allocator in the Linux kernel, maintains a list of pointers to freed (i.e., unallocated) objects, known as a *freelist*, which allows for constant-time (de-)allocations. However, to leverage the speedup benefits of hardware caching and TLBs, and to reduce memory consumption, each SLUB object stores its own freelist pointer. This design/weakness has been leveraged by the aforementioned exploits to successfully compromise the kernel in the presence of memory errors. Sadly, this is a recurring optimization pattern adopted by several Linux subsystems. For example, linked lists, a ubiquitous structure used by the kernel, are designed such that an object's list neighboring pointers are maintained within the object itself, a design weakness that has been abused by several exploits [25, 31, 32, 97, 106].

These shortcomings, however, are not entirely overlooked by the kernel community. For instance, SLUB's freelist pointers are *XOR-encrypted* before being stored in potentially compromised objects. Unfortunately, per our analysis (see Section 4.1), SLUB uses an XOR-based cipher-block encryption algorithm in ECB mode to

protect freelist pointers, which cannot provide strong guarantees about the integrity of the encrypted data [64]. As we demonstrate in Section 4.3, this pointer protection scheme can still be circumvented under the adversarial capabilities presumed by existing exploits. Additionally, although linked lists benefit from list integrity checks, they can only detect trivial memory errors, falling short against more advanced ones (we discuss this limitation in Section 4.1).

Considering the lack of robust hardening in the kernel's heap allocator, we present ISLAB: a novel hardening scheme that ensures the integrity of metadata used by kernel-space slab allocators. ISLAB enhances SLUB by leveraging a simple, yet effective technique: it places allocator metadata into segregated memory, outside the attackers' reach. We carefully design ISLAB to use CPU data caches and TLBs efficiently, avoiding any increase in (system-wide) performance and memory overheads. Additionally, we demonstrate how this technique can be adopted by other kernel subsystems that manage memory via in-object metadata, such as linked lists. To that end, we introduce ILIST, a kernel extension that borrows ideas from ISLAB and segregates list pointers into shadow memory.

Segregating sensitive pointers from corruptible objects protects against certain vulnerabilities, such as localized overflows/underflows and use-after-frees. However, kernel objects expose several other pointers that could be abused by adversaries to obtain read/write access to sensitive data. Previous works [36, 53, 61, 73, 80, 81, 83, 96, 100, 105] demonstrated that such threats can be countered by shielding memory contents via hardware extensions (e.g., Intel VT-x, MPK, CET, SMAP [38]). However, existing intra-kernel memory isolation solutions exhibit suboptimal performance overhead [33, 80], or require major kernel modifications, potentially introducing additional attack vectors [33]. In a departure from such approaches, we build a novel kernel memory isolation framework based on SMAP [38], which allows for lightweight data isolation [100, 104], is supported by virtually *every* contemporary x86 CPU (as opposed to extensions like Intel MPK), and even has an ARM counterpart (i.e., PAN [67]). As such, we introduce kSMAP: a set of novel memory protection primitives based on SMAP.

Moreover, we identified that existing memory isolation solutions still leave several attack vectors open in the face of strong attackers armed with arbitrary memory access primitives. First, although existing schemes protect slabs that store sensitive data [80], they still manage sensitive objects via unprotected metadata that lie outside the slab—and this can be abused for circumventing the isolation boundary. Second, existing isolation-based solutions omit protecting sensitive data, stored in CPU registers, which may temporarily get spilled in unprotected memory on task preemption—a coercing task, running in parallel on a different CPU thread can corrupt them. We highlight all of the above, in more depth, in Section 4.2. On the contrary, kSMAP can be used by many kernel subsystems (SLUB, `struct cred`, and more) to isolate both sensitive data and metadata, not only when tasks are scheduled on the CPU but also when they are preempted, with low overhead.

In summary, our work makes the following contributions:

- We present a security analysis of SLUB and linked lists, as well as of existing kernel isolation schemes, against basic and advanced memory errors, along with proof-of-concept exploits that demonstrate how they can be circumvented.
- We design two innovative memory protection schemes, called ISLAB and ILIST, for hardening kernel slab-based memory allocators, and list managers, against memory errors, without performance slowdowns and high memory consumption.
- We design a novel kernel memory isolation framework, called kSMAP, which relies on SMAP to protect arbitrary kernel memory, and an enhanced variant of ISLAB that uses it to protect data and metadata of select sensitive objects against attackers with arbitrary memory read/write primitives; kSMAP also protects sensitive state during interrupt handling.
- We implement and evaluate ISLAB atop SLUB, the default Linux kernel allocator, and ILIST atop the Linux kernel lists manager and kSMAP, while protecting process credentials.

2 BACKGROUND

2.1 Slab-based Allocation in Linux

The Linux kernel uses primarily two *dynamic-memory allocators* to manage in-kernel memory: a *page allocator* and an *object allocator*. The former leverages the *buddy system* to satisfy memory requests on a page granularity [49]; the latter leverages the *slab* approach to facilitate efficient memory allocation on a sub-page granularity [9]. Linux implements three different variants of the slab allocator: SLOB, SLAB, and SLUB, with the latter being the default [56]. The slab allocator uses the underlying page allocator to reserve one or more physically-contiguous memory pages to form *object slabs* that maintain consecutive objects of the same size (i.e., *type*). This way, object slabs avoid internal fragmentation and efficiently serve (de-)allocation requests for the respective object type.

SLUB uses the data structure `kmem_cache` to manage multiple object slabs (some per-CPU, others per-NUMA node) for a specific object type [12]. The per-CPU slabs are maintained by the `kmem_cache_cpu` structure, which is (de-)allocated via the kernel's `percpu` allocator [94]. The per-NUMA node slabs are managed via the `kmem_cache_node` objects, which are (de-)allocated from their own dedicated slabs cache. At the same time, the kernel stores slab-specific information in the data structure `struct page` [12], representing the physical pages that make up the slab [42]; these are kept in the `vmemmap` [45] area of the kernel address space.

To keep track of free objects inside a slab, SLUB organizes them in linked lists called *freelists* [9], which are stored within the slabs themselves. That is, every unallocated object inside the slab holds a pointer to the next free object, allowing SLUB to reduce memory consumption and the working set of the cache. Furthermore, the `kmem_cache_cpu` and `kmem_cache_node` structures store the per-{CPU, NUMA node} freelist head pointers, while the associated `struct page` of a slab maintains another freelist head pointer, which allows for parallel de-allocations with the per-CPU slab. As we highlight later in Section 4.1, in-slab freelist pointers and out-of-slab metadata are *security-critical*, and ISLAB focuses on protecting both. Many kernel subsystems that manage (collections of) objects also adopt the same idiom for storing *security-critical* metadata. For instance, the doubly-linked list API of the Linux kernel places two pointers—one to the previous and another one to the next list element—directly into an object. As we highlight later in Section 4.1, (doubly-)linked list pointers present *critical* metadata, and hence ILIST focuses on protecting them.

2.2 Supervisor Mode Access Prevention

In response to return-to-user attacks [43], both Intel and AMD introduced SMAP: a processor extension that mitigates rogue user-space memory accesses during kernel execution [38]. SMAP triggers an exception if the CPU accesses a virtual address that is mapped as user-mode memory (i.e., by having the U/S bit set in the respective page tables), while running in Ring 0. Nevertheless, as the kernel frequently accesses user-space memory legitimately, e.g., for fetching user data during a syscall, disabling and enabling SMAP is facilitated by the low-latency stac and clac instructions, which operate on the AC flag of the RFLAGS register. Specifically, SMAP is disabled when the AC flag is set, and enabled otherwise. RFLAGS is a per-CPU register, architecturally spilled/filled on interrupt requests to/from the interrupted task's stack (or the IST [93], if configured), which may facilitate concurrent tasks to access different security domains, without the kernel's intervention. However, as RFLAGS is stored on unprotected (kernel) stacks, the saved AC bit may be set by attackers to force the interrupted task to return with SMAP disabled, thereby allowing access to otherwise inaccessible memory. kSMAP addresses this issue, as we describe in Section 5.2.

Previous works repurposed SMAP to isolate sensitive memory in user processes [100, 104]. Although their solution suffers from notable limitations, which we discuss in Section 8, it proved the effectiveness of SMAP for intra-process memory isolation. Specifically, switching isolation domains via stac/clac exhibits lower latency (≈18 cycles) compared to other alternatives, such as VT-x (2∗vmfunc, ≈292 cycles) and PKU (2∗wrpkru, ≈56 cycles), which have been used by previous works to isolate sensitive data [36, 61, 73, 80, 81, 96]. Thus, kSMAP extends the use of SMAP in kernel-space by isolating sensitive kernel memory, in addition to blocking rogue kernel accesses on user-space memory.

2.3 Memory Errors

Software written in memory- and type-unsafe lagnuages may generally encounter *out-of-bounds (OOB)*, *use-after-free (UAF)*, *double free (DF)*, or *invalid free (IF)* errors on heap-allocated objects [110]. Attackers can exploit such vulnerabilities to corrupt or leak the *data* stored in allocated objects, or the *metadata* of unallocated ones. Historically, attackers opted for the latter, as corruptible metadata represents a straightforward way to gain *arbitrary* read/write capabilities in both user applications [76, 88] and kernel code [34, 54, 70, 101].

For example, by modifying the next pointer in a free object on a freelist, SLUB (and other freelist-based allocators) can be tricked into giving attackers access to targeted addresses. These can be crafted to overlap with sensitive data of other objects, such as function pointers or process credentials, which attackers can further corrupt to obtain code execution or escalate their privileges [42, 43, 75, 91]. Such primitives can also be obtained by corrupting the metadata in linked list entries. In Section 4.1, we discuss in how SLUB fails to mitigate many real-world exploitation scenarios, and highlight the security improvements brought by ISLAB.

3 THREAT MODEL

We assume attackers that have complete control over an *unprivileged* user, seeking to exploit memory errors in kernel code [110].

Attackers may trigger vulnerabilities (e.g., OOB, UAF, DF, IF) through the interaction with the OS via buggy kernel interfaces, such as pseudo-filesystems (procfs, debugfs [18, 48]), the system call layer, and virtual device files (devfs [51]). In all such scenarios, ISLAB prevents the slab freelist pointers from being tampered with, as they are stored in shadow memory. ISLAB can also detect and prevent exploits that abuse (some) DF and (all) IF vulnerabilities to *directly* manipulate data stored in allocated objects, such as function pointers or process credentials, which can lead to code-reuse [78] or data-oriented [80] attacks. Although ISLAB alone does not mitigate attacks that abuse OOB or UAF bugs to tamper with (generic) object data, ILIST can prevent those that target the metadata of doubly-linked lists [25, 31, 32, 97, 106]. ILIST, however, does not guard general-purpose (data, code) pointers stored in memory objects.

We assume an attacker armed with arbitrary R/W capabilities who tries to mount data-oriented attacks by corrupting sensitive in-slab objects, s.a., process credentials, and their out-of-slab management metadata, like the fields of a struct page, or of the kmem_cache_{node, cpu} structures, which manage the sensitive slabs. ISLAB defends against such attacks by protecting both in-slab data and out-of-slab metadata via kSMAP. We assume pointers to the objects isolated by kSMAP are protected against replay attacks via existing bidirectional referencing schemes [14]. In addition, to prevent attackers from clearing the U/S bit of isolated pages, we assume that page tables are protected via orthogonal techniques, such as PTRand [22] or HLAT [85].

Moreover, we assume that attackers can mount code reuse attacks to circumvent kSMAP's isolation domain, by manipulating code pointers stored on the stack or in memory objects, respectively. We prevent this by scanning the kernel code (including loadable modules) via existing techniques [96] and by instrumenting stray stac instructions with a matching clac. Note that legitimate stac/clac sequences do not require additional *security checks* to uphold isolation, such as the call gates proposed in prior domain-based isolation work [36, 96, 98], since clac always enables isolation.

Furthermore, kSMAP enables in-kernel memory isolation by marking kernel memory as U/S = 1, making it accessible in user space. To prevent user processes from accessing it, kSMAP requires separate page tables for user processes and the kernel, which is currently provided by Kernel Page Table Isolation (KPTI) [35, 44]. Moreover, we assume the attacker can synchronize coercing tasks on different CPU threads, and corrupt the preemption state of one of them while it processes sensitive data. ISLAB with kSMAP is able to prevent this scenario by also isolating the preempted state, rendering such data-oriented attacks ineffective. Finally, we assume that adversaries cannot load kernel-level rootkits [52, 99], while attacks exploiting micro-architectural flaws [13, 50, 59, 107] are out of scope.

4 PROBLEM STATEMENT

4.1 Linux SLUB and Lists Manager

SLUB stores slab freelists within the object slabs themselves, making them prone to corruption via memory errors in kernel code [15, 16, 110]. On one hand, equipped with UAFs, DFs, IFs, or OOBs within a victim slab, or arbitrary memory corruption primitives, attackers may choose to tamper with security-critical fields of allocated victim objects [80]. On the other hand, they can target freelist pointers

in freed chunks, which grants them strong exploitation primitives onto security-critical objects [34, 54, 62, 70, 101]. Although SLUB does not prevent attacks that target the former, it currently implements hardening techniques that hinder the latter. In particular, before storing a freepointer in a freed slab object, SLUB encrypts it using a trivial block cipher algorithm in ECB mode [41]. The algorithm uses the eight-byte freepointer as *plaintext*, an eight-byte random value generated per-object cache as the *secret key*, the XOR operation as the block cipher encryption method, and the freepointer address as an eight-byte *salt* to prevent replay attacks.

Block cipher encryption in ECB mode only guarantees the confidentiality of the encrypted blocks, but not their integrity [64]. In the case of SLUB, encrypted ciphertexts can still be corrupted and go unnoticed as long as the decrypted freepointer yields a valid slab address. For example, corrupting the first 12 bits of a freepointer ciphertext is *guaranteed* to decrypt to a valid address within the slab's page, making UAF-based or DF-based exploits possible. Consequently, it is clear that ECB-mode encryption is not suitable for ensuring pointer integrity. Moreover, the current encryption implementation XORs the most-significant bits of the freepointer with the least-significant bits of the salt, which are equal for every slab object, thus failing to provide sufficient uniqueness against replay attacks on the least significant bits of the ciphertext.

Furthermore, SLUB also adopts a trivial protection against DFs by only making sure that the object at the top of a freelist is not the victim of the DF. While this could suffice against some exploits, such as CVE-2017-2636 [79], it cannot detect DF cases where the victim object lies at *arbitrary* locations within the freelist. As far as IFs go, SLUB prevents IF attempts by checking if the freed address lies within its dedicated slab, which can be easily computed from the freed virtual address (see Section 2.1). Moreover, as it does not incorporate any form of intra-kernel memory isolation mechanism on its slabs, or the out-of-slab metadata that lies in its control structures `kmem_cache{_node|_cpu}`, and `struct page`, SLUB is susceptible to exploits that leverage arbitrary read/write primitives.

Similarly, the kernel maintains an object's linked list pointers (i.e., `next` and `prev`) within the object itself, where they lie exposed to attacker-controlled data. Although the kernel binds a list object to its neighbors (by checking whether their `next` and `prev` point back to the object), it only performs these checks on a subset of all list operations. Additionally, this hardening scheme can be bypassed by attackers who can craft forged list objects with a valid backward/-forward pointer to the victim object [42, 58].

4.2 Selective Memory Protection

Several hardware-based in-kernel memory isolation frameworks exist [14, 33, 80] that aim to protect sensitive kernel data, s.a., `struct cred` and page tables. However, their focus is on isolating the slabs where the sensitive data are stored, while leaving unprotected the *out-of-slab metadata* that manages it. For example, both xMP [80] and PrivWatcher [14] omit protecting the freelist pointer stored in the `struct page` objects of slabs that store `struct cred` objects. Moreover, to avoid introducing performance overhead via frequent domain switches, PrivWatcher manages the isolated `struct cred` objects via freelists that are stored in regular, unprotected, memory.

Attackers with omnipotent read/write capabilities may target such sensitive metadata and craft a sequence of operations that lead to corrupting the critical data, effectively bypassing the isolation domain. We demonstrate how this weakness can be abused via a concrete exploitation scenario described in Section 4.3.

Additionally, in order to facilitate execution preemption, isolation-based techniques must save a preempted tasks' isolation state (i.e., trusted vs. untrusted) and restore it when the task gets re-scheduled. Specifically, xMP must manage the EPT index, while IskiOS [33] must manage the value of the PKRU register. This opens a window of opportunity for attackers as they can now target the saved isolation state and restore a malicious task with isolation disabled. xMP addresses this by protecting the saved EPT index via *hash-based message authentication codes* (HMACs) implemented in software, which are, however, susceptible to replay attacks (in certain scenarios) as demonstrated by similar work on pointer hardening [109]. As far as IskiOS and PrivWatcher goes, they do state how they handle this scenario.

Moreover, none of these isolation frameworks prevent interrupts from being triggered while tasks process sensitive data with isolation disabled. This leads to the CPU registers that potentially store sensitive fields to be temporarily stored on the task's unprotected interrupt stack, where attackers may *corrupt* them. For example, attackers may corrupt the stack-saved state of a task that got interrupted while the kernel is checking the `uid` field of a `struct cred` object before opening a privileged file (s.a., `/etc/shadow`) for writing. Upon resuming the execution, the restored register state contains data fetched from corrupted memory, effectively allowing the malicious task to circumvent the isolation domain.

4.3 Proof-of-Concept Exploits

For conducting our security evaluation (see Section 6.4), we surveyed several known exploits against CVEs found in the Linux kernel. We discovered two of them [47, 54] that target SLUB's freelist obfuscation hardening. However, they first need to leak the obfuscated freepointer, and they require XOR'ing with `NULL` as operand (typically done on its last object to mark the end of a freelist). We complement their approach by demonstrating that attackers can target obfuscated freepointers arbitrarily, without requiring leaking them first. For that, we adapt an existing exploit (E_1) that originally leveraged SLUB's freelists without obfuscation enabled, and demonstrate how it is still effective even with freelist obfuscation in place. Additionally, we adapt another exploit (E_2) and demonstrate how the complex hierarchy of memory management structures adopted by SLUB can be abused to corrupt sensitive data even when said data is isolated with existing schemes [33, 80].

4.3.1 E_1: CVE-2021-27365 [34]. The exploit abuses CVE-2021-27365, which reveals that `iscsi_host_get_param()` does not check the length of a previously set attribute—this can be abused by attackers to overwrite the contents of an adjacent slab. The exploit also uses CVE-2021-27363 and an additional information leak to circumvent KASLR, and get the base address of the kernel and the modules. It then uses these addresses to corrupt the freelist pointer of a freed object from the adjacent slab, making it point to a security-critical object in the `.data` section of a module.

Subsequent allocations will return the security-critical object onto a memory chunk that is attacker-controlled, thus allowing the attacker to tamper with its contents and gain local privilege escalation. As also noted by the authors, the exploit is unsuccessful on Linux distributions that enable freepointer encryption in object slabs. To demonstrate the contrary, we adapted the exploit, using the techniques discussed in Section 4.1, and successfully bypassed SLUB's freelist hardening to compromise the kernel.

First, we performed *heap spraying* to polute the target slab until all but a single object were under our control. We then leveraged the overflow in the allocated objects and overwrote the slab index bits of the remaining object's encrypted freelist pointer (i.e., bits [12:14], since the vulnerable object's size is 4K), with an arbitrary value, which is guaranteed to point onto one of our allocated slab objects upon decryption. We then triggered two more allocations in SLUB, at the end of which we had one extra object under our control than the slab normally allows. Next, we drained the slab by freeing all but the extra object, triggering SLUB to free the slab page back to the buddy allocator. While still holding an active reference to the freed slab, we requested the allocation of a security-critical object from an empty slab, for which SLUB had to allocate a fresh page from the buddy allocator. The returned page was the one we still had a dangling pointer on, allowing us to manipulate the contents of the security-critical object and compromise the kernel.

4.3.2 E_2: CVE-2021-41073 [97]. The exploit abuses CVE-2021-41073 found in the `io-uring` subsystem, which allows an attacker to invoke `kfree` onto an allocated object from the `kmalloc-32` cache, leading to a UAF on the victim object. The attacker uses this to build a stronger primitive, which it first uses to leak both KASLR and the address of its own task's `struct cred` object, and then to inject a fake eBPF program that elevates the attacker's privileges by overwriting the `uid` field of its leaked credential. (ISLAB renders the attack unsuccessful, since eBPF programs lie outside the SMAP domain, where they cannot access `struct cred` objects.)

Nevertheless, instead of tampering-with `struct cred` objects directly, we adapted the exploit and aimed to elevate our privileges by manipulating the metadata used to manage `struct cred` slabs. Specifically, we obtained the address of the `struct page` object corresponding to our `struct cred`—this is trivial to achieve since `struct page` objects lie at a fixed offset from their slabs in memory. Then, we overwrote the freelist pointer stored in the victim `struct page` with the address of our task's `struct cred`. Next, we spawned an SUID-set program, e.g., `passwd`, which triggered the allocation of a new `struct cred` with elevated privileges. The allocator returned the freelist head from `struct page`, which points to our unprivileged `struct cred`, and overlapped it with the privileged credential, thereby resulting in (local) privilege escalation.

5 DESIGN AND IMPLEMENTATION

ISLAB and ILIST achieve metadata integrity by *segregating* them from within memory objects, in accordance to our threat model (see Section 3). Specifically, they "pull out" the *slab freelist* and *object list-pointers*, and store them into separate, *shadow* memory regions. However, in addition to ensuring the *integrity* of memory management metadata, our schemes aim to inflict *no slowdown* on the system's runtime performance and *minimize* memory consumption.

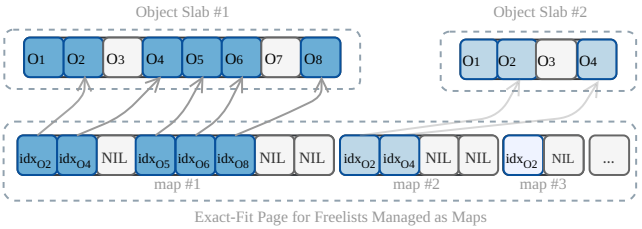

Figure 1: ISLAB manages slab freelists as maps allocated via an exact-fit allocator. Colored objects in slabs are unallocated; white objects are allocated. Each slab object has the same index in the freelist map.

As such, we explored several designs and determined that leveraging *idioms* of SLUB and linked lists to craft custom metadata segregation techniques leads to more efficient behavior than adopting a universal approach. Furthermore, ISLAB leverages kSMAP, a novel framework for intra-kernel memory isolation, to protect sensitive slabs and their associated out-of-slab metadata against attackers with arbitrary R/W primitives.

5.1 Metadata Segregation

There are several ways to design a secure allocator, which is demonstrated by the plethora of works applied to user-space allocators [2, 3, 11, 27, 28, 30, 65, 71, 89, 90, 92, 102, 112]. However, considering that SLUB has been extensively optimized to fit the needs of today's demanding compute landscape, we set out to secure it via a conservative strategy, aiming to keep the modifications introduced by ISLAB to a minimum. As such, ISLAB is also designed to manage free objects via freelists, mainly because they provide constant time (de-)allocations—however, and most importantly, ISLAB *segregates* freelists in shadow memory. We rejected equipping ISLAB with bitmaps [26], since they require linear time (in the worst case) for searching the next free slot in the slab—this may have a significant impact on performance for slabs with a large number of objects (e.g., 512 objects in 8-byte slabs).

Having decided to use segregated freelists in ISLAB, we explored several mechanisms and structures for managing them in shadow memory. A simple solution entails constantly (de-)allocating shadow memory slots to store/release freepointers of slab objects every time they are (de-)allocated by SLUB. However, we argue that such an approach poses two downsides for performance: (1) the shadow memory slots need to be (de-)allocated too frequently and (2) the freelist of a slab ends up scattered over multiple cache lines and over multiple pages, leading to misses in the CPU caches and TLBs, on subsequent slab accesses. To avoid both (1) and (2), we design ISLAB's freelists using an approach that requires fewer (de-)allocations, packing multiple slots onto the same cache line and memory page. As such, we introduce freelists that are managed as *segregated maps*. A freelist map mirrors the characteristics of the original in-slab freelists, except that it uses *smaller-length indices* to represent freepointers and its slots lie *adjacent* in the shadow memory. In the following section we describe the details of how freelist maps work.

5.1.1 Segregated Freelists as Maps. We design a customized freelist segregation mechanism that packs all metadata entries of a freelist onto a *minimum* number of cache lines, and onto a *single* virtual page. With this technique ISLAB allocates a chunk of memory that can fit the entire freelist of a slab, and uses it as a *map* to manage object (de-)allocations. The object index in the freelist map is the same as the object index in the slab, granting each slab object a fixed entry in the map. Additionally, since slab objects have a fixed slot in the freelists maintained by ISLAB, we use it to keep the object's allocation status. This helps ISLAB in detecting certain double-free (DF) attempts. Specifically, when an object is allocated, we mark its freelist entry in the map with a magic allocation value. On deallocations, if the magic value is not set, ISLAB detects the DF. ISLAB only needs 2 bytes per entry to store the index of the next free object in the slab, which is large enough to represent all object indices of a slab. Figure 1 displays ISLAB-map's memory layout.

Exact-fit Allocator. ISLAB is faced with the challenge of efficiently (de-)allocating the shadow memory that stores the freelist maps. A straightforward solution for this would be to use SLUB as the memory allocator. However, as freelist maps do not have fixed sizes, a slab allocator is not optimal for ISLAB. For example, after analyzing all possible object sizes in SLUB, we noticed that there is only one object cache with 85 freelist entries (for 48-byte objects). Thus, a slab allocator would reserve an entire slab cache just to manage 48-byte freelist maps. In addition, we found that the possible object sizes can lead to 39 distinct freelist lengths; a slab-like allocator would thus have to maintain 39 different slab caches for these freelist lengths—this hurts performance and wastes memory.

Inspired by the allocator of glibc (ptmalloc [103]), we design a nimble, *exact-fit* memory allocation strategy that configures memory chunks for requested sizes on demand. As such, contrary to slab allocators, the exact-fit allocator may keep objects of different sizes on the same memory page. At first, the exact-fit allocator reserves one large chunk of 8 memory pages (configurable option), which we dub topchunk. On allocation requests, the exact-fit allocator splits topchunk into two smaller chunks: one that exactly fits the requested size, and one which becomes the new topchunk (with the remaining size). When exact-fit chunks are released back to the freelist allocator, they are cached in a linked list for the respective size, which we dub fastbin. Then, subsequent freelist allocations of a size are served first from the fastbin of the requested size, without having to chop topchunk. Consequently, memory pages used for hosting freelists incur small fragmentation, thus reducing memory consumption. Freelists of different lengths lie on the same memory page, effectively reducing TLB pressure.

5.1.2 Segregated Linked Lists as Magazines. Following the same philosophy behind ISLAB, we leverage idioms of linked lists to design ILIST, aiming to pack the metadata of an object list onto a minimum number of cache lines, and thus, a single virtual page. However, ILIST must support object insertion/deletion at/from arbitrary positions within a list. As such, ILIST cannot use a boilerplate map structure to store metadata efficiently; rather, we introduce linked list *magazines*, which can grow in *both* directions, and where entries can be *relocated* while in use. ILIST first allocates a chunk of memory, the size of a cache line, where the initial head of a list is inserted on initialization.

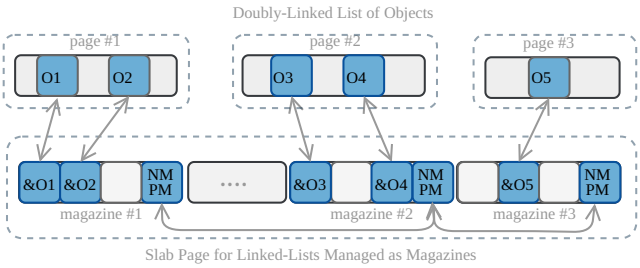

Figure 2: ILIST manages linked lists metadata in magazines allocated by SLUB. Colored boxes within upper pages are objects on a doubly-linked list, and each object maintains a bidirectional reference with its list metadata. NM and PM stand for *next magazine* and *prev magazine* and link the magazines belonging to the same linked list. Unlabeled boxes represent empty metadata slots, where objects can be inserted between their neighbors.

On subsequent object insertions, ILIST iterates all free slots either at the right (head insertions) or left (tail insertions) of the object given as list head, and places the inserted object's metadata on the last one found. If no free neighboring slot is available, ILIST shifts all of its occupied neighbors one position to the right or left, and places the metadata of the inserted object on the freed slot.

Neighboring chunks that are occupied in the magazine represent neighboring objects in the doubly-linked list. If the magazine runs out of free slots, or, due to shifting, the last metadata chunk is pushed out, ILIST allocates a new magazine and links it to the previous one—using each cache line to keep 2 pointers for linking neighboring magazines, and 6 pointers for representing back-references to list-objects. However, note that the magazines belonging to the same object list may end up on different pages, potentially increasing TLB pressure. Deleting an object from the list simply results in wiping its pointer from its associated magazine. When a magazine becomes empty, it is released back to the metadata allocator, and the links between magazines get adjusted. ILIST implements list traversals by walking all linked magazines of a list, collecting non-empty slots within each magazine.

Figure 2 illustrates the layout of ILIST. Contrary to ISLAB, ILIST requires maintaining an individual mapping between each list object and its metadata. Our solution to achieving this is to store within a list object a forward pointer to its associated metadata, which lies *outside* the object in segregated memory. Furthermore, the metadata entry stores a backward pointer to the associated object, which allows ILIST to detect corruptions on the forward reference. This way, ILIST can quickly and securely access an object's metadata by simply referencing its forward pointer [14].

5.2 Data and Metadata Isolation

ISLAB leverages kSMAP to protect both the data and metadata that belong to sensitive objects against memory errors. kSMAP *partitions* the kernel's virtual memory in two *protection domains*: a safe and an unsafe region. The memory that forms the safe region is mapped with the U/S bit enabled in page tables, while the rest of the memory is mapped with said bit disabled.

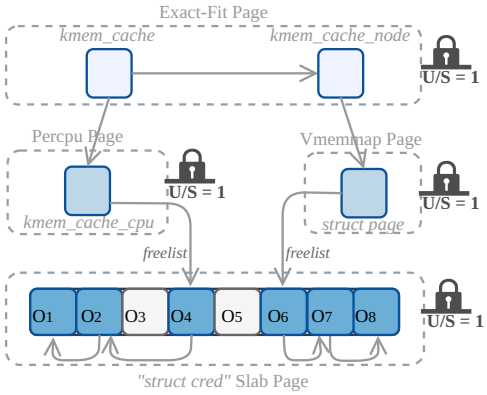

Figure 3: ISLAB-ksmap-cred integrates kSMAP into ISLAB and the struct cred subsystem. The isolated pages are marked as user-mode pages (U/S = 1) in the kernel page table. Access to the isolated struct cred objects is only possible with SMAP disabled. All associated metadata of struct cred objects are isolated likewise.

Relaxing/restricting access to the safe region requires disabling/-enabling AC (i.e., SMAP) in RFLAGS, which is done via the fast (privileged) stac and clac instructions. Note that kSMAP currently leverages KPTI [44] to prevent userland from accessing the SMAP-isolated kernel memory (when executing in user mode; see Section 2.2).

kSMAP's design is intended for serving isolated pages to the slab allocator, which works well with kernel address spaces that map the entire physical memory (i.e., the so-called *direct-mapped/*physmap *area* [42]). On CPU models that support huge pages, the direct-mapped area uses 2MB/1GB pages, which speeds-up address translation. To retain this speedup, kSMAP also isolates direct-mapped addresses in their page table(s) at 2MB granularity. However, we designed kSMAP as a nimble *buddy allocator* that can serve isolated pages of any page order. kSMAP first allocates a 2MB memory block from the page allocator, which it then marks as a user page in its page table entry. (Note that because of KPTI—or, in general, separate user/kernel page tables—memory pages in kernel page tables that are marked as user-mode pages are not accessible in userland.) As the slab or exact-fit allocators request isolated pages, kSMAP seeks the smallest available buddy capable to fit the invoiced page order. If a matching order is found, the isolated memory is simply returned; else, the next available buddy is halved repeatedly until the inquired page order is obtained. On deallocating an isolated page, kSMAP coalesces it with its buddy (if the latter is also freed), and keeps coalescing the resulting buddies until all freed buddies are merged together (hence minimizing memory waste).

To demonstrate its effectiveness, we harden ISLAB with kSMAP and configure it to serve SMAP-protected memory pages for managing struct cred slabs and their associated metadata, as shown in Figure 3, thus guarding them against bogus memory read/-write operations [80]—we dub this variant ISLAB-ksmap-cred. We choose struct cred as a case study for kSMAP since they are one of the main targets (leveraged by kernel exploits) to achieve privilege escalation and take over a system [14].

In what follows, we describe how ISLAB-ksmap-cred works and provide insights about the kernel adjustments introduced by kSMAP.

In-slab Data and Out-of-slab Metadata. We introduce several new flags (s.a., SLAB_KSMAP) that allow configuring slab caches that are protected by kSMAP (ISLAB-ksmap-cred leverages these when creating the cred_jar, thus hosting struct cred objects on protected pages). However, kSMAP never relies on such flags to disable the isolation domain in the various allocators it extends, since kSMAP must maintain them for unprotected slab caches too, where they may get corrupted by attackers. Rather, these are merely used to determine on the execution paths of the slab allocator *when* kSMAP should be invoked instead of the original allocators (s.a., the page and the percpu allocators). Note that all allocator invocations for kSMAP-protected slabs, as well as for kSMAP itself, must be done with SMAP disabled, otherwise the CPU triggers a #PF exception, intercepted by the kernel, which kills the offending task—this prevents attackers from abusing such flags.

Furthermore, ISLAB-ksmap-cred places the kmem_cache and kmem_cache_node structures on isolated pages, allocated by the exact-fit allocator (see Section 5.1.1). Next, it configures the percpu allocator to serve the kmem_cache_cpu structures from pages protected via kSMAP, and isolates the struct page objects associated with the struct cred slabs. Note that percpu places virtual addresses to isolated/protected pages both in the direct-map and the vmalloc region—for kSMAP-enabled slabs the percpu allocator serves SMAP-protected pages from both regions. Lastly, once initialized, it marks the global pointer to kmem_cache of struct cred objects as read-only. These are all the metadata used by SLUB to manage struct cred object allocations, and thanks to the SMAP-based isolation they cannot be tampered with at runtime.

To allow freeing isolated struct cred objects via RCU [46], without switching isolation domains, we create temporary, non-isolated RCU objects, which store a forward reference to the struct cred that needs to be freed. Attackers may corrupt this reference to either replay struct cred objects or forge instances stored in unprotected memory that they control. ISLAB-ksmap-cred stores in struct cred a backward reference to its RCU object, and checks it for validity before freeing in RCU context, which prevents against replaying struct cred objects. ISLAB-ksmap-cred also checks that the freed virtual address is part of the ranges maintained by kSMAP, which ensures that forged struct cred instances are not freed by RCU. To facilitate that, kSMAP stores the PFN (page frame number) [42] of each of its 2MB page in an array, which is traversed by ISLAB-ksmap-cred (and other kSMAP-protected subsystems) to prevent accessing forged memory.

To facilitate legitimate accesses to struct cred objects, we manually instrument all necessary kernel subsystems to temporarily disable the SMAP domain before accessing struct cred, and re-enable it once the access is completed. Our policy is to instrument domain switches at the caller side—that is, kernel functions invoking routines from the struct cred subsystem execute the stac/clac instructions to disable/enable isolation. This allows kSMAP to reduce the number of domain switches by clustering multiple routines that access struct cred objects under a single disable/enable isolation window. For the prototype presented in this paper we instrumented 85 kernel functions, which represents a small fraction of the total number in the codebase (≈524918).

Interrupt State. kSMAP can withstand adversaries who aim to bypass its isolation by corrupting the domain state of an interrupted task. To prevent that, kSMAP must protect two additional components: (1) the saved RFLAGS, which holds the SMAP isolation state (enabled/disabled); and (2) the saved general purpose registers, which may process sensitive data that got fetched by the CPU from SMAP-isolated memory. As we highlighted in Section 2.2, both of these are typically stored on the interrupted task's stack, where attackers with (arbitrary) read/write capabilities can get access to. (Since kSMAP does not target user-mode processes, we do not protect the interrupt state for user tasks.)

First, kSMAP protects the RFLAGS register for every interrupt processed by a kernel task, regardless whether SMAP is enabled or not. This is to prevent the attacker from manipulating the saved RFLAGS of an interrupted task, whose SMAP protection is enabled, and tricking the kernel into resuming with SMAP disabled, thereby getting access to the entire isolated memory part. However, RFLAGS is automatically saved on the stack once an interrupt is processed, which happens architecturally without software intervention. Similarly, RFLAGS is restored, along with the other IRQ-saved registers, automatically via the `iret` instruction upon returning from an interrupt. The kernel has limited control over these architectural operations, thus posing several challenges for kSMAP. Second, kSMAP protects all general purpose registers when an interrupt is triggered, while SMAP is disabled in RFLAGS, which is an indicator that the CPU is processing sensitive data. Otherwise, attackers may corrupt these saved registers, and e.g., overwrite the `uid` field that was fetched from a `struct cred`, undermining our isolation goals. Similarly, upon returning from an interrupt request, general purpose registers are restored from the interrupt stack. However, these are not automatic operations performed by the CPU, and hence remain the responsibility of the kernel.

kSMAP handles both cases by introducing an *isolated interrupt stack* for each task, protected by kSMAP itself, and using it to store sensitive items. Specifically, on each interrupt entry, kSMAP temporarily switches to the isolated stack, and pushes all necessary items depending on the runtime context: if SMAP is enabled, kSMAP pushes only the RFLAGS register; otherwise, it pushes the RFLAGS register (via `pushf`) and all general purpose registers. Upon returning from an interrupt, it switches to the isolated stack, temporarily, and restores the saved items depending on the saved RFLAGS.

Nevertheless, to prevent the CPU from popping RFLAGS from the unsafe stack, which is architecturally performed by the `iret` instruction, we return instead via a snippet of instructions that restore the interrupted state manually (i.e., RIP, CS, RSP, and SS) from the normal stack, and RFLAGS from the isolated stack via the `popf` instruction. This way, attackers cannot tamper with an interrupted task's isolation domain. Note that using a stack structure for storing interrupt state facilitates the system's ability to process multiple (potentially pending and/or recursive) interrupts from a single task, in the same fashion as its regular stack does. Additionally, in order to prevent attackers from corrupting the pointer to the isolated stack, stored in the `task_struct` of each task, kSMAP ensures its integrity via the dual-referencing technique, which it also used to protect `struct cred` pointers in RCU context. We set the size of the isolated interrupt stack to 1KB (configurable option).

5.3 Implementation Details

Our prototype implementation uses Linux kernel v5.11 (commit: `59450bbc`). ISLAB, ILIST, and kSMAP collectively consist of ≈13.3 KLOC in C, primarily added (or edited) in 293 source code files under the following subsystems: `kernel`, `fs`, `mm`, `net`, and `arch`. (A significant part of our patchset touches `include/`, as well as `drivers/`, in order to support our benchmarking testbed.) `ISLAB-ksmap-cred` consists of ≈2.3 KLOC in C, primarily added (or edited) in 74 files under the following subsystems: `arch`, `security`, `mm`, `kernel`, `fs`.

6 EVALUATION

We evaluated ISLAB and ILIST, when used as replacements to Linux's default memory allocator (i.e., SLUB) and lists manager. First, we analyzed their performance and memory overhead, as these are crucial for in-kernel memory managers. Next, we described how our extensions eliminate a significant class of vulnerabilities that SLUB and kernel lists exhibit, and we highlighted attack vectors that still remain open. We conducted separate experiments for evaluating the metadata segregation mechanisms (i.e., ISLAB and ILIST, enabled in turn), and the isolation approach (i.e., `ISLAB-ksmap-cred`) individually. Our tests were carried out on a host armed with a 64-core AMD EPYC CPU (2 sockets, 32 cores/socket), 8 NUMA nodes, and 128GB RAM. Our research prototype is available as open-source software, and can be used by the community for adopting and/or extending our work.

6.1 Runtime Overhead

In order to evaluate the runtime overhead of our extensions, we deployed a set of micro- and macro-benchmarks that make extensive use of SLUB (for testing ISLAB), kernel lists (for testing ILIST), and `struct cred` objects (for testing `ISLAB-ksmap-cred`). Additionally, we collected the results on equivalent benchamrks from the xMP paper [80], which protects `struct cred` objects via virtualization extensions, and compared them with the results on `ISLAB-ksmap-cred`, highlighting our superior runtime efficiency.

6.1.1 LMbench. We employed the LMbench [63] micro-benchmark to examine our extensions on tests that stress individual components of the underlying kernel. Figure 4 shows that the worst-case slowdown incurred by ISLAB is 7% for `open/close` and `fork+exit`. In all other tests, ISLAB incurs negligible (< 5%) or no slowdown at all. Figure 4 also shows that ILIST exhibits moderate slowdown in less than $\approx\frac{1}{3}$ of the tests, which operate on lists of network/IPC packets; a maximum slowdown of 20% is observed on `UDP socket`. However, in all the other tests, ILIST exhibits negligible or no slowdown at all. `ISLAB-ksmap-cred` enables extra protection for both `struct cred` slabs and their out-of-slab metadata, hence exhibiting considerable slowdown on tests that stress the filesystem: i.e., up to 1.42x overhead on `stat` and 1.40x on `open/close`. However, $\approx\frac{2}{3}$ of the tests exhibit negligible or no slowdown at all. Figure 4 also shows that we significantly outperform xMP in almost all tests, and are only slightly slower in `sig deliver`. While xMP incurs 2.5x slowdown on `read` and `write`, `ISLAB-ksmap-cred` only incurs 1.22x and 1.29x. On `stat`, `fstat`, and `open/close` xMP incurs 1.52x, 2.07x, and 1.76x, respectively.

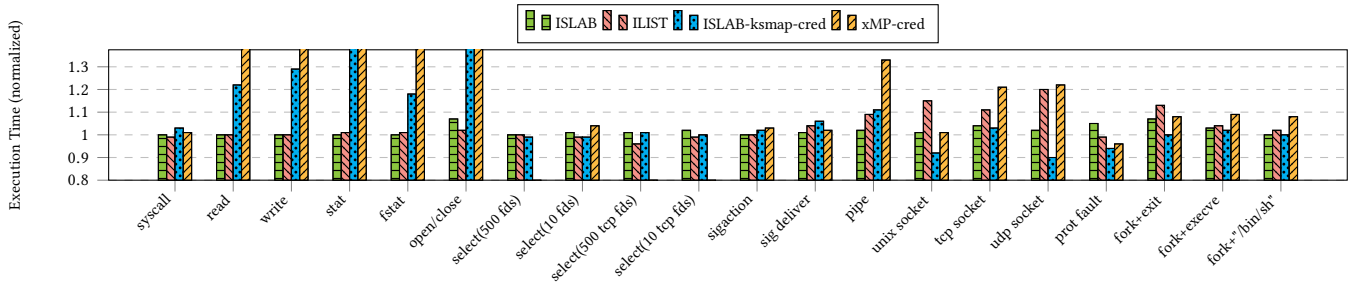

Figure 4: LMbench results on `ISLAB`, `ILIST`, and `ISLAB-ksmap-cred`, averaged over 10 runs and normalized to the unmodified SLUB and lists manager, respectively.

6.1.2 Phoronix Test Suite. LMbench is a collection of stress-tests that do not necessarily capture the performance of ISLAB and ILIST on real-world, end-to-end workloads. Moreover, LMbench does not utilize the available CPU cores to their full potential, as only a few of them are active on tests that stress the scheduler. We therefore conducted additional experiments via several macro-benchmarks from the Phoronix Test Suite (PTS) [77]. Figure 5 shows the results when our extensions are used vs. the baseline (i.e., unmodified SLUB and lists manager). While ISLAB encounters no overhead in all tests, ISLAB-ksmap-cred exhibits merely 1% slowdown on {unpack, compile}-linux. ILIST encounters worst-case overhead of 5% on hackbench and 2% on unpack-linux, while exhibiting no slowdown in all the rest. Figure 5 also shows that we outperform xMP in five tests, are even in two, and slightly slower on gnupg. While on apache xMP incurs 1.09x slowdown, we incur none.

ISLAB-ksmap-cred's source of overhead is mainly attributed to kSMAP's domain switches when accessing isolated objects (`struct cred` and their associated out-of-slab metadata), and when (re-)storing the interrupt state on the isolated stack during interrupt handling. However, since our segregation mechanisms mostly introduce additional memory accesses on the code paths of SLUB and linked lists, we conducted measurements at the micro-architectural level to better understand the origins of ISLAB's and ILIST's occasional slowdown. Specifically, we profiled the execution of hackbench using `perf` [1], which collects additional runtime information via the CPU's performance monitoring counters (PMCs). The PMCs track hardware events (e.g., memory access patterns) relevant for our investigation since they influence runtime performance. We configured `perf` to collect the number of misses encountered when the CPU accessed the L1 and L2 data caches, the L3 cache, and the L1 and L2 data TLB. Any miss in these units incurs latency penalties as the CPU must access components that require additional cycles. Note that `perf` itself also introduces inference, and, therefore, the execution time is expected to differ from the previous runs.

Figure 6 and Figure 7 summarize the cache and TLB accesses collected via PMCs for ISLAB and ILIST, respectively, but plotted separately for clarity. Surprisingly, in Figure 6, ISLAB inflicts a similar number of L1-, L2-, and L3-cache misses compared to SLUB. Additionally, Figure 6 shows that ISLAB also exhibits superior L2 DTLB access, as the measured number of misses is significantly smaller than SLUB's, while the number of misses in the L1 DTLB is similar. These measurements indicate that ISLAB-map's strategy of keeping freelists belonging to multiple slabs on the same

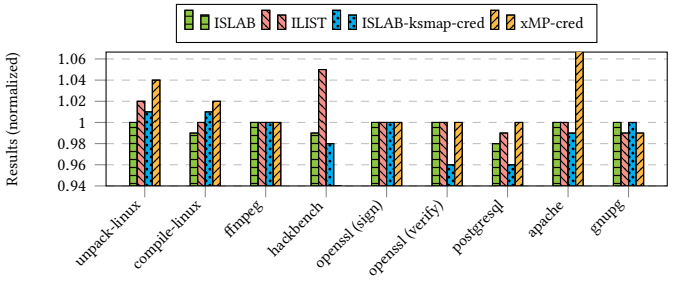

Figure 5: Performance overhead of macro-benchmarks from PTS, deployed in turn on `ISLAB`, `ILIST`, and `ISLAB-ksmap-cred`. The default parameters were used for each benchmark, except for hackbench (16 proc. groups), pgbench (100 scaling factor, 100 clients), and apache (100 concurrent requests) to saturate (i.e., increase the utilization of) the CPU.

cache line and memory page leads to fewer data cache and TLB misses than SLUB (on hackbench), despite the additional memory (de-)allocations and protection against DF attempts.

Figure 7 shows that ILIST inflicts a slightly higher number of L1-, L2-, and L3-cache misses compared to the unmodified lists manager. These measurements correlate with the system's execution time, which exhibits slowdowns as soon as the number of cache misses increases. Furthermore, Figure 7 shows that ILIST exhibits a higher number of misses in the L1 and L2 DTLB than the lists manager.

6.2 Memory Overhead

ISLAB and ILIST require additional memory for storing the segregated metadata for the hardened slab allocator and lists manager, respectively. To evaluate their memory consumption, we counted the number of physical pages used by ISLAB and ILIST after the OS has finished booting. This measurement demonstrates sufficiently enough the performance of ISLAB and ILIST, as far as memory consumption goes, because Linux allocates a large number of object slabs (≈15K) and list metadata objects (≈346K) during bootstrap.

ISLAB required 320 physical pages to store its freelist entries. It maintains 2 bytes per freelist entry and uses the exact-fit allocator, which stores chunks (freelists) of arbitrary sizes on the same physical page(s), and caches them in `fastbins` once they are freed. Based on our analysis on possible slab object sizes supported by SLUB, we noticed that they can lead to 39 distinct freelist lengths.

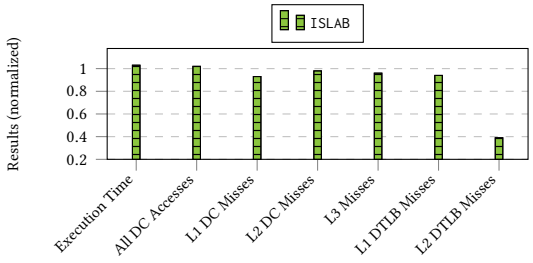

Figure 6: Execution time vs. data-cache and TLB misses on hackbench for `ISLAB`. The `loops` parameter of hackbench was set to 50000 and `num groups` to 16. Results are averaged over three runs and normalized to vanilla `SLUB`.

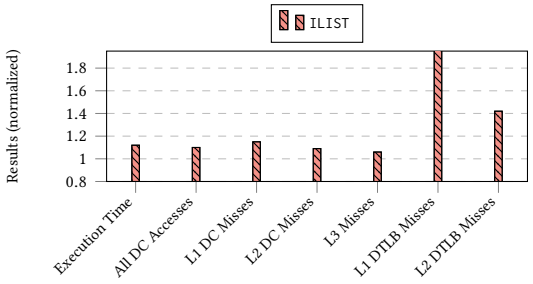

Figure 7: Execution time vs. data-cache and TLB misses on hackbench for `ILIST`. The `loops` parameter of hackbench was set to 50000 and `num groups` to 16. Results are averaged over three runs and normalized to vanilla lists manager.

While some are unique for a single object-size, others recur across different object-sizes, which increases the likelihood that cached freelists can be reused by subsequent slabs. This empowers `ISLAB` to keep its internal fragmentation, and consequently, its memory consumption, low. `ILIST` required 1127 memory pages to store the list metadata stacks, and 80 pages to store the metadata entries that must be freed in RCU context. It maintains 8 bytes per list entry (backward references) and needs 2 additional pointers per magazine to link them (i.e., 16 bytes). Also, magazines may have empty "holes" in between list objects, to maintain the list order, thus exhibiting some internal fragmentation.

`kSMAP` increases memory consumption in the hosting kernel with each 2MB chunk allocated from the page allocator. However, once it reserves and isolates the 2MB chunks, `kSMAP` further allocates buddies of arbitrary page orders to other kernel subsystems, such as `ISLAB`, just like the page allocator. Additionally, `kSMAP` reserves one page per task to configure an isolated interrupt stack, used to store the task's sensitive state (`RFLAGS` and general purpose registers) during interrupt handling. Nevertheless, the isolated stack is freed back to `kSMAP` once the task is terminated. During the aforementioned experiment, `kSMAP` held 121 pages in use for the isolated stacks. In addition, `ISLAB-ksmap-cred` further allocates one additional temporary object for every `struct cred` object freed with the RCU mechanism. However, these temporary objects are returned to the system once the `struct cred` objects are freed (i.e., after the RCU grace period). In our experiment, `kSMAP` held 51 pages in use for `cred`-RCU objects. Moreover, `kSMAP` uses additional memory for the two object caches: (1) for isolated stacks and (2) for `cred`-RCU objects. Specifically, these require 320B for the `kmem_cache` itself, 64B per NUMA-node for the `kmem_cache_node` objects, and 32B per active CPU for the `kmem_cache_cpu` objects. Furthermore, `ISLAB-ksmap-cred` does not require additional shadow memory for segregating freelist pointers of `struct cred` objects, as it keeps them within the slab, where they lie isolated. Overall, after bootup `kSMAP` held 4MB in use—negligible for today's RAM sizes.

6.3 Security Analysis

6.3.1 `ISLAB`. `ISLAB` (§5.1.1) protects corruptible freelist pointers by storing them outside of the slab objects themselves. This guarantees both their integrity and confidentiality in the presence of UAFs, DFs, IFs, and OOBs, which target heap allocator metadata, as attackers cannot reach the segregated freelist pointers—unless they possess *arbitrary* read/write primitives. Nevertheless, similarly to `SLUB`, `ISLAB` alone cannot prevent the corruption of security-critical fields in allocated objects. Additionally, `ISLAB`'s approach of keeping a direct mapping between slab objects and their freelist entries in segregating memory allows it to maintain their allocation status, which empowers `ISLAB` to prevent DF scenarios where the victim object lies at any location in the freelist (s.a., CVE-2017-2636 [79]). Providing this feature in `SLUB` would require keeping allocation status within objects, where attackers can corrupt and circumvent it. Yet, `ISLAB` cannot prevent the DF case where a victim object is first allocated and then freed by a second `free`. Moreover, `ISLAB` adopts the same IF hardening as `SLUB`, by checking that freed addresses belong to valid slabs. Finally, `ISLAB` alone does not isolate the segregated freelists, nor its slabs, leaving them exposed in the face of arbitrary read/write primitives.

6.3.2 `ILIST`. Similarly, `ILIST` (§5.1.2) protects linked list pointers, against memory corruption/disclosure, by storing them outside list objects. Although it still keeps a forward pointer to its list metadata entry within every list object, `ILIST` detects corruption by ensuring on every object access that the backward pointer, stored within the metadata entry, matches the accessed object. Additionally, to prevent attackers from crafting malicious metadata entries with valid bidirectional references (which is possible with the existing list integrity checks in Linux; see Section 4.1), `ILIST` ensures that each entry belongs to the dedicated list-metadata slab.

6.3.3 `ISLAB-ksmap-cred`. The previous variants cannot guarantee the integrity of an object's data, or that of memory management metadata, in the presence of stronger attackers that poses arbitrary read/write primitives. Existing solutions [80] can only partly mitigate such scenarios, since they merely protect the slabs that store senstive objects (s.a., page tables, process credentials, or keyring data). While this prevents *directly* tampering-with target objects, attackers can still abuse a slab's associated metadata. For example, as we demonstrate in Section 4.3, an attacker can corrupt the freelist pointer in the `struct page` of a slab for `struct cred` objects, and make it point to the privileged `struct cred` that is used by privileged (i.e., `root`) tasks [58]. Then, a subsequent `struct cred` allocation request will return the privileged (i.e., bogus) one, effectively allowing a (local) attacker to escalate their privileges.

This could also be achieved by applying the same technique against the other slab control structures, like kmem_cache_node and kmem_cache_cpu, all of which store freelist pointers to struct cred slabs. ISLAB-ksmap-cred (§5.2), which relies on our SMAP-based (selective) memory isolation framework (see Section 5.2), isolates both the physical pages of selected sensitive objects and their associated metadata. Additionally, since kSMAP stores interrupted task state on a per-task kSMAP-isolated stack, attackers cannot corrupt it from coercing tasks running in parallel on different CPU threads. Specifically, the SMAP state of a task is protected, preventing attackers from returning from an interrupt with SMAP illegitimately disabled, and general purpose registers are protected, preventing attackers from corrupting sensitive fields of protected objects (s.a., the uid field of a struct cred object).

Moreover, we protect the pointers to the isolated stack and to cred objects themselves, s.a., the ones stored in non-isolated task_struct and RCU objects, via bidirectional referencing, which prevents attackers from tampering with them. Similarly, critical pointers to the isolated struct cred objects are protected via bidirectional referencing. We also prevent attackers from using pointers to fake struct cred objects by checking that they belong to struct pages that are maintained by kSMAP. Thus, kSMAP effectively prevents all options for an attacker with arbitrary R/W primitives that target sensitive objects.

6.4 Real-world Exploits

In order to demonstrate the effectiveness of our defenses, we deployed three existing exploits against CVEs found in the Linux kernel [32, 34, 97], which target SLUB, the lists manager, and struct cred objects, respectively, and confirmed that ISLAB mitigates [34], ILIST mitigates [32], and ISLAB-ksmap-cred mitigates [97].

Additionally, we surveyed several other known exploits targeting the said subsystems in Linux kernel, and determined that five can be mitigated by ISLAB [54, 62, 70, 79, 101], five by ILIST [25, 31, 55, 66, 106], and five by ISLAB-ksmap-cred [7, 8, 21, 23, 74]. Moreover, ISLAB and ISLAB-ksmap-cred can also withstand our proof-of-concept exploits described in Section 4.3. First, attackers cannot corrupt ISLAB's freelist pointers or ILIST's linked-list pointers via vulnerable objects, since they lie segregated in shadow memory. Second, even if attackers gain arbitrary read/write primitives, ISLAB equipped with kSMAP prevents them from corrupting sensitive struct cred objects, including their associated out-of-slab metadata stored in struct page, effectively preventing their abuse for privilege escalation.

7 DISCUSSION AND FUTURE WORK

Similarly to ISLAB and ILIST, our techniques can be extended to harden other memory manipulation APIs that maintain corruptible in-object metadata, like the next pointer stored by struct msg_msg objects (i.e., a popular target for constructing an arbitrary read/write primitive) [6, 69, 95], red-black trees, hash tables, or reference counters, which are security-critical [60]. Linked-lists stored on the stack are currently handled in a crude manner, and are simply discarded by the kernel after use—ILIST has no way of knowing whether they are still in use or can be freed.

To lower the memory overhead of ILIST in this case, we propose patching the kernel code, either manually or automatically (e.g., via Coccinelle [57]), to cleanup temporary lists precisely. Additionally, in rare cases, legacy kernel subsystems do not use the standardized list-manipulation API and access list pointers directly, which could potentially lead to system crashes as they are segregated under ILIST. They have to either be adjusted manually or automatically (i.e., via means of compiler-assisted or source-code rewriting).

In this iteration we configured kSMAP to protect struct cred slabs and their metadata, however, it could also be leveraged to isolate other sensitive kernel data (s.a., page tables or modprobe_path). Moreover, although we currently instrument the kernel manually with domain switching routines to allow legitmate access to isolated data, kSMAP can be extended with automated selective data protection frameworks, s.a., DynPTA [72], to avoid manual labor. The length of kSMAP's non-isolated execution windows, which may include potentially unsafe references, could be shrunk down by the kernel compiler by clustering all accesses to isolated objects into an optimal number of unsafe-access windows (for performance), without including other types of references (for security). Moreover, instructions capable of manipulating the AC bit in RFLAGS should be further hardened via call gates in order to prevent attackers with code-execution primitives from abusing them to disable kSMAP. Additionally, since the pushf/popf instructions do not transfer the value of the *resume flag (RF)* into RFLAGS, and setting this flag manually is not facilitated by Intel, debugging the kernel with hardware breakpoints is incompatible with kSMAP; however, kernel debuggers can use software breakpoints (i.e., INT3) instead.

ISLAB equipped with kSMAP is generally compatible with other hardware extensions that have been proven effective in earlier memory isolation studies, s.a., VT-x [80], MPK/PKU [17, 73, 96], and CET [105]. Nevertheless, kernel support for MPK (namely PKS [19]) and CET shadow stacks are not yet rolled out. Additionally, ARM processors ship with PAN [67], whose functionality is equivalent to SMAP's, making kSMAP portable to ARM architectures. To be supported on architectures that use 57 bits for virtual addresses, ISLAB can use the remaining top 7 bits plus the 3 least significant bits to store indices for slab objects.

8 RELATED WORK

FreeGuard [89], GUARDER [90], and Shadow-Heap [11] protect object freelists by storing them in shadow memory, outside of the attackers' reach. They are able to detect DFs and IFs by keeping status information for each object. ISLAB employs similar techniques (in a broad sense). However, when considering designing the segregated freelists, none of these designs explicitly address the performance slowdowns induced by cache- and TLB-misses, inflicted by managing the segregated freelists in shadow memory. Shadow-Heap keeps the original metadata within objects, and checks their integrity by comparing them with the protected shadow copies on every object (de-)allocation. This severely impacts an application's performance, as also noted by its authors. Moreover, none of the aforementioned solutions protect the metadata of sensitive objects against stronger threat models, which involve arbitrary read/write primitives, and are not directly compatible with SLUB, which maintains a complex hierarchy of memory management structures (see Section 2.1).

In contrast, ISLAB leverages freelist maps for optimal system performance, and can guarantee the integrity of sensitive objects and their associated metadata via kSMAP. While older slab implementations came with segregated metadata [9, 10], they also incurred performance downsides. In the seminal work of Bonwick [9], object metadata are managed by bufctl data structures that contain mappings to the slab and the freelists. While for objects smaller than 512 bytes these are stored inside the objects themselves, for larger objects they are segregated and mapped with a hashtable. Nevertheless, ISLAB: (1) does not require a mapping from the object to metadata and (2) includes optimizations to reduce memory overhead. Magazines, as introduced by Bonwick and Adams [10], employ a stack data structure for freelists to facilitate faster (de-)allocations as well as per-cpu caches. In comparison, ISLAB employs a map structure and retains all scaling benefits of SLUB, including assigning dedicated slabs to CPUs, as well as a specific lock-free implementation and further memory consumption optimizations. We are not aware of any prior work that leverages the magazines as implemented by ILIST to design segregated linked lists.

PAL [109], PATTER [108], and Camouflage [24] protect code pointers in kernel objects via the Pointer Authentication (PA) extension of ARM CPUs, thus aiming to mitigate code-reuse attacks in the kernel setting [29, 40]. Using dedicated instructions, they keep a message authentication code (MAC) in the unused segment of a function pointer, by cryptographically signing the pointer with a secret key stored in a set of CPU-specific registers and a context that ensures integrity against rogue overwrites and replay attacks. However, the relatively small value space of the MAC (ranging from 11 to 31 bits) allows it to be feasibly bruteforced by attackers, which is a limitation acknowledged by the authors, and even demonstrated by the recent PACMAN [82] attack. In addition, to this date PA is only available on very few ARM processors, and x86 vendors have not expressed their intention to introduce similar extensions in Intel or AMD chips. In contrast, the pointer segregation techniques employed by ISLAB and ILIST do not depend on custom hardware extesions, while SMAP, leveraged by ISLAB-ksmap-cred, is present in both Intel and AMD chips, and it has an equivalent on ARM processors, called PAN. Moreover, there are currently no known flaws in the implementation of SMAP or PAN. Finally, all aforementioned solutions only protect function pointers in kernel objects, while leaving other pointer types, such as linked lists, exposed.

Several in-process hardware-based memory isolation techniques have also been proposed in the past years. xMP [80] proposed a selective memory isolation framework, for both kernel and user space, which leverages Intel VT-x to protect sensitive data, such as struct cred objects in the kernel or cryptographic material in userland. Nevertheless, xMP induces non-negligible performance overhead, and relies on the presence of hyper-privileged code. In contrast, kSMAP leverages lightweight hardware extensions employed by all modern architectures, and does not depend on any super-privileged mode. Moreover, as we highlighted in Section 4.2, xMP exhibits untackled security flaws that would allow attackers to undermine its isolation domains, especially in interrupt context. SEIMI [100, 104] isolates sensitive memory in user-mode applications via SMAP, emphasizing the extension's superior execution latency compared to VT-x and MPK.

Nevertheless, as SMAP is a privileged instruction, designed to be executed in Ring 0, SEIMI has to redesign the widely-accepted privileged execution hierarchy, moving userland in kernel-space and the kernel at the hypervisor level. This results in poor compatibility and portability, which, in turn, leads to high runtime overhead. In contrast, kSMAP is a novel memory isolation framework, built for isolating intra-kernel memory, which does not require significant kernel changes. Zhong et al. [111] have also studied using SMAP to isolate sensitive kernel data.

9 CONCLUSION

We introduced several kernel hardening extensions that alleviate the weaknesses of kernel memory managers in the presence of memory errors: (1) ISLAB enhances the security of kernel slab allocators; (2) ILIST hardens kernel linked lists; and (3) kSMAP provides a framework for selectively isolating kernel memory via SMAP. ISLAB and ILIST segregate memory management metadata of freelists and linked lists into dedicated memory regions, where they lie protected in the presence of memory safety vulnerabilities. Moreover, we equipped ISLAB with kSMAP and protected the struct cred subsystem against attackers that have arbitrary in-kernel memory read/write access, both when tasks are running and when they are preempted. Finally, we evaluated and analyzed how our hardening extensions reduce the attack surface of the Linux kernel over existing techniques.

AVAILABILITY

Our ISLAB prototype is available at: https://git.sec.in.tum.de/islab

ACKNOWLEDGMENTS

We thank our shepherd, Juanru Li, and the anonymous reviewers for their valuable feedback. This work was funded in part by the Bavarian Ministry of Science and Arts (STMWK), under the project "Security in everyday use of digital technologies (ForDaySec)," and the National Science Foundation (NSF) through awards CNS-2238467, CNS-2104148, and CNS-1749895. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, NSF, or STMWK.

REFERENCES

[1] 2021. perf(1) – Performance analysis tools for Linux. https://man7.org/linux/man-pages/man1/perf.1.html.
[2] Sam Ainsworth and Timothy M Jones. 2020. MarkUs: Drop-in Use-After-Free Prevention for Low-Level Languages. In *IEEE Symposium on Security and Privacy (S&P)*. 578–591.
[3] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security Symposium (SEC)*. 177–192.
[4] Alex Plaskett. 2021. CVE-2021-31956 Exploiting the Windows Kernel (NTFS with WNF) – Part 1. https://research.nccgroup.com/2021/07/15/cve-2021-31956-exploiting-the-windows-kernel-ntfs-with-wnf-part-1/.
[5] Alexander Popov. 2021. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel. https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.
[6] Awarau and pql. 2022. CVE-2022-29582 An io_uring vulnerability. https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/.
[7] Maher Azzouzi. 2021. CVE-2017-11176. https://github.com/MaherAzzouzi/LinuxKernelStudy/tree/main/CVE-2017-11176.
[8] David's Blog. 2022. How The Tables Have Turned: An analysis of two new Linux vulnerabilities in nf_tables. http://blog.dbouman.nl/2022/04/02/How-The-Tables-Have-Turned-CVE-2022-1015-1016/.
[9] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer Technical Conference*. 87–98.

[10] Jeff Bonwick and Jonathan Adams. 2001. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX Annual Technical Conference (ATC)*. 15–33.

[11] Johannes Bouché, Lukas Atkinson, and Martin Kappes. 2020. Shadow-Heap: Preventing Heap-based Memory Corruptions by Metadata Validation. In *European Interdisciplinary Cybersecurity Conference (EICC)*. 1–6.

[12] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel*. 294–350.

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (SEC)*. 249–266.

[14] Quan Chen, Ahmed M Azab, Guruprasad Ganesh, and Peng Ning. 2017. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*. 167–178.

[15] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. 2020. A Systematic Study of Elastic Objects in Kernel Exploitation. In *ACM Conference on Computer and Communications Security (CCS)*. 1165–1184.

[16] Yueqi Chen and Xinyu Xing. 2019. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *ACM Conference on Computer and Communications Security (CCS)*. 1707–1722.

[17] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P Kemerlis, and Nikos Vasilakis. 2023. BinWrap: Hybrid Protection against Native Node. js Add-ons. In *Proceedings of the ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*. 429–442.

[18] Jonathan Corbet. 2009. An updated guide to `debugfs`. https://lwn.net/Articles/334546/.

[19] Jonathan Corbet. 2020. Memory protection keys for the kernel. https://lwn.net/Articles/826554/.

[20] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *IEEE Symposium on Security and Privacy (S&P)*. 161–175.

[21] cutesmilee's blog. 2022. Exploiting CVE-2019-2215. https://cutesmilee.github.io/kernel/linux/android/2022/02/17/cve-2019-2215_writeup.html.

[22] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables.. In *Network and Distributed System Security Symposium (NDSS)*.

[23] Vincent Dehors. 2021. Exploitation of a double free vulnerability in Ubuntu `shiftfs` driver (CVE-2021-3492). https://www.synacktiv.com/publications/exploitation-of-a-double-free-vulnerability-in-ubuntu-shiftfs-driver-cve-2021-3492.html.

[24] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. 2020. Camouflage: Hardware-assisted CFI for the ARM Linux kernel. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[25] Di Shen. 2017. The Art of Exploiting Unconventional Use-after-free Bugs in Android Kernel. https://pacsec.jp/psj17/PSJ2017_DiShen_Pacsec_FINAL.pdf.

[26] Apple Security Engineering and Architecture (SEAR). 2022. Towards the next generation of XNU memory safety: `kalloc_type`. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/.

[27] Jason Evans. 2011. Scalable Memory Allocation Using `jemalloc`. https://engineering.fb.com/2011/01/03/core-infra/scalable-memory-allocation-using-jemalloc/.

[28] Yi Feng and Emery D Berger. 2005. A Locality-Improving Dynamic Memory Allocator. In *Workshop on Memory System Performance (MSPC)*. 68–77.

[29] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[30] Sanjay Ghemawat and Paul Menage. 2007. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[31] Google Project Zero. 2019. Bad Binder: Android In-The-Wild Exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[32] Google Project Zero. 2021. CVE-2021-22555: Turning \x00\x00 into 10000$. https://github.com/google/security-research/blob/master/pocs/linux/cve-2021-22555/writeup.md.

[33] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. 2021. IskiOS: Intra-kernel Isolation and Security using Memory Protection Keys. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[34] GRIMM. 2021. New Old Bugs in the Linux Kernel. https://blog.grimm-co.com/2021/03/new-old-bugs-in-linux-kernel.html.

[35] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems (ESSoS)*. 161–176.

[36] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX Annual Technical Conference (ATC)*. 489–504.

[37] Ian Beer. 2019. In-the-wild iOS Exploit Chain 1. https://googleprojectzero.blogspot.com/2019/08/in-wild-ios-exploit-chain-1.html.

[38] Intel. 2023. Intel® 64 and IA-32 Architectures Software Developer's Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[39] Jann Horn. 2021. How a simple Linux kernel memory corruption bug can lead to complete system compromise. https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html.

[40] Di Jin, Vaggelis Atlidakis, and Vasileios P Kemerlis. 2023. EPF: Evil Packet Filter. In *USENIX Annual Technical Conference (ATC)*. 735–751.

[41] Kees Cook. 2017. mm: Add SLUB free list pointer obfuscation. https://patchwork.kernel.org/patch/9864165/.

[42] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium (SEC)*. 957–972.

[43] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *USENIX Security Symposium (SEC)*. 459–474.

[44] The Linux Kernel. 2023. Page Table Isolation (PTI). https://www.kernel.org/doc/html/next/x86/pti.html.

[45] The Linux Kernel. 2023. *Physical Memory Model*.

[46] The Linux Kernel. 2023. What is RCU? – "Read, Copy, Update". https://www.kernel.org/doc/html/next/RCU/whatisRCU.html.

[47] kileak. 2021. VULNCON CTF 2021 – IPS. https://kileak.github.io/ctf/2021/vulncon-ips/.

[48] Thomas J Killian. 1984. Processes as Files. In *USENIX Summer Technical Conference*. 203–207.

[49] Kenneth C Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* 8, 10 (1965), 623–624.

[50] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*. 1–19.

[51] Greg Kroah-Hartman. 2003. `udev` – A Userspace Implementation of `devfs`. In *Ottawa Linux Symposium (OLS)*. 263–271.

[52] Christopher Kruegel, William Robertson, and Giovanni Vigna. 2004. Detecting Kernel-level Rootkits through Binary Analysis. In *Annual Computer Security Applications Conference (ACSAC)*. 91–100.

[53] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 147–163.

[54] kylebot's Blog. 2022. [CVE-2022-1786] A Journey To The Dawn. https://blog.kylebot.net/2022/10/16/CVE-2022-1786/.

[55] Lam Jun Rong. 2022. `io_uring` – new code, new bugs, and a new exploit technique. https://www.starlabs.sg/blog/2022/06-io_uring-new-code-new-bugs-and-a-new-exploit-technique/.

[56] Christoph Lameter. 2014. Slab Allocators in the Linux Kernel: `SLAB`, `SLOB`, `SLUB`. In *Open Source Summit (LinuxCon)*.

[57] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *USENIX Annual Technical Conference (ATC)*. 601–614.

[58] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *ACM Conference on Computer and Communications Security (CCS)*. 1963–1976.

[59] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (SEC)*. 973–990.

[60] Jian Liu, Lin Yi, Weiteng Chen, Chengyu Song, Zhiyun Qian, and Qiuping Yi. 2022. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *USENIX Security Symposium (SEC)*. 125–142.

[61] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *ACM Conference on Computer and Communications Security (CCS)*. 1607–1619.

[62] Maxime Peterlin, Philip Pettersson, Alexandre Adamski, and Alex Radocea. 2020. Exploiting a Single Instruction Race Condition in Binder. https://www.longterm.io/cve-2020-0423.html.

[63] Larry W McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference (ATC)*. 279–294.

[64] Alfred J Menezes, Paul C van Oorschot, and Scott A Vanstone. 2018. *Handbook of Applied Cryptography*. CRC press.

[65] Otto Moerbeek. 2009. A new `malloc(3)` for OpenBSD. https://www.openbsd.org/papers/eurobsdcon2009/otto-malloc.pdf. In *EuroBSDCon*.

[66] Arthur Mongodin. 2022. [CVE-2022-34918] A crack in the Linux firewall. https://www.randorisec.fr/crack-linux-firewall/.

[67] James Morse. 2015. arm64: kernel: Add support for Privileged Access Never. https://lwn.net/Articles/651614/.

[68] Andy Nguyen. 2020. BleedingTooth: Linux Bluetooth Zero-Click Remote Code Execution. https://google.github.io/security-research/pocs/linux/bleedingtooth/writeup.html.

[69] Nick Gregory. 2022. The Discovery and Exploitation of CVE-2022-25636. https://nickgregory.me/post/2022/03/12/cve-2022-25636/.

[70] Vitaly Nikolenko. 2016. CVE-2016-6187: Exploiting Linux kernel heap off-by-one. https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit.

[71] Gene Novark and Emery Berger. 2010. DieHarder: Securing the Heap. In *ACM Conference on Computer and Communications Security (CCS)*. 573–584.

[72] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. 2021. DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection. In *IEEE Symposium on Security and Privacy (S&P)*. 1919–1937.

[73] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference (ATC)*. 241–254.

[74] Manfred Paul. 2020. CVE-2020-8835: Linux Kernel Privilege Escalation via Improper eBPF Program Verification. https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification.

[75] Enrico Perla and Massimiliano Oldani. 2010. *A Guide To Kernel Exploitation: Attacking the Core*. 47–99.

[76] Phantasmal Phantasmagoria. 2005. The Malloc Maleficarum. https://seclists.org/bugtraq/2005/Oct/118.

[77] Phoronix Test Suite. [n. d.]. Open-Source Automated Benchmarking. https://www.phoronix-test-suite.com.

[78] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. kRˆ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *European Conference on Computer Systems (EuroSys)*. 420–436.

[79] Alexander Popov. 2017. Race for Root: The Analysis of the Linux Kernel Race Condition Exploit. https://media.ccc.de/v/SHA2017-295-race_for_root_the_analysis_of_the_linux_kernel_race_condition_exploit.

[80] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*. 563–577.

[81] Weizhong Qiang, Yong Cao, Weiqi Dai, Deqing Zou, Hai Jin, and Benxi Liu. 2017. Libsec: A Hardware Virtualization-based Isolation for Shared Library. In *IEEE International Conference on High Performance Computing and Communications (HPCC); IEEE International Conference on Smart City (SmartCity); IEEE International Conference on Data Science and Systems (DSS)*. 34–41.

[82] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *International Symposium on Computer Architecture (ISCA)*. 685–698.

[83] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, Jonathan M Smith, Andre DeHon, et al. 2021. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 296–311.

[84] Dan Rosenberg. 2011. A Heap of Trouble: Exploiting the Linux Kernel SLOB Allocator. http://vulnfactory.org/research/slob.pdf.

[85] Satoshi's notes. 2023. Intel VT-rp – Part 1. remapping attack and HLAT. https://tandasat.github.io/blog/2023/07/05/intel-vt-rp-part-1.html.

[86] SecWiki. 2023. Linux Kernel Exploits. https://github.com/SecWiki/linux-kernel-exploits.

[87] SecWiki. 2023. Windows Kernel Exploits. https://github.com/SecWiki/windows-kernel-exploits.

[88] Shellphish. 2023. Educational Heap Exploitation. https://github.com/shellphish/how2heap.

[89] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *ACM Conference on Computer and Communications Security (CCS)*. 2389–2403.

[90] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: A Tunable Secure Allocator. In *USENIX Security Symposium (SEC)*. 117–133.

[91] sqrkkyu and twzi. 2007. Attacking the Core: Kernel Exploiting Notes. *Phrack* (2007).

[92] `jemalloc`. 2023. memory allocator. https://jemalloc.net.

[93] The Linux Kernel. [n. d.]. Kernel stacks on x86-64 bit. https://www.kernel.org/doc/Documentation/x86/kernel-stacks.

[94] The Linux Kernel. 2023. `percpu` *memory allocator*.

[95] Theori BLOG. [n. d.]. Linux Kernel Exploit (CVE-2022-32250) with `mqueue`. https://blog.theori.io/linux-kernel-exploit-cve-2022-32250-with-mqueue-a8468f32aab5.

[96] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium (SEC)*. 1221–1238.

[97] Valentina Palmiotti. 2022. Put an `io_uring` on it: Exploiting the Linux Kernel. https://chompie.rip/Blog+Posts/Put+an+io_uring+on+it+-+Exploiting+the+Linux+Kernel.

[98] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing. In *European Conference on Computer Systems (EuroSys)*. 266–282.

[99] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. 2009. Countering Kernel Rootkits with Lightweight Hook Protection. In *ACM Conference on Computer and Communications Security (CCS)*. 545–554.

[100] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *IEEE Symposium on Security and Privacy (S&P)*. 592–607.

[101] Wang, Yong. 2019. From Zero to Root: Building Universal Android Rooting with a Type Confusion Vulnerability. In *Zer0Con*.

[102] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *USENIX Security Symposium (SEC)*. 2453–2470.

[103] Wolfram Gloger. 2006. `ptmalloc`. http://www.malloc.de/en/.

[104] Chenggang Wu, Mengyao Xie, Zhe Wang, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, Min Yang, and Tao Li. 2023. Dancing With Wolves: An Intra-Process Isolation Technique With Privileged Hardware. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 20, 3 (2023), 1959–1978.

[105] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2022. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-process Memory Isolation. In *ACM Conference on Computer and Communications Security (CCS)*. 2989–3002.

[106] Xingyu Jin and Richard Neal. 2021. The Art of Exploiting UAF by Ret2bpf in Android Kernel. In *Black Hat Europe (BHEU)*.

[107] Wenjie Xiong and Jakub Szefer. 2021. Survey of Transient Execution Attacks and their Mitigations. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–36.

[108] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. 2019. ARM Pointer Authentication based Forward-Edge and Backward-Edge Control Flow Integrity for Kernels. *arXiv preprint arXiv:1912.10666* (2019).

[109] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *USENIX Security Symposium (SEC)*. 89–106.

[110] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2022. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *USENIX Security Symposium (SEC)*. 71–88.

[111] Bingnan Zhong and Qingkai Zeng. 2021. SecPT: Providing Efficient Page Table Protection based on SMAP Feature in an Untrusted Commodity Kernel. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 215–223.

[112] Saman Zonouz, Mingbo Zhang, Pengfei Sun, Luis Garcia, and Xiruo Liu. 2018. Dynamic Memory Protection via Intel SGX-Supported Heap Allocation. In *IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC); IEEE International Conference on Pervasive Intelligence and Computing (PICom); IEEE International Conference on Big Data Intelligence and Computing (DataCom); IEEE International Conference on Cyber Science and Technology Congress (CyberSciTech)*. 608–617.