Fence2Pwn: KFENCE-Enabled Kernel Exploitation Bypassing Slab Hardening and Memory Tagging

Ernesto Martínez García
Graz University of Technology

Lukas Maar
Graz University of Technology

Martin Unterguggenberger
Graz University of Technology

Stefan Mangard
Graz University of Technology

Abstract

While the Linux kernel remains a prime target due to its privileged execution model, exploitation has become substantially more difficult in recent years. Kernel hardening efforts have increasingly focused on the slab allocator, aiming to mitigate entire vulnerability classes (e.g., via memory tagging) or disrupt exploit techniques (e.g., via heap object segregation). However, it remains unclear whether these protections are consistently enforced across all allocation paths.

In this paper, we find that one allocation path is excluded from all state-of-the-art slab defenses: the path used for KFENCE allocations. KFENCE is designed as a low-overhead sampling-based memory-safety error detector for production kernels. It is enabled by default and active in billions of systems including on Android, Red Hat Enterprise Linux, and most Linux distributions. We show that it unintentionally enables a new exploit technique, Fence2Pwn. Concretely, Fence2Pwn leverages several KFENCE-specific behaviors. Three are particularly notable: (i) using a timing side channel, we detect when the kernel falls back to KFENCE allocations; (ii) KFENCE allocations are served by KFENCE-managed caches rather than size- and type-segregated slab caches, enabling bypass of heap segregation; and (iii) KFENCE allocations are untagged, enabling bypass of memory tagging. We evaluate Fence2Pwn by profiling our timing side channel, KFENCE's object slot dynamics as well as different environments and noise floors. Finally, we demonstrate that Fence2Pwn exploits this slab-defense gap: Fence2Pwn uses KFENCE to exploit an existing UAF-based memory-reuse vulnerability, which allows it to overlay security-relevant objects of different types and sizes.

1 Introduction

The Linux operating system kernel is highly susceptible to software vulnerabilities that are generally categorized as memory-safety errors. Such issues, including Out-Of-Bounds (OOB) and Use-After-Free (UAF) bugs, enable an adversary to corrupt security-relevant resources in kernel memory. The kernel is a lucrative target for threat actors, where a single kernel vulnerability can enable their primary objective: *escalation of privilege*. In particular, an exploitable memory-corruption primitive can lead to full system compromise, for instance by mutating sensitive kernel data (e.g., credentials).

To achieve privilege escalation, prior work presents kernel exploitation techniques [24, 29, 31, 35, 38, 47, 53, 57] that can be coarsely classified into two attack paths: control-flow hijacking and data-oriented attacks. In both cases, the attacker exploits the kernel interface to trigger initial memory corruption, e.g., via an OOB or UAF error. In control-data attacks, an adversary typically corrupts a function pointer, which then serves as a primitive to hijack control flow for a code-reuse attack. Alternatively, the initial vulnerability is used to manipulate a data pointer, enabling an arbitrary read/write primitive and subsequent modification of sensitive data.

To limit these kernel exploitation techniques, multiple hardening measures have been incorporated into the Linux kernel. For instance, SMEP/SMAP on x86-64 [11] and PXN/PAX on ARM [40] raise the bar for control-data attacks such as ret2usr and pivot2usr [24]. Moreover, to limit more sophisticated code-reuse attacks, kernel control-flow integrity measures [39] enforce control flow to a predefined call graph. Beyond mitigating control-data-based techniques, recent hardening efforts aim to prevent attackers from progressing past the initial phase of exploitation by isolating vulnerable objects. In particular, the slab allocator—the kernel heap allocator—implements heap object segregation for high-value objects, i.e., privilege-escalation-relevant data structures. As a result, common heap vulnerabilities that allow memory corruption cannot directly overwrite these structures due to size- and type-based segregation of allocator caches.

Furthermore, memory tagging has recently been added to the slab allocator on top of these hardening techniques. Specifically, the Kernel Address Sanitizer (KASAN) [51] is a configurable kernel feature for memory sanitization that supports different modes, i.e., generic KASAN and software tag-based KASAN, as well as hardware tag-based KASAN

that utilizes the ARM Memory Tagging Extension (MTE) [5], to detect common heap vulnerabilities. This means that a kernel hardened with memory tagging is significantly more difficult for an adversary to exploit in practice, since initial memory corruption is detected and the attack is disrupted.

Taken together, these defenses significantly strengthen the slab allocator and raise the bar for heap-based exploitation. This raises the question of whether the slab allocator is the only allocation path or whether a less-protected path exists.

In this work, we show that a substantially less protected alternative allocation path exists and present Fence2Pwn, which leverages this path to bypass state-of-the-art slab defenses, including heap object segregation and memory tagging. Fence2Pwn is a kernel exploitation technique that abuses Kernel Electric Fence (KFENCE) [25], a memory-safety error detector enabled by default in production kernels. Although KFENCE is intended as a debugging tool to increase security by discovering bugs, we show that it provides an allocation path that bypasses these defenses and thus undermines kernel security. In particular, we identify several weaknesses in KFENCE's allocation behavior that enable Fence2Pwn: *First*, its time-consuming sampling behavior allows us to identify KFENCE allocations through a timing allocator side channel. *Second*, we find that KFENCE allocations are served by KFENCE-managed caches that effectively bypass the size- and type-based segregation of slab caches. *Third*, we uncover that even on hardened kernels that leverage memory tagging (i.e., hardware tag-based KASAN with ARM MTE), KFENCE allocations remain untagged.

Consequently, we demonstrate that Fence2Pwn enables cross-cache-style reuse by reclaiming a freed UAF slot as a security-relevant object from a different allocator cache, bringing sensitive kernel data structures (e.g., credentials) back within reach for an adversary. Classic cross-cache reuse relies on page-allocator memory reuse and is mitigated by `SLAB_VIRTUAL` [22,48] and greatly limited by memory tagging such as ARM MTE [5]. In strong contrast, Fence2Pwn bypasses these defenses by moving the relevant allocations onto the KFENCE path. We present a systematic security analysis of how KFENCE-served allocations bypass broadly utilized slab defenses, e.g., heap object segregation and KASAN-based sanitization. Specifically, we show that our attack methodology is able to bypass software-based KASAN. In addition, we detail how Fence2Pwn is able to circumvent KASAN with ARM MTE memory tagging, which is actively used for hardening production kernels.

Moreover, we provide a detailed evaluation of Fence2Pwn's slab bypass, including the timing side channel and KFENCE's object slot dynamics. We evaluate across different environments—a hardened Linux kernel, Fedora 43 Workstation, and Android 16—and under realistic noise floors introduced via the Phoronix Test Suite [43], showing that Fence2Pwn remains applicable in the evaluated scenarios. To demonstrate real-world applicability, we show how Fence2Pwn can be weaponized with CVE-2023-52926 to induce controlled object overlap and metadata corruption. Furthermore, we present a study on affected Linux distributions, highlighting the deployment impact of this slab-defense gap. Here we find that a large number of Linux distributions, including Android 16 GKI and Red Hat Enterprise 10, have KFENCE enabled by default and are thus affected by Fence2Pwn.

Lastly, we outline the security implications of our findings and discuss potential mitigations. As an immediate mitigation, we recommend disabling KFENCE for critical systems exposed to kernel exploitation. For future mitigations we conclude that finding a sustainable, long-term mitigation of Fence2Pwn is challenging, i.e., addressing all uncovered weaknesses without disrupting its intended behavior.

Contributions. The main contributions of this work are:

(1) KFENCE Weaknesses: We analyze KFENCE's security and identify 5 weaknesses that expose a slab-defense gap in production kernels.

(2) Fence2Pwn: We introduce a novel KFENCE-enabled kernel exploitation technique that leverages this gap to bypass slab allocator hardening techniques, including heap object segregation and memory tagging.

(3) Attack Evaluation: We demonstrate UAF-based exploitation, showcasing that Fence2Pwn is able to overlay security-relevant objects of different object types and sizes through a memory-reuse attack.

Outline. The remaining paper is structured as follows: Section 2 provides the background. Section 3 presents the study's workflow. Section 4 analyzes the security of KFENCE. Section 5 presents Fence2Pwn, and Section 6 evaluates it. Section 7 discusses our results. Section 8 concludes this work.

Responsible Disclosure. We reported our findings to the Linux Kernel Security Team and Google in December 2025. The Linux Kernel Security Team acknowledged our findings and led mitigation work addressing some weaknesses described in Section 4, with patches merged upstream and into the Android common branch: `09833d99` [1, 2] disabling KFENCE when kernel memory tagging is active, addressing **(W3)**; `da735962` [36, 37] specialized opt-in fault handling toggle for KFENCE, addressing **(W5)**; `870ff192` [44, 45] randomizing the KFENCE freelist order, as a preventive mitigation addressing Section 7 Future Work.

2 Background & Related Work

This section provides the background, covering general kernel exploitation concepts. In addition, we introduce the Kernel Address Sanitizer (KASAN) and the Kernel Electric Fence (KFENCE) Linux allocator mechanism.

2.1 Kernel Exploitation

Kernel exploitation typically proceeds in three stages: establishing a read primitive, establishing a write primitive, and finally triggering a kernel event. Exploit techniques first transform a vulnerability into a read primitive, which is used to locate security-relevant target objects and thereby defeat Kernel Address Space Layout Randomization (KASLR) [14]. Next, techniques obtain a write primitive that manipulates fields of these target objects. Finally, an event is triggered that uses the manipulated fields, leading to control-flow hijacking or data-oriented corruption and compromising the system.

While control-flow hijacking aims to gain arbitrary code execution in the kernel, data-oriented attacks aim to obtain an arbitrary read/write primitive for privilege escalation. Control-flow hijacking has been constrained by a growing set of hardware [11, 40] and software [39, 50] mitigations. This makes hijacking attacks increasingly difficult and has contributed to the rise of innovative and novel data-oriented attacks [20, 29, 31, 32, 35, 42, 53, 54].

2.2 Kernel Allocators

The Linux kernel primarily allocates dynamic memory via two allocators: the *page allocator* and the *slab allocator*. The page allocator manages physical pages and serves coarse-grained allocations. It forms the basis of page-ordered allocations. The slab allocator sits on top of the page allocator and provides fine-grained object allocations by organizing memory (called slabs) into allocator caches of fixed-size objects. This design improves performance but also shapes the attack surface by constraining which objects can be colocated or reused. Accordingly, heap-based attacks typically target allocator behavior to steer placement and reuse.

Many heap-based exploits rely on reclaiming a freed object slot with attacker-controlled data. This can be done *in-cache*, where the vulnerable and victim objects come from the same slab cache and thus share size and type constraints. The other option is *cross-cache*, where allocations of different sizes or types are made to land in the same slot despite allocator segregation, by steering reuse across the page allocator. In-cache techniques are generally more reliable [21, 35, 55] when the target object shares a cache with attacker-controllable data, while cross-cache techniques are needed to reach high-value targets that are segregated into different caches. Modern allocator-based defenses [10, 12, 22, 34] also reshape allocator behavior, primarily by segregating kernel heap objects at different granularities. For instance, `RANDOM_KMALLOC_CACHES` [12] introduces boot-time randomized slab caches based on the allocation site, while `SLAB_BUCKETS` [10] separates user-size-controlled objects. While these segregation-based mitigations make in-cache reuses more difficult as fewer objects share the same slab cache, they can make cross-cache reuses more reliable, as less uncontrolled allocation noise is per cache. To address this, `SLAB_VIRTUAL` [22, 48] deterministically prevents cross-cache reuses.

2.3 ARM Memory Tagging Extension

The ARM Memory Tagging Extension (MTE) [5], introduced for the ARMv8.5-A architecture, enhances processors with logical integrity checks that can be used as a production memory safety detector or a pre-release sanitizer/debugger.

The ARM MTE concept relies on meta-information, called memory tags, assigned to memory locations with a tag granularity of 16 B. Furthermore, ARM's Top Byte Ignore (TBI) [5] feature allows metadata to be encoded in the non-canonical address region of pointers. More concretely, ARM MTE leverages 4-bit memory tags to enforce access checks (i.e., logical integrity) for memory interactions. This means that for memory operations, the tag encoded in the pointer is compared with the tag associated with the memory granule. A tag match is considered benign program behavior, whereas a mismatch indicates a possible memory safety violation.

Depending on the mode in use (synchronous, asynchronous, or asymmetric [3, 5]), this violation is reported immediately (or delayed until the nearest kernel entry) to the operating system that handles the error, e.g., by forcing the program to terminate. ARM's memory tagging technology has recently seen adoption for runtime memory safety in practice, for example, in Android devices [3] and Apple SoCs [4].

2.4 Kernel Address Sanitizer

KASAN is a runtime memory-safety error detector, initially intended for debugging due to its high memory and runtime overhead. Upstreamed in 2015 for Linux `4.0` [52], KASAN introduced a high-cost high-confidence memory-safety error detector.

KASAN offers three modes of operation: (1) Generic mode; (2) Software tag-based mode; (3) Hardware tag-based mode. Generic KASAN is supported on most architectures, including `x86-64` and `arm64`. Tag-based KASAN modes are only available on supported hardware.

(1) Generic Mode. Keeps track of memory allocations via shadow memory, similar to userspace address sanitizers. Solely intended for debugging purposes. Memory accesses are checked with compile-time instrumentation against the shadow memory to ensure the access is valid. Supports the deterministic verification of: SLAB, SLUB, `page_alloc`, *vmap*, *vmalloc*, *stack*, and global memory.

(2) Software Tag-based Mode. Utilizes the TBI feature of `arm64` CPUs to implement software memory tagging. Intended for internal testing builds. Allocations store a random tag (value) on the TBI part of an address, bounding the tag to the allocated memory area. Memory accesses are checked with compile-time instrumentation, ensuring the address and

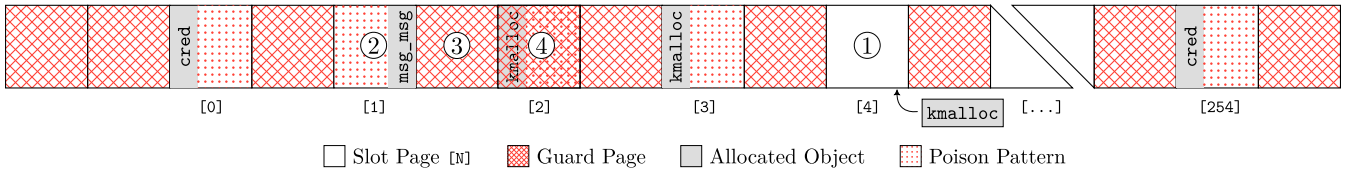


Figure 1: High-level overview of the KFENCE pool internals. On object allocation ①, slots are taken from the head of the freelist. Objects are placed at the start or end of a memory page ②, with the rest of the page filled with poison values. All surrounding pages are guard pages ③ that trap on access. Freed objects have their pages protected ④ until re-allocated, and freed objects are placed at the tail of the freelist (FIFO). Poison patterns are only checked during object free.

the accessed memory have matching tags. Supports the probabilistic verification of: SLAB, SLUB, `page_alloc`, *vmalloc*, and stack memory.

(3) Hardware Tag-based Mode. Uses ARM's MTE to implement a hardware-accelerated tag-based KASAN mode. Intended for production usage. Similar mode of operation as software tag-based mode, with hardware acceleration for *tagging* and performing *tag* checks. Supports the probabilistic verification of: SLUB, `page_alloc`, and non-executable *vmalloc* memory.

2.5 Kernel Electric Fence

KFENCE is a low-overhead sampling-based memory-safety error detector for the Linux kernel designed for production usage. Originally developed by Google as part of the Linux Kernel Sanitizers project [18] in 2020. Upstream Linux `5.12` merged KFENCE in 2021 as a runtime memory validator for production builds [52]. KFENCE detects memory corruptions on kernel-allocated memory, such as OOB accesses, UAF errors, and invalid-frees with low probability and overhead. The primary motivation for KFENCE is its suitability for deployment in production kernels due to its low overhead. Given sufficient uptime and a large user base, KFENCE can expose memory corruption bugs on *hard-to-reach* code paths that are unlikely to be reached in testing or fuzzing environments. As a result, KFENCE has been incorporated into several production-grade kernel configurations, including Android 12 and later, Red Hat Enterprise Linux (RHEL), and Oracle Linux, among others (see Table 3).

Usage. KFENCE integrates into the primary allocation path of the kernel on the `slab_alloc_node` function, which underlies both `kmalloc` and `kmem_cache_alloc`. For each allocation request, the allocator first attempts to service the request through KFENCE via `kfence_alloc`. KFENCE, via its own sampling mechanism, decides if the allocation should be served by KFENCE, in which case the object is returned and the function is short-circuited to return the allocated object. If the allocation is not served by KFENCE, it then continues to the default slab allocator codepath through `__slab_alloc_node`. Due to the conservative sampling-based strategy of KFENCE, the vast majority of allocations continue to be served by slab. The sampling strategy of KFENCE is implemented as a gate timer. Every 100 milliseconds, a delayed work function sets the global integer `kfence_enabled` to a configurable negative value of `kfence_burst`. Following `kfence_burst` number of allocations are considered to be served by KFENCE instead of slab, if requirements are met. In the default configuration, every 100 milliseconds, KFENCE serves *one* allocation globally.

Memory Pool. All allocations served by KFENCE reside in a fixed-size memory region, referred to as the KFENCE pool and referenced by `__kfence_pool`. The pool is allocated early during memory subsystem initialization via `memblock_alloc` in `mm_core_init`, or, if KFENCE is enabled at runtime, via `alloc_pages_exact`. The pool consists of contiguous pages that are organized into two distinct types: object pages and guard pages. Object pages are accessible and contain a single allocated object, placed either at the beginning or the end of the page, while the remaining page content is filled with a canary pattern. Guard pages are protected and trigger a fault upon any access. Each object page is always surrounded by guard pages, enforcing spatial isolation between objects. Figure 1 visualizes the KFENCE pool layout.

Object Allocation. On a KFENCE allocation, KFENCE first checks if the `kfence_allocation_gate` is open (i.e., positive). Then, it performs the following checks in order: (1) Allocation size must be less or equal than `PAGE_SIZE`. (2) Allocation must come from the default zone (i.e., no DMA). (3) Slab cache must not have `skip_kfence` enabled. (4) No similar long-lived allocation is already served.[1] (5) KFENCE pool has enough capacity. If all requirements pass, KFENCE selects the next free pool slot from the *freelist* and moves to the object allocation stage. First, it unprotects the page, marking the Page Table Entry (PTE) as present. Second, it decides the placement of the object: either at the start or at the end of the page, at random. Third, it initializes the entire page contents with a KFENCE canary value, for checking on *free*. Finally, initializes memory or calls SLAB object constructors if needed, and returns the object.

Object Deallocation. On deallocation, KFENCE-served objects go through additional checks. First, KFENCE en-

[1] Only applies in case there's memory pressure on the KFENCE pool

Figure 2: The high-level workflow comprising of four main components.

sures the passed address matches the pool slot object metadata, catching potential *double-free* vulnerabilities. Second, it marks the object metadata as freed, checking the corresponding canary values of the object page. Third, it protects the page by marking the PTE as not present, catching potential future *use-after-free* vulnerabilities. Finally, it appends the pool object slot to the end of the KFENCE freelist, causing the freelist to maintain a FIFO access pattern. KFENCE's design keeps *free* objects as long as possible in the freelist, as *free*'d objects are protected, thus increasing the probability of detecting double-free or use-after-free vulnerabilities.

Pressure Regime. To prevent frequent allocations from hoarding the KFENCE memory pool, KFENCE enters a special *pressure regime* when the pool utilization is above 75%. Here, KFENCE stops serving allocations originating from call stacks that already have an active KFENCE-served object.

3 High-Level Overview

Our work has the following components as shown in Figure 2.

First, we show that although most kernel heap allocations go through the slab allocator's hardened paths, a distinct allocation path exists that is effectively outside these protections. This path is used by KFENCE [25], a low-overhead heap-safety mechanism intended for production bug detection, and it is enabled on many production platforms, e.g., Red Hat Enterprise Linux, Android GKI, Fedora Linux, and Arch Linux. Enabling KFENCE creates a slab-defense gap, which we analyze in-depth (see Section 4). We identify five security-relevant weaknesses **(W1-W5)** that arise when KFENCE is enabled. At a high level, these weaknesses fall into two categories: (i) a fundamental timing difference between slab and KFENCE allocation paths, which acts as a side channel to distinguish them, and (ii) the absence of hardening strategies on KFENCE-served allocations, leaving such objects outside the scope of defenses that normally protect slab allocations.

Second, we introduce Fence2Pwn, which exploits the security weaknesses identified above. Fence2Pwn is a KFENCE-enabled exploitation technique that bypasses state-of-the-art slab-hardening defenses (see Section 5). It first leverages the identified timing side channel (i) in kernel allocations to detect when the Linux kernel falls back to KFENCE, rather than the slab allocator, for heap object allocations. While timing is not reliably observable in all allocation contexts [28, 35], Fence2Pwn additionally leverages KFENCE's periodic allocation behavior. It provides an independent and complementary signal for identifying KFENCE-served allocations. Combining both signals allows Fence2Pwn to determine whether a given object allocation is served by KFENCE. Once an allocation is detected to be KFENCE-served, and therefore outside the slab's hardenings (ii), Fence2Pwn proceeds with exploitation under conditions where slab hardening is inactive.

Third, we show Fence2Pwn's effectiveness by performing a memory-reuse attack (see Section 6). Specifically, we show that Fence2Pwn enables reusing the freed memory slot of a vulnerable UAF object as a security-relevant object from a different cache. This attack closely resembles the capabilities of slab-based cross-cache reuses [55], which exploits the memory reuse of the page allocator. Since cross-cache reuse is a very powerful technique and has been used in countless kernel attacks [6, 15, 20, 21, 23, 30–32, 35, 53, 55], it has been mitigated by the software-based defense `SLAB_VIRTUAL` [48] and greatly limited by the hardware-based scheme memory tagging [5]. In effect, Fence2Pwn re-enables this strong cross-cache reuse capability by moving the relevant allocations onto the KFENCE path. Finally, we evaluate Fence2Pwn across kernel versions, consumer devices, CPU architectures, and system stress conditions. The results demonstrate that Fence2Pwn is applicable across the evaluated target environments.

Fourth, we discuss the broader implications of Fence2Pwn (see Section 7). We begin by analyzing its security impact. Because Fence2Pwn re-enables the core capability of slab-based cross-cache reuse [55], it can support exploit strategies that rely on this reuse primitive. For example, DirtyCred [31] leverages cross-cache reuse to pivot a generic invalid-free into a double-free of a credential, escalating privileges. As another example, exploiting CVE-2022-29582 [6] from a `filp` UAF under modern defenses would require multiple cross-cache reuses: first from the `filp` cache to `kmalloc-cg-512` (for `msg_msgseg`), and then to `kmalloc-rnd*-512` (for `tls_context`). Moreover, we identify the affected deployments which has KFENCE activated, including hardened platforms such as Android and Red Hat Enterprise Linux. Finally, we discuss mitigations: as an immediate measure, disabling KFENCE closes the gap exploited by Fence2Pwn,

while longer-term approaches are currently under discussion and development by the Linux kernel team.

Threat Model. We assume an adversary with native-code execution in an unprivileged userspace process on the target system. This corresponds to the standard untrusted userspace domain and aligns with prior work on kernel exploitation and heap manipulation [29, 31, 35, 38, 47, 53, 57]. Further, we assume the adversary has a known UAF vulnerability in the kernel heap allocator.

We consider a target running a hardened Linux kernel. In particular, we assume state-of-the-art mitigations are enabled, including KASLR [14], SMEP/SMAP on x86-64 [11] or PXN/PAX on ARM [40], kernel control-flow integrity [39], slab freelist randomization [16], slab freelist hardening [8], slab bucket segregation [10], randomized `kmalloc` caches [12], and kernel-heap virtualization [48]. Collectively, these defenses raise the bar for heap shaping and reuse attacks by restricting cross-cache reuse [48] and tightening in-cache reuse through segregation [10, 12]. We additionally assume that memory-safety mechanisms are present to reduce the memory-corruption exploitability, either via software instrumentation (e.g., generic KASAN [51]) or hardware memory tagging (e.g., hardware tag-based KASAN with ARM MTE [5]). Our attack does not rely on microarchitectural vulnerabilities [19, 27], hardware side channels [33, 56], or hardware fault injection [49]. Crucially, we assume that the target has KFENCE enabled [25], as per common and default kernel configurations (see Table 3).

Observability. During Fence2Pwn, KFENCE may detect and report intermediate stages of the technique, emitting warnings to the kernel log due to KFENCE's detection semantics. We do not treat such warnings as a practical limitation: Fence2Pwn is intended to be embedded in a privilege-escalation attack. Upon successful escalation, the adversary can tamper with locally stored log artifacts to reduce forensic visibility. Conversely, if escalation fails, the resulting log entries are comparable to the observable artifacts produced by failed kernel privilege-escalation attempts in general.

Exploitation Goal. The goal of our exploitation technique, Fence2Pwn, is to (re-)enable powerful heap-manipulation primitives in hardened kernels by creating attacker-influenced heap layouts for allocations enabled through KFENCE. Specifically, Fence2Pwn enables controlled overlap of memory slots between objects of different sizes and security contexts (an effect analogous to cross-cache reuse [55]) even when conventional slab hardening would otherwise prevent or substantially constrain such layouts. In this paper, we focus on exploiting Fence2Pwn in the presence of Use-After-Free (UAF) vulnerabilities, which remain a common and practically relevant bug class in modern kernels. In Section 6.6, we show how to weaponize Fence2Pwn with the real-world CVE-2023-52926 to achieve this goal, and analyze the vulnerability characteristics that make Fence2Pwn more favorable in practice.

```
1  void * ... __kasan_slab_alloc(..., void *object, ...)
2  {
3          // ...
4          if (is_kfence_address(object))
5                  return (void *)object;
6
7          tag = assign_tag(cache, object, false);
8          tagged_object = set_tag(object, tag);
9          // ...
10         return tagged_object;
11 }
```

Figure 3: `__kasan_slab_alloc` snippet. Pointers corresponding to KFENCE objects are not tagged by KASAN tag-based modes (e.g., hardware memory tagging).

4 KFENCE Weaknesses

KFENCE is designed as a low-overhead, sampling-based bug-detection mechanism for production kernels that complements an additional allocation path to the primary memory allocator. Its design prioritizes detection and diagnostic value rather than other general allocator properties. In this section, we analyze KFENCE from the perspective of allocator security, in the context of heap exploitation. Our analysis is agnostic to specific exploitation techniques and does not assume misuse or implementation defects. We examine the security-relevant characteristics that emerge from KFENCE's design.

Some of our findings may be intentional trade-offs of KFENCE, given its diagnostic goals and non-security-first design constraints. However, enabling KFENCE introduces an additional allocation path that expands the kernel's attack surface and creates new avenues for exploitation, as we will demonstrate in Section 5. For that reason, we treat these characteristics as weaknesses from a security perspective when KFENCE is enabled, independent of whether they constitute flaws in KFENCE's intended functionality.

W1: Allocation-Path Observability. KFENCE introduces timing differences in the allocation path relative to regular slab allocation. These differences arise from KFENCE-specific operations, including metadata management, additional locking, and related bookkeeping. Because KFENCE is integrated transparently into the slab allocation path, these timing variations make KFENCE-served allocations distinguishable from slab-served allocations. Moreover, KFENCE's sampling mechanism introduces a deterministic allocation periodicity (on a general scenario), resulting in recurring latency spikes that further characterize KFENCE-served allocations. As a result, the underlying allocator used is distinguishable based on observable system behavior.

W2: Lack of Type and Size Segregation. KFENCE serves allocations from a globally shared memory pool, independent of object type, size, or allocation context. This design simplifies pool management and reduces the overall allocator memory overhead. However, it also results in

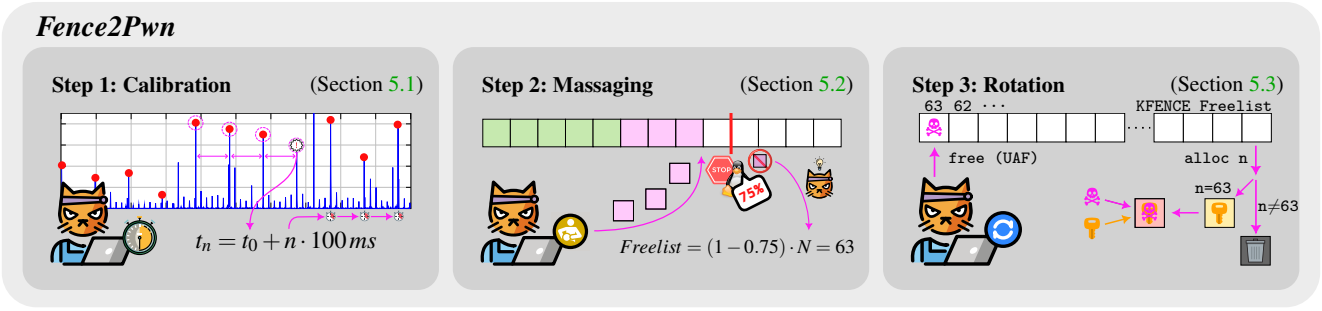


Figure 4: **Fence2Pwn** performs calibration, massaging, and rotation to overlap vulnerable with security-critical kernel objects.

objects of different types occupying the same memory locations over time, across successive slot allocation and deallocation cycles. Modern production kernels employ slab object segregation mechanisms, such as `SLAB_BUCKETS` [10] or `SLAB_MERGE_DEFAULT=n` [9] to limit the impact of vulnerabilities on allocated objects, with `SLAB_VIRTUAL` [48] proposed for stricter segregation. In addition, slab hardening includes randomization mechanisms, such as `RANDOM_KMALLOC_CACHES` [12] and `SLAB_FREELIST_RANDOM` [16], to reduce external predictability over the allocator state. The design of KFENCE does not incorporate the aforementioned hardening properties, most notably guarantees related to type segregation.

W3: Lack of KASAN Instrumentation. KASAN provides memory-safety enforcement for the Linux kernel across both debugging and production configurations. In its generic mode, KASAN employs shadow memory and compiler instrumentation to validate memory accesses at runtime, while in its hardware tag-based mode, it leverages architectural support (e.g., ARM's MTE) to enforce tag-based access checks. KFENCE operates on memory independently of these mechanisms and does not integrate with KASAN's instrumentation. Additionally, object pointers returned from KFENCE allocations are not tagged on kernels operating in tag-based KASAN modes (see Figure 3), and we observe equivalent behavior under KASAN's shadow-memory-based generic mode. As a result, the memory-safety guarantees provided by KASAN do not consistently extend to KFENCE-managed allocations.

W4: Deterministic Poisoning and Deferred Validation. KFENCE employs deterministic poison values to mark unused portions of an object slot page. Memory accesses that modify poisoned values within the object page are not detected at access time, but validation is performed during object deallocation. Because validation occurs only when an object is freed, the object lifecycle defines a window during which poisoned values may be modified and subsequently restored before any check is performed. Moreover, if the object on the corresponding slot page is not freed, integrity checks on its poisoned regions are not triggered.

W5: Non-Fatal Error Handling. KFENCE supports two configurable error-handling modes: a logging mode and a panic mode, controlled by the `panic_on_warn` kernel setting. In logging mode, detected violations are reported without terminating execution, whereas in panic mode, the kernel halts upon detection. Given that KFENCE is intended for deployment in production, high-uptime systems, the logging mode represents the practical preferred configuration to protect system uptime. Our observations align with this assumption: all observed systems use KFENCE in logging mode. Under this configuration, execution continues after a reported violation, allowing for further malicious interaction with the allocator.

5 Fence2Pwn

In this section, we present Fence2Pwn, a technique that leverages KFENCE allocation paths to induce controlled memory overlap between kernel objects of different caches and sizes. It does so by exploiting and combining the weaknesses that we presented in Section 4. By operating outside the regular allocation regime, Fence2Pwn is not constrained by modern allocator slab defenses. Fence2Pwn utilizes the shared memory pool used by KFENCE to reclaim memory slots for objects of any type or size **(W2)** and without address sanitization instrumentation **(W3)** like MTE, while Fence2Pwn exploits the other weaknesses for practical exploitation.

Overview. Fence2Pwn consists of three phases as illustrated in Figure 4. *First*, a calibration phase synchronizes allocations with KFENCE's periodic sampling behavior. *Second*, a massaging phase drives the KFENCE pool into a state with a predictable number of free slots. *Third*, a rotation phase advances the KFENCE freelist to reclaim a previously freed slot, inducing controlled overlap between kernel objects. We achieve the controlled memory overlapping of different kernel objects, which can be used by multiple end-to-end privilege escalation techniques as discussed in Section 7.

5.1 Calibration

The calibration phase leverages the allocation-path observability weakness **(W1)** to detect and infer future KFENCE allocation events. KFENCE is integrated transparently into the kernel allocation path and serves allocations according to

an internal sampling mechanism, configured via `CONFIG_KFENCE_SAMPLE_INTERVAL`. Because KFENCE allocations follow a different execution path than regular slab allocations, they incur a different measurable allocation latency.

By correlating these latency outliers with KFENCE's periodic sampling behavior, an attacker can synchronize with KFENCE's internal allocation schedule. The attacker triggers frequent kernel allocations from userspace and measures their allocation latencies using a timer. While the desired allocations are issued by inducing syscalls, the syscall's latencies are measured via `rdtsc` on x86-64 or `arm64-vct` on `arm64`, both unprivileged [17]. From the resulting timing measurements, the attacker isolates allocations exhibiting the slower and periodic KFENCE pattern. KFENCE activation times can be modeled as a periodic sequence in Equation (1), where initial activation time t_0 denotes the first observed KFENCE activation and T_{KFENCE} represents the KFENCE sampling interval. On default configurations, this interval is $100ms$ on desktop Linux and $500ms$ on Android GKI.

$$t_n = t_0 + n \cdot T_{KFENCE}, \quad n \in \mathbb{N} \tag{1}$$

This temporal model enables a synchronization mechanism with future KFENCE allocations. Given the calibrated reference time t_0 and the periodic sequence in Equation (1), an attacker configures a high-precision timer to issue function calls that coincide approximately with subsequent KFENCE activations. For our evaluation, we used the unprivileged timer accessible via the syscall interface, e.g., `timerfd_settime`. The attacker may issue additional allocations before and after the expected KFENCE window to compensate for the overall subtle temporal drift and scheduler randomness.

Naive Spraying. In scenarios where precise calibration is not feasible, an attacker can proceed without explicit calibration. Rather than relying on synchronization with KFENCE allocation windows, an attacker with sufficient object spraying capabilities can continuously trigger allocations and assume a subset of these will be served by KFENCE. This strategy results in a less precise, but possible, version of the attack.

5.2 Massaging

The massaging phase shapes the KFENCE allocation pool into a state with a predictable number of active slots. KFENCE implements a distinct allocation regime activated once pool utilization exceeds a 75 % threshold. Beyond this, KFENCE does not serve further allocations originating from call stacks that already have an active KFENCE-served object. The transition into this regime can be observed: allocation request from the same call stack stop to be served by KFENCE. Therefore, the number of free slots can be inferred to be approximately 25 % of the total number of slots in the pool.

Using the synchronized timer from the calibration phase, an attacker issues kernel allocations (e.g., `msg_msg`) close to the KFENCE activation window. Requests issued near these windows are likely served by KFENCE. KFENCE continues to serve such allocations until pool utilization reaches 75 %, at which point it transitions into a distinct utilization regime. In this regime, KFENCE does not serve further allocations originating from call stacks that already have an active object served by KFENCE. As a result, subsequent attacker-related allocations are no longer served by KFENCE, indicating the utilization threshold has been reached.

After having entered this special regime, an attacker can estimate the size of the freelist as 25 %, as the special regime triggers at 75 % pool utilization. The default KFENCE pool size is 255 objects, while on Android GKI it is 63.

Naive Spraying. Synchronization is not strictly required in all scenarios. Repeatedly issuing allocations at a significant frequency (e.g., one allocation per millisecond) results in a predictable fraction of allocations being served by KFENCE. Under this assumption, approximately one out of every hundred allocations is expected—in the best-case scenario—to be routed through KFENCE, on its default configuration. By issuing a sufficiently large number of allocations, it is therefore possible to populate the KFENCE pool until it reaches the special pressure regime. Once this regime is active, further sprayed allocations from the same call stack are no longer served by KFENCE, having no impact on the overall strategy. Consequently, an attacker can spray an upper-bound estimate of allocations to proceed with a less precise attack.

Pre-Existing Special Regime. Because we can detect whether KFENCE is already in the special regime, we proceed as following when detected. We repeat allocations for an extended period following KFENCE's periodic schedule, aiming to reclaim KFENCE-served slots. When the system releases KFENCE-served objects from other services, these slots are then reclaimed by our allocations. After some time, we are likely to control multiple objects that, once freed, reduce the system below the 75 % threshold. This converts the special-regime case into the basic scenario described above.

Since repeated allocations aligned with KFENCE's periodicity are non-intrusive, we can repeat this phase multiple times. If we cannot successfully transition to the basic scenario even after multiple attempts, we can proceed as follows. If the vulnerable object, as well as the object to be reclaimed, allows spraying, the massaging step can be skipped, at the cost of a less precise estimate of the effective freelist size. If not, and no KFENCE-served objects are released during this period, this phase may need to be retried later.

Timer Precision. Over time, scheduling effects and system activity may introduce temporal drift between the timer and KFENCE's internal sampling mechanism. To compensate this drift, the timer may require adjustment. Alternatively, additional allocations can be issued within each expected KFENCE activation window to account for misalignments.

5.3 Rotation

The rotation phase advances the KFENCE freelist in a controlled manner to reclaim a previously freed object slot. We refer to the object occupying the slot prior to reuse as the *vulnerable object*, and to the subsequently allocated object that reclaims this slot as the *victim object*. KFENCE manages free object slots using a FIFO freelist: when a KFENCE-served object is freed, its corresponding slot is appended to the tail of the freelist, while subsequent KFENCE allocations consume slots from the head of the freelist. Consequently, a freed slot is only reused after the freelist has been fully traversed.

Given the freelist size estimate, we induce a slot reuse by carefully advancing the freelist as follows: *First*, if KFENCE-served objects remain in the freelist from the massaging phase, a bounded number of these are freed to move KFENCE out of the special regime. *Second*, the vulnerable object is allocated through KFENCE and immediately freed, placing the entry at the tail of the freelist. *Third*, a sequence of allocations and deallocations is issued to advance the KFENCE freelist, while tracking the approximate remaining distance to the initially freed slot. *Finally*, shortly before completing a full traversal, we issue the allocation of the victim object, which reclaims the slot corresponding to the initially freed vulnerable object.

Untagged Pointer. Because KFENCE-served objects are not subject to pointer tagging **(W3)**—e.g., with ARM MTE—, dangling references to an object slot remain valid when the corresponding slot is later reclaimed. As a result, accessing the reclaimed slot through a dangling pointer does not require overcoming pointer-tag matching, such as imposed by the MTE with a success probability of $\frac{1}{16}$ (i.e., the probability that two tags coincide).

Object Alignment. KFENCE serves objects up to `PAGE_SIZE` on an object page, surrounded by guard pages. KFENCE randomly places the served object within the object page: either at the beginning or at the end, accounting for alignment. As a result, when a slot is reclaimed, the new object may not overlap the same position as the previously served object. This misalignment can be addressed in two ways. *First*, if one of the objects occupies the full KFENCE object page, placement randomization does not occur, and the objects necessarily overlap. *Second*, the rotation step can be repeated until both objects are placed at matching offsets within the object page. Because KFENCE reports detected violations via non-fatal diagnostics **(W5)** in common configurations, repeated attempts remain feasible. If diagnostics are fatal, the attacker can retry the attack **(W4)**: if the overlap attempt fails, the attacker either restores the corrupted poison values or avoids freeing the object occupying the affected slot, thereby fixing the poison check or preventing its.

Imprecise Reclaim. During the rotation phase, concurrent activity from other subsystems or processes may induce additional KFENCE activity that is not accounted for in the estimate freelist progression. Such interference can introduce discrepancies between the inferred and actual freelist state, causing the reclaimed slot to appear earlier than expected. To mitigate this effect, a margin can be incorporated into the reclaim process. Rather than issuing a single allocation at the end of the rotation phase, a batch of allocations can be performed to increase the likelihood that one of them reclaims the slot corresponding to the vulnerable object. Issuing multiple allocations of a simple object from the same call stack is only possible if KFENCE is not under the special utilization regime. For this reason, if the attacker holds objects from the massaging step, a subset of them are freed to reduce the pool utilization percentage of KFENCE. Otherwise, the attacker can find different allocation paths for the same object.

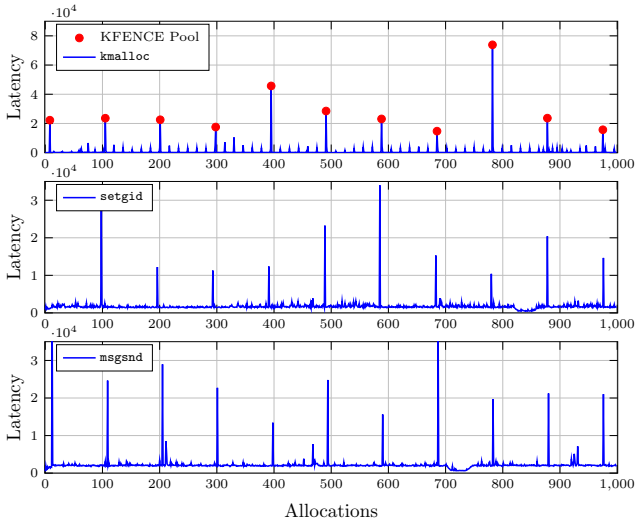


Figure 5: Privileged (i.e., `kmalloc`) and unprivileged (i.e., `setgid` and `msgsnd`) latency measurements via `rdtsc`, showing observable latency and periodicity patterns.

6 Evaluation

This section evaluates KFENCE's security-relevant properties: the observable allocations, allocation dynamics, the integration with kernel sanitizers, and Fence2Pwn's success rate.

Environment. Our primary evaluation environment consists of `x86_64` KVM guests running Linux `6.17.4` on QEMU `10.1.2` with a *buildroot* environment. We also evaluate on bare-metal devices—the ASUS Zenbook S14 and the Pixel 8a—with default production-ready distributions—Fedora 43 and Android 16. Experiments are conducted on a host equipped with an Intel Core Ultra 7 258V processor, with the exception of Android-related experiments.

6.1 Timing Side Channel

We evaluate the KFENCE-introduced timing side channel by first characterizing its allocation behavior in a privileged

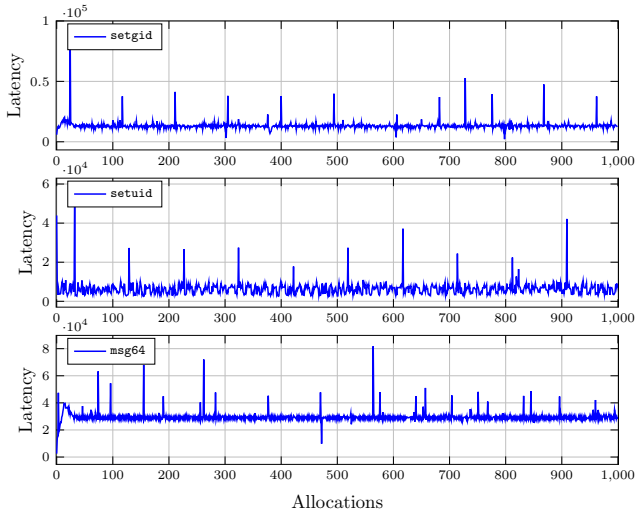


Figure 6: Unprivileged latency measurements of `setgid`, `setuid`, and `msgsnd` in cycles, issued at a fixed rate of one allocation per millisecond. Measurements taken on an ASUS Zenbook S14 with Fedora 43 Workstation under normal usage conditions.

setting. We then show that the same behavior is observable from unprivileged userspace.

First, in a privileged context via a kernel module, we measure the latency of kernel allocations and record whether each allocation is served by the slab allocator or by KFENCE. We issue periodic `kmalloc` calls every millisecond and log both their latency in cycles and the isolated allocator that served them, establishing a ground-truth timing profile. *Second*, we measure the latency of unprivileged userspace operations that trigger kernel allocations, invoking `setgid` and `msgsnd` individually. Figure 5 visualizes the results. In the privileged `kmalloc` measurement, KFENCE-served allocations consistently incur higher latency than slab-served allocations. Additionally, latency spikes form a distinct periodic pattern—once every 100 allocations, each separated by 1*ms*—, matching KFENCE's default sampling interval of 100*ms*. On the unprivileged measurements, the same timing pattern reappears: both `setgid` and `msgsnd` exhibit periodic latency spikes aligned with the KFENCE sampling mechanism cadence observed in the privileged setting. These results demonstrate that allocation latency and periodicity suffice to distinguish KFENCE-served allocations from slab-served allocations.

We conduct our experiments using allocation and free patterns to prevent the allocation differences served by the slab or page-allocator, as shown by prior work [26, 28, 35], thereby reducing the risk of misinterpreting slow allocations.

Figure 6 shows the results for the repeated timing side channel experiment using the same unprivileged measurements on an bare-metal ASUS Zenbook S14 UX5406SA running Fedora 43 Workstation with Linux `6.18.6`. We evaluate this

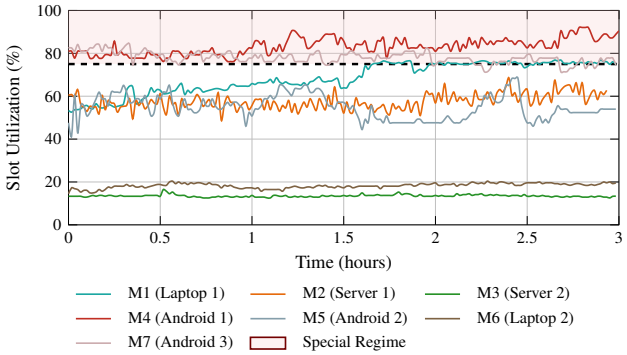


Figure 7: Slot utilization rate of the KFENCE pool over a continuous period of three hours on different machines.

machine with high uptime, background activity, and overall system noise. Similar to Figure 5, we observe the same characteristic latency and periodic KFENCE pattern.

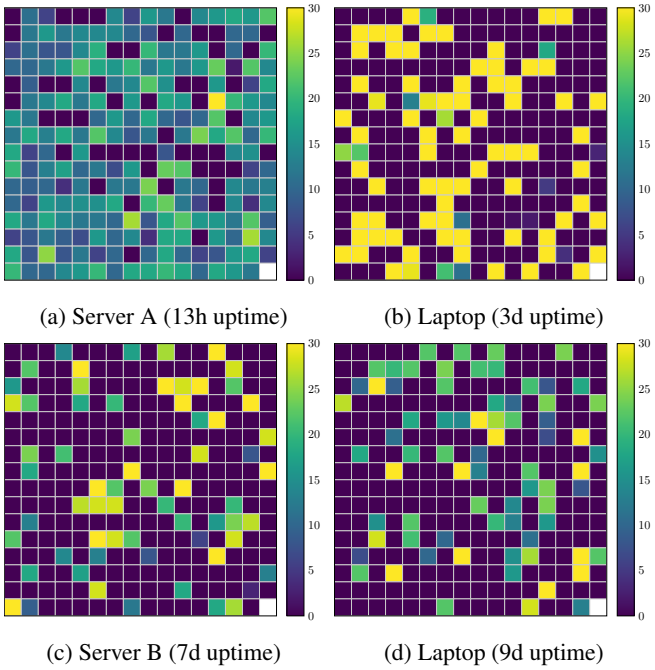
Timer. For timing measurements, we use the libcpucycles library [7], which exposes unprivileged cycle-counter based timers across architectures. On *x86-64*, libcpucycles reads the cycle counter via `rdtsc`. On *arm64*, it uses the unprivileged virtual counter exposed as `arm64-vct`. Although `arm64-vct` is less precise than `rdtsc`, it is sufficient for detecting the KFENCE timing pattern and for synchronizing with KFENCE allocation windows, as we show later in Section 6.4.

6.2 Object Slot Dynamics

This section evaluates KFENCE object slot dynamics through two experiments conducted across different systems.

First we monitor the slot utilization rate of KFENCE on different systems over a continuous period of three hours, showing how under normal usage the system acquires KFENCE slots. We sample the state of the KFENCE pool once per minute, reported by the KFENCE debug interface. We conduct this experiment across 7 system configurations (see Figure 7) spanning multiple kernels and distributions, including Android 16, Red Hat Enterprise Linux 10, Fedora 43, and Rocky Linux 10. We observe that low-uptime systems stayed under 75 %, while most systems with higher uptime converge at 75 %. Crucially, for all of our observations the utilization never reaches 100 % and hardly reaches 90 %.

Second we examine how system uptime influences activity within each KFENCE slot. We measure per-slot activity over an one-hour window, sampling the state of each KFENCE slot once per second with varying uptimes. Slot activity is defined as the number of state changes on a slot during the measurement period, e.g., a slot changes state from allocated to deallocated or vice versa. Figure 8 shows the obtained measurements. On long-lived systems, KFENCE allocation activity becomes concentrated on a small subset of slots, while many slots exhibit near-zero activity. These inactive slots



(a) Server A (13h uptime) (b) Laptop (3d uptime)

(c) Server B (7d uptime) (d) Laptop (9d uptime)

Figure 8: Individual KFENCE slot activity on different systems. Data represents the number of changes on the individual slot status: allocated to deallocated or vice versa.

remain allocated for extended periods and rarely re-enter the freelist. As a result, high-uptime systems concentrate most of the memory reclaim activity on fewer slots.

6.3 Address Sanitization Bypass

This section evaluates the interaction between the KFENCE allocator and the different KASAN modes used in both debugging and production systems. These experiments show the following claims: *First*, we show that KFENCE-served objects are not tagged with ARM MTE on the latest Android. *Second*, we demonstrate that KASAN operating in shadow-memory (generic) mode does not track KFENCE-served memory.

Generic Shadow Memory. We implemented a kernel module that allocates an object via `kmalloc` with `PAGE_SIZE-2` length, then we repeatedly issued allocations until we obtained both a KFENCE and a slab-served object. For each object, we perform a 1-byte *out-of-bounds* read and record whether the access is detected or proceeds without triggering any diagnostics. The experiments are conducted on a kernel with `KASAN_GENERIC` and `KASAN_VMALLOC`. We repeated the experiment 100 times. We observe that the slab-served object reliably triggers a slab *out-of-bounds* report in all of the cases, whereas the KFENCE-served object read is not detected.

Android Memory Tagging. We also developed a synthetic kernel module that allocates memory via `kmalloc`, simulating a vulnerable victim object allocation. For each allocation, we record the *top byte* of the pointer and the allocator that served it: slab or KFENCE. We performed 10.000 kernel allocations on a Pixel 8a with Android 16 (latest at publication) and `memtag-kernel` enabled in the bootloader. From the set of allocated objects, KFENCE-served pointers consistently carried the `0xFF` *catch-all* tag (i.e., no MTE sanitization), while slab-served allocations carried a random tag.

Table 1: Fence2Pwn success rate.

Configuration		Fence2Pwn			Outcome				Success
Kernel	Victim Object	*r*	*p*	*a*	S&U	S&D	F&U	F&D	
Baseline									
`6.17.4`	`cred`	72	1	4	90	0	0	10	90%
`6.17.4`	`cred`	72	1	7	79	0	12	9	79%
`6.17.4`	`cred`	72	1	3	75	0	3	22	75%
`6.17.4`	`cred`	72	1	6	72	0	11	17	72%
`6.17.4`	`cred`	72	1	5	68	0	4	28	68%
`6.17.4`	`cred`	72	1	2	36	0	8	56	36%
Single Victim Object									
`6.17.4`	`cred`	74	1	1	37	0	3	60	37%
`6.17.4`	`cred`	71	1	1	34	0	12	54	34%
`6.17.4`	`cred`	70	1	1	33	0	4	63	33%
`6.17.4`	`cred`	73	1	1	33	0	3	64	33%
`6.17.4`	`cred`	75	1	1	32	0	7	61	32%
`6.17.4`	`cred`	72	1	1	31	0	1	68	31%
Multiple Kernel Versions									
`6.12.67`	`cred`	72	1	4	75	0	3	22	75%
`6.6.121`	`cred`	72	1	4	60	0	13	27	60%
`6.1.161`	`cred`	72	1	4	65	0	8	25	65%
`5.15.198`	`cred`	72	1	4	52	0	7	41	52%
Multiple Objects									
`6.17.4`	`msg_msg`	72	1	4	54	0	30	16	54%
`6.17.4`	`pipe_buffer`	72	1	4	34	65	0	0	34%

Notes: S&U success and undetected; S&D success and detected; F&U failure and undetected; F&D failure and detected.
Parameters: *r* guessed freelist length; *p* vulnerable objects; *a* victim objects.

6.4 Fence2Pwn

This section evaluates Fence2Pwn in different environments, victim objects, kernel versions, CPUs, and noise levels.

We evaluate the effectiveness of Fence2Pwn by measuring its overlap success rate across configurations, kernel versions, and victim object types. Each experiment consists of 100 independent virtual machines in which Fence2Pwn is used to overlap a vulnerable UAF object with a victim object belonging to a different slab cache. Outcomes are classified based on whether an overlap is achieved and whether KFENCE detects a memory-safety violation during the attempt. To model an UAF-vulnerable object, we utilize a synthetic vulnerability implemented in a kernel module, which allows us to arbitrarily allocate/free objects while retaining a dangling reference.

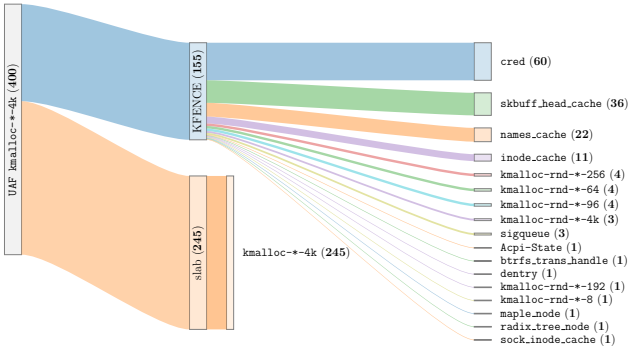
For the majority of this evaluation, we overlap a vulnerable object via `kmalloc` with a `cred` object, which includes the privilege level from an attacker-spawned process. We measure the success of an experiment by overwriting the `cred->euid` field of a process assuming the overlap, and verifying that the *effective UID* changed via `ps -aux` and `/proc/<pid>/status`. An attempt is considered detected (from the system's perspective) if KFENCE reports a memory-

safety issue via `dmesg` or its *sysfs* statistics interface. For victims objects other than `cred`, as a general mechanism, successful overlap is detected via separate KFENCE instrumentation by recording the address returned by KFENCE-served allocations. Each experiment is parameterized by three Fence2Pwn values: r corresponds to the attacker's estimate of the effective freelist length after the massaging phase; p is the number of vulnerable UAF objects to allocate; and a, the number of victim allocations issued during the rotation phase.

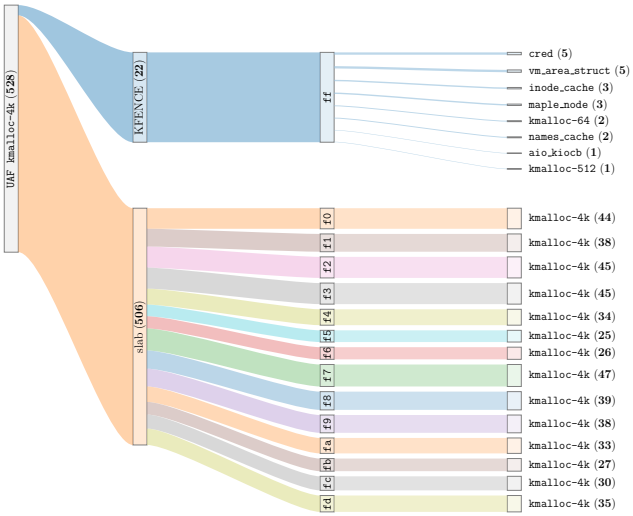
Table 1 summarizes the results of these experiments, using `cred`, `msg_msg`, and `pipe_buffer` as victim security-relevant objects. As baseline, Fence2Pwn achieves success rates up to 90 % by allocating 4 victim `cred` objects on the rotation phase. In the single-victim configuration, we show that Fence2Pwn remains feasible even with a precise memory reclaim: one single vulnerable and victim object. Experiments across different kernel versions and victim objects demonstrate that Fence2Pwn is not specific to a single kernel version or victim object in our UAF-based setting. Observed variations in success rates are primarily attributed to subtle timing differences and allocator behavior changes that necessitate re-calibration of Fence2Pwn internal implementation parameters, such as timing delays and allocation pacing.

Consumer Device Evaluation. We assess Fence2Pwn on two consumer devices representative of real-world environments, complementing the previously obtained results on the Fence2Pwn evaluation. These experiments span different kernel versions, configurations, and CPUs (*x86-64* and *arm64*). Specifically, we evaluate Fence2Pwn on a Google Pixel 8a running Android 16 GKI with `memtag-kernel` (Kernel MTE) enabled, and on an ASUS Zenbook S14 running Fedora 43 Workstation. On Android, due to the different CPU architecture, we use the unprivileged and less-precise `arm64-vct` [7]. For each device, we perform Fence2Pwn with a conservative configuration of 8 vulnerable objects and 50 victim objects. Figure 9 summarizes the observed outcomes, visualizing which allocator served the UAF object (i.e., slab or KFENCE), and the pointer tag for Android. Both diagrams show how using Fence2Pwn, we reclaim a `kmalloc-*-4k` object as objects from different slab caches, prominently from the `cred` cache. This shows how KFENCE-served objects do not follow the heap segregation guarantees and memory is reclaimed by objects that should be segregated, which is not intended. On Android, the results demonstrate that objects served by KFENCE all have a matching *catch-all* `0xff` tag, while slab-served objects carry a randomized tag.

Stress Noise Evaluation. Fence2Pwn relies on the ability to route attacker-related allocations through the KFENCE pool. On systems under high load, this step can become the limiting factor of Fence2Pwn, as KFENCE sampling is global and allocation opportunities are shared among all running processes. We therefore evaluate that attacker allocations can still be served by KFENCE in the presence of substantial system stress. We execute Fence2Pwn concurrently with a



(a) ASUS Zenbook S14 (*x86-64*); Fedora 43 Workstation



(b) Pixel 8a (*arm64*); Android 16 with `memtag-kernel`.

Figure 9: Fence2Pwn on consumer devices, showing the reclaim of an attacker object from a `kmalloc-*-4k` cache. Slab-served objects are only reclaimed from the same cache and with randomized tags. KFENCE-served objects are reclaimed by objects of different caches and without MTE tags.

range of realistic workloads generated using the Phoronix Test Suite [43]. These workloads induce varying levels of CPU utilization, scheduling contention, and kernel allocator pressure, all of which directly or indirectly affect the object's placement. For each workload, we record whether the Fence2Pwn UAF object allocations in the rotation phase were served by the slab allocator or by KFENCE, crucial for succeeding on the attack. We configure Fence2Pwn to use 8 vulnerable objects. Table 2 summarizes the results, showing that Fence2Pwn continues to place the UAF objects into the global KFENCE pool across most workloads. In some cases, fewer or no objects are served by KFENCE, which we attribute to increased system pressure and allocator activity, e.g., during `pts/nginx`.

Table 2: Fence2Pwn Evaluation Under Noise.

Phoronix Test Suite	System Conditions				UAF	
Test Name	CPU %	Threads	kmalloc/s	kfree/s	slab	kfence
`system/openssl`	97	1056	1695	3245	3	5
`system/libreoffice`	9	1377	2213	5228	7	1
`pts/phpbench`	12	1424	2172	4112	6	2
`system/stockfish`	99	1067	1565	3092	3	5
`system/compress-7zip`	60	1064	1805	3777	3	5
`pts/nginx`	93	1081	2247	330635	8	0
`pts/redis`	3	1062	2079	3726	6	2
`system/sqlite`	9	1097	7386	16892	6	2
`pts/sysbench`	85	1069	1709	3135	5	3

Conditions: Mean of measurements taken periodically each 5 seconds.
System: Zenbook S14 + Fedora 43 Workstation (Kernel `6.18.7`)
Fence2Pwn: $r = 72$, $p = 8$, $a = 50$.

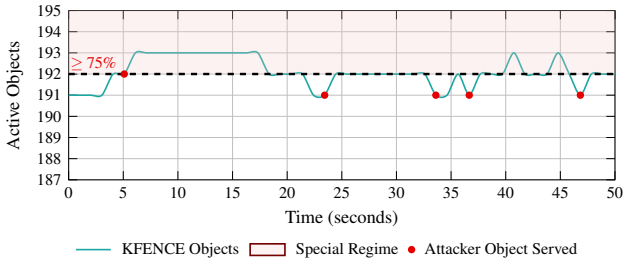


Figure 10: KFENCE Slot Stealing. An attacker accumulates KFENCE-served objects during KFENCE utilization dips.

6.5 Slot Stealing

During the massaging phase of Fence2Pwn (see Section 5.2), attacker objects used to drive KFENCE into its special pressure regime are freed before the rotation phase. This reduces the pool utilization below 75 %, causing KFENCE to exit the special regime and allowing an attacker to allocate more than one object per call stack into KFENCE, if required. However, on long-lived systems, KFENCE may already be operating in this regime before the attack begins. This experiment evaluates whether an attacker can still accumulate KFENCE-served objects under these conditions, in order to free them later and drive KFENCE below the utilization rate threshold.

We monitor KFENCE pool utilization on a bare-metal system with an uptime of six days, where the KFENCE pool utilization is at 75%, similar to other systems with a high utilization rate (see Figure 7). Meanwhile, we issue allocations corresponding to Fence2Pwn's second phase and record whether they are served by KFENCE. We then correlate the KFENCE pool utilization with attacker allocations that were served by KFENCE. Figure 10 shows that an attacker can accumulate objects even in high utilization rate scenarios. During the small natural dips below 75 %—caused by common object lifecycles— the attacker is able to accumulate objects. Each red dot represents the measured timestamp where KFENCE served an object from Fence2Pwn. At $t = 50$ in Figure 10, an attacker could free all objects corresponding to the red slots, driving the KFENCE utilization below 75 % and leading to an equivalent initial scenario as KFENCE being under 75 %.

6.6 CVE-2023-52926

To complement the synthetic vulnerabilities used in Section 6.4, we demonstrate that Fence2Pwn can be applied to a real-world kernel vulnerability. We use CVE-2023-52926, a *use-after-free* write vulnerability in the `io_uring` subsystem. The vulnerability allows a buffer object, `io_buffer_list`, to be freed while a pending `IORING_OP_READ` request still retains a dangling reference to it. When the request completes, the kernel accesses the stale reference and increments `io_buffer_list->head`, yielding a constrained UAF write primitive. Specifically, it allows an increment of the value at offset `0x16` inside a `kmalloc-64` object.

In this experiment, we use an unmodified Linux `6.6.66` kernel containing the original CVE-2023-52926 vulnerability. We apply Fence2Pwn through the native `io_uring` interface, rather than through a synthetic vulnerability. As in the previous evaluations, we first calibrate KFENCE's sampling period (Stage 1) and massage the KFENCE pool with placeholder allocations (Stage 2), such as `msg_msg`. During the rotation phase (Stage 3), we use the calibrated timer to place p `io_buffer_list` allocations on the KFENCE path and subsequently free them. We then continue the rotation by allocating and freeing r dummy objects, advancing the KFENCE freelist. Finally, we reclaim the dangling KFENCE slots as `msg_msg` objects and trigger the UAF write on `io_buffer_list`. This increments the `msg_msg` type header, which we can observe through the message-queue interface. We use this corruption as a visible witness of controlled object overlap, not as a claim of an end-to-end exploit for CVE-2023-52926. The primitive provided by this CVE is limited: it increments a fixed field through the stale `io_buffer_list` reference, and in our experiment this does not by itself yield a stronger exploit primitive. A malicious attacker could instead choose a different reclaim object depending on the intended root-pivot path, e.g., targeting refcounts for wrap-arounds, pointers, or buffer-length fields, a strategy explored by prior work on control metadata fields [58]. To summarize, this experiment demonstrates that Fence2Pwn can weaponize a real UAF vulnerability up to the allocator-control stage, moving the vulnerable object onto the KFENCE path and reclaiming its dangling slot with an attacker-selected victim object.

Vulnerability Characteristics. Fence2Pwn is applicable when the vulnerable object can be routed through KFENCE. Kernel exploitation is carried out through userspace-exposed kernel interfaces—usually syscalls—, which differ in behavior: number, order, and timing of allocations performed through the call path. These interface-specific properties thus determine whether the attacker can align KFENCE's sampling window with the vulnerable allocation. The most favorable

case is a vulnerable object allocated early in the call path, ideally as the first KFENCE-eligible allocation triggered by the attacker-controlled operation. If the vulnerable allocation occurs later in the call path, an attacker can offset the calibrated KFENCE trigger to account for the delay between entering the vulnerable call path and reaching the target allocation. Unrelated allocations before the vulnerable object are not inherently prohibitive, but they must be predictable or sufficiently separated in time from the target allocation, so that the attacker can calibrate the allocation of the vulnerable object during the synchronized KFENCE sampling window.

Conversely, vulnerabilities become less suitable when the target allocation is directly surrounded by many uncontrollable KFENCE-eligible allocations, as it becomes harder to synchronize the vulnerable allocation with the KFENCE timer, and it will likely route one of the unrelated allocations through KFENCE. Finally, the temporal behavior of the vulnerability must leave enough time for the KFENCE slot to be reused. Contrary to traditional slab exploitation, KFENCE activity is sampled at a certain rate, meaning that the freed UAF object cannot be immediately reallocated and accessed. Immediate or quick free-and-use sequences are therefore less suitable, whereas delayed completions, deferred work, or attacker-controlled accesses provide a suitable window for Fence2Pwn's rotation phase.

Vulnerable page-sized allocations—whether fixed-size or elastic—are particularly favorable, as they force the object to occupy the full KFENCE slot and thereby eliminate KFENCE's randomized object placement. For smaller objects, whose allocation size is not rounded up to a full page for cache-alignment reasons, KFENCE places the object either at the beginning or at the end of the object page. Consequently, a reclaimed object overlaps the dangling UAF object only if both allocations are placed on the same side of the KFENCE slot, yielding a 50 % overlap probability per attempt.

CVE-2023-52926 Characteristics. CVE-2023-52926 satisfies the vulnerability characteristics required by Fence2Pwn. First, the vulnerable object, `io_buffer_list`, is allocated through an attacker-triggerable `io_uring` operation. The allocation of the target object occurs early in the relevant syscall path and is stable across repeated invocations, providing favorable conditions for aligning with the calibrated KFENCE sampling window. Second, the free and the later use of the vulnerable object are temporally separated and attacker-controlled: the `io_buffer_list` can be freed by unregistering the provided buffer ring, while the pending `IORING_OP_READ` request retains the dangling reference consumed on completion.

This separation gives Fence2Pwn sufficient time to rotate the KFENCE freelist and reclaim the freed slot with a victim object, previous to triggering the UAF write. Finally, `io_buffer_list` is neither page-sized nor elastic. Thus, the allocation cannot be forced to occupy an entire KFENCE object page. Instead, KFENCE places the object either at the beginning or at the end of the object page. Consequently, the dangling pointer refers to one of these to placement, and a subsequently reclaimed victim object overlap only if KFENCE chooses the same side of the page. This yields a 50% chance of overlap when the target slot is properly reclaimed. We observe this behavior experimentally: unfavorable placements within the same KFENCE slot page led to KFENCE diagnostic messages, whereas same-side placements corrupted the `msg_msg` metadata as intended. Since KFENCE errors are non-fatal due to **(W5)**, the experiment can be retried until the placement matches. After successful exploitation, the attacker could remove the messages from KFENCE.

7 Discussion

In this section, we discuss the security implications of Fence2Pwn, the affected distributions, immediate and future mitigations, as well as limitations and future work. To find effective defenses limiting Fence2Pwn, we are in the process with the Linux kernel security team and Google (Section 9), where parts of the solutions are based on the discussion.

Security Implications. This work presents Fence2Pwn, a KFENCE-enabled kernel exploitation technique that re-enables cross-cache-style object overlay, including for security-critical objects, by moving allocations onto a less protected path. This is particularly impactful because modern slab-hardening mechanisms make memory-reuse attacks substantially more difficult, including memory tagging, heap segregation (e.g., `SLAB_BUCKETS` [10] and `RANDOM_KMALLOC_CACHES` [12]), and deterministic cross-cache mitigation (i.e., `SLAB_VIRTUAL` [22, 48]). By restoring the overlay capability under hardened kernels, Fence2Pwn can provide a building block for exploitation chains to achieve privilege escalation.

Affected Distributions. Table 3 showcases the KFENCE configurations deployed across major Linux distributions.[2] We classify a system as susceptible to Fence2Pwn when KFENCE is both enabled and active, and we validate this by demonstrating real susceptibility on a subset of systems. We consider KFENCE is active when the number of KFENCE objects and the sampling interval are greater than zero. We further examined whether distributions make use of the KFENCE skipping toggle for security-critical object types, and found no distribution that enables this mechanism in practice.

KASAN Instrumentation. Software-based KASAN modes are not suitable for protecting KFENCE allocations in production kernels, as they rely on compiler-inserted instrumentation throughout the kernel. Therefore, it imposes an overhead that conflicts with KFENCE's low-overhead sampled production use case. The production-suitable KASAN mode is hardware tag-based KASAN, which uses hardware

[2] Ubuntu introduced KFENCE during the 21.10 development cycle but disabled it citing instability in certain scenarios [41]. Canonical noted that an investigation is ongoing to try to re-enable KFENCE in the future.

Table 3: KFENCE and slab config across major distributions.

Distribution	SLAB					KFENCE					
	1	2	3	4	5	6	7	8	9	10	11
Generic Linux 6.17 (`defconfig`)	✗	✗	✓	✗	✗	✗			✓		
Generic Linux 6.17 (`hardening`)	✓	✗	✓	✓	✗	✓	255	100	✓	✗	
Android 16 GKI (`6.1`)		✓	✗		✓	✓	63	500	✓	✗	0
Android 16 GKI (`6.12`)	✓	✓	✗	✗	✓	✓	63	500	✓	✗	?
RedHat Enterprise 10	✓	✗	✗	✗	✓	✓	255	100	✓	✗	0
Fedora 43	✓	✗	✗	✓	✓	✓	255	100	✓	✗	0
Rocky Linux 10	✓	✗	✗	✗	✓	✓	255	100	✓	✗	0
Arch Linux	✓	✗	✓	✗	✓	✓	255	100	✓	✓	0
CentOS 10	✓	✗	✗	✗	✓	✓	255	100	✓	✗	0
Alma Linux	✓	✗	✗	✗	✓	✓	255	100	✓	✗	0
NixOS 25.11	✓	✗	✓	✓	✓	✓	255	100	✓	✗	0
Oracle Linux 10 (RHEL)	✓	✗	✗	✗	✓	✓	255	100	✓	✗	0
Oracle Linux 10 (UEK)	✓	✗	✓	✓	✓	✓	255	0	✓	✗	0
Debian 13	✓	✗	✓	✗	✓	✓	255	0	✓	✗	0
Ubuntu 24.04 LTS (`6.14`)	✓	✗	✓	✓	✓	✓	255	0	✓	✗	0
Ubuntu 26.04 LTS (dev)	✓	✗	✓	✓	✓	✓	255	0	✓	✗	0
SUSE Enterprise 16	✓	✗	✓	✓	✓	✓	255	0	✓	✗	0

(1) `CONFIG_SLAB_BUCKETS`
(2) `CONFIG_KASAN_HW_TAGS`
(3) `CONFIG_SLAB_MERGE_DEFAULT`
(4) `CONFIG_RANDOM_KMALLOC_CACHES`
(5) `CONFIG_MEMCG`
(6) `CONFIG_KFENCE`
(7) `CONFIG_KFENCE_NUM_OBJECTS`
(8) `CONFIG_KFENCE_SAMPLE_INTERVAL`
(9) `CONFIG_HAVE_ARCH_KFENCE`
(10) `CONFIG_KFENCE_DEFERRABLE`
(11) `/sys/kernel/slab/cred/skip_kfence`

accelerated tags (e.g., ARM MTE) to provide low-overhead memory-safety checks. However, KFENCE-managed allocations are deliberately excluded from tag-based KASAN instrumentation to preserve KFENCE-specific diagnostics. Applying hardware memory tagging to KFENCE objects would cause invalid accesses to be handled by the KASAN reporting path before KFENCE observes them. This would suppress KFENCE's richer diagnostic report, including the allocation and deallocation stack traces stored in KFENCE's metadata.

Potential Mitigations. As an *immediate mitigation*, we recommend disabling KFENCE in environments exposed to kernel-level exploitation until stronger hardening or further research becomes available. KFENCE can be disabled via the boot parameter `kfence.sample_interval=0` or the configuration `CONFIG_KFENCE_SAMPLE_INTERVAL=0`. On hardware supporting ARM MTE, we recommend disabling KFENCE and enabling KASAN's hardware tag-based mode.

As a *future mitigation*, we recommend applying the same slab's threat model to KFENCE. While a complete mitigation is likely difficult, several measures can raise the bar for exploitation. Changes must preserve KFENCE's intended behavior and maintain acceptable runtime and memory overheads. We consider Fence2Pwn to be fully or partially mitigated once the weaknesses in Section 4 are adequately addressed.

First, removing the timing side channel **(W1)** entirely is likely infeasible or impractical. An alternative is to introduce timing variation so that KFENCE scheduling becomes less predictable. This would only partially mitigate Fence2Pwn and make exploitation harder, since an attacker can still spray allocations that are likely to be served by KFENCE.

Second, slab-hardening efforts should be extended to KFENCE to address **(W2)**. For example, heap segregation could be implemented in KFENCE by maintaining separate KFENCE pools for different slab caches. This would preserve heap segregation for KFENCE-served objects, preventing objects from unrelated caches from reusing the same KFENCE slot. However, this design's memory overhead scales with the number of slab caches. On a modern Fedora kernel, we observe 531 slab caches. Creating a sibling KFENCE pool for each cache would therefore require 531 KFENCE pools. Since a default pool occupies 2 MiB, such a design would incur more than 1 GiB of memory overhead. Another option is to integrate a `SLAB_VIRTUAL`-like technique [48], where a virtual address is never reused for different types. This overhead would further increase with the addition of new slab caches for stronger heap segregation. Additionally, KFENCE could incorporate a quarantine-based defense to prevent memory overlap. However, as with the rejected quarantine-based defense in the Linux kernel [46], effectiveness may be limited against spraying techniques. An alternative is to adapt a page-aliasing approach [13] so multiple virtual addresses map to a single KFENCE pool. Each KFENCE allocation would receive a new virtual page while mapping to the same physical page. However, careful analysis of virtual-page reuse is required, e.g., for objects of the same type.

Third, kernels hardened with memory tagging should not leave KFENCE-served allocations untagged **(W3)**. An alternative proposed solution is to automatically disable KFENCE at runtime on systems that use KASAN in tag-based modes. Both proposed solutions would mitigate **(W3)**.

Fourth, to resist against poison value tampering **(W4)**, the used canary pattern should be unpredictable for an attacker (e.g., pseudorandom values), similar to hardened freelist pointers. However, this only provides partial protection, as the canary is still only verified at object free. Mechanisms that cause an immediate trap are preferred, e.g., memory tagging.

Finally, KFENCE currently handles detected errors by reporting them as warnings **(W5)**. A stricter response (e.g., faulting) would better align with hardened deployments. However, it would conflict with KFENCE's original intent.

Upstreamed Mitigations. As part of our disclosure to the Linux Kernel Security Team, three proposed mitigations have been merged into Linux upstream and Android. Commit `09833d99` [1, 2] disables KFENCE when kernel memory tagging is active, thereby an attacker cannot route objects to KFENCE to avoid heap memory tagging. Commit `da735962` [36, 37] introduces a KFENCE-specific, opt-in, fault-handling mode, addressing the non-fatal error handling behavior of KFENCE. Commit `870ff192` [44, 45] randomizes the KFENCE freelist order, providing a preventive mitigation for possible future spatial-corruption scenarios, as discussed in Section 7. Together, these patches mitigate im-

portant aspects of the attack surface identified in this paper, while the broader problem of aligning slab's heap segregation model with KFENCE remains open.

Limitations and Future Work. Currently, our proof-of-concept of Fence2Pwn exploits KFENCE's weaknesses by overlaying security-relevant objects of different types and sizes via a UAF-based memory-reuse attack. Our evaluation therefore focuses on UAF vulnerabilities, and we do not claim an empirical demonstration for spatial bugs such as OOB writes. Some KFENCE properties may also be relevant to spatial corruption scenarios, but characterizing those cases requires separate analysis and is left for future work.

8 Conclusion

This paper introduced a novel kernel exploitation technique, Fence2Pwn, that leverages KFENCE to bypass modern slab hardening. We identified weaknesses in KFENCE, most importantly: we can detect KFENCE-served allocations via a timing side channel; KFENCE-served allocations do not adhere to any heap segregation; and KFENCE allocations are untagged. We showed how KFENCE's design characteristics allow UAF-based memory reuse to circumvent modern slab defenses. This enables the overlay of security-relevant objects of different types and sizes. Moreover, we showed that KFENCE-served allocations bypass both the Linux kernel's production-ready hardware memory-tagging KASAN mode (MTE)—e.g., Android's `memtag-kernel`—and shadow-memory-based generic KASAN, thereby weakening the protection expected from these sanitization configurations. Finally, we outlined the set of affected distributions—including widely deployed ones such as Android GKI and Red Hat Enterprise Linux—and provided immediate mitigation guidance and recommendations for longer-term hardening.

9 Ethical Considerations

We responsibly disclosed our findings to the Linux Kernel Security Team and to the Android Security Team via their respective official procedures on December 16, 2025. Following this disclosure, the Linux Kernel Security Team has been actively developing countermeasures and patches, some of which are already merged upstream. All experiments were conducted on machines owned, operated, and controlled by the author(s) exclusively, and did not involve (sensitive) data belonging to any other individuals of any kind. In the following, we provide our stakeholder-based ethics analysis, detailing the relevant *stakeholders*, the considered *mitigations*, and the resulting *decision*.

Stakeholders. Linux is widely used in our society's software infrastructure; thus, this work may affect a broad set of stakeholders. In particular, the Linux kernel is used by end users on consumer devices and by enterprise users operating large-scale and critical Linux-based infrastructure (such as data centers and cloud services). In summary, the key stakeholders considered in this work are:

- Author(s): The author(s) of this work and their affiliated institution(s), who are responsible for adhering to ethical research practices, conducting experiments solely in safe environments, and responsibly disclosing potential vulnerabilities.
- End Users: Private individuals running Linux on consumer devices. These users may be affected by kernel-level exploitation, e.g., through the extraction of private data. This group might be especially vulnerable due to limited visibility and/or awareness of kernel-level threats, including system compromise, and might be unaware of the importance of software and system updates.
- Enterprise Users: Organizations (for-profit and non-profit) operating Linux-based systems, e.g., cloud servers or embedded/automotive systems, that may face risks from kernel exploitation, including system compromise, service disruption, and extraction of sensitive data.
- High-Risk Individuals: Journalists, activists, politicians, and other publicly or politically exposed figures who may be disproportionately affected by potential surveillance enabled by kernel-level exploitation, which may pose threats to personal safety, freedom of expression, and political participation.
- Linux Kernel Developers: Linux kernel developers responsible for maintaining the Linux kernel, who may be affected by this work, e.g., the design, implementation, and review of possible mitigations.

Mitigations. As immediate mitigation against the identified weaknesses of KFENCE of this work, we strongly recommend disabling KFENCE support for all Linux-based systems until a long-term solution for the presented kernel exploitation technique is found. This aligns with the response from the Linux kernel developers who are currently working on patches to address the identified issues in this work.

Decision. While we acknowledge that research into kernel security can cause harm, we believe that revealing these findings is crucial in demonstrating how threat actors may bypass current slab-hardening efforts, and in encouraging the urgent action that has already been taken or completed as a result of our disclosure. We therefore conclude that publishing our findings, alongside responsible disclosure, yields greater long-term security benefits than the associated risks, and is the most ethical course of action for stakeholders.

10 Open Science

We release all synthetic exploits, the CVE-2023-52926-based allocator-control experiment, the experimental environment, and all evaluation artifacts[3].

[3] https://doi.org/10.5281/zenodo.20411196

Acknowledgements

We thank the anonymous reviewers and the Shepherd for their valuable feedback that improved this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the CREIS project (FFG grant number 926841) and the RESIST project (FFG grant number 915106).

References

[1] Alexander Potapenko. mm/kfence: disable KFENCE upon KASAN HW tags enablement. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=09833d99db36d74456a4d13eb29c32d56ff8f2b6, 2026. Accessed: 2026-05-27.

[2] Alexander Potapenko. mm/kfence: disable KFENCE upon KASAN HW tags enablement. https://android.googlesource.com/kernel/common/+/09833d99db36d74456a4d13eb29c32d56ff8f2b6, 2026. Accessed: 2026-05-27.

[3] Android Developers. Arm Memory Tagging Extension (MTE). https://developer.android.com/ndk/guides/arm-mte, 2024. Accessed: 2025-02-26.

[4] Apple Security Engineering and Architecture (SEAR). Memory Integrity Enforcement: A complete vision for memory safety in Apple devices. https://security.apple.com/blog/memory-integrity-enforcement/, 2025. Accessed: 2025-09-10.

[5] ARM. Arm Architecture Reference Manual for A-profile architecture, 2 2022.

[6] Awarau and pql. CVE-2022-29582 An io_uring vulnerability, 2022. URL: https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/.

[7] Daniel Bernstein. libcpucycles, 2026. URL: https://cpucycles.cr.yp.to.

[8] Kees Cook. mm: add slub free list pointer obfuscation, 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2482ddec670f.

[9] Kees Cook. mm: Allow slab_nomerge to be set at build time , 2017. URL: https://www.openwall.com/lists/kernel-hardening/2017/06/19/33.

[10] Kees Cook. mm/slab: Introduce kmem_buckets_create and family, 2024. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b32801d1255be1da62ea8134df3ed9f3331fba12.

[11] Jonathan Corbet. Supervisor mode access prevention, 2012. URL: https://lwn.net/Articles/517475/.

[12] Jonathan Corbet. Randomness for kmalloc(), 2023. URL: https://lwn.net/Articles/938637/.

[13] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *IEEE International Conference on Dependable Systems and Networks*, 2006.

[14] Jake Edge. Kernel address space layout randomization, 2013. URL: https://lwn.net/Articles/569635/.

[15] ETenal. CVE-2022-27666: Exploit esp6 modules in Linux kernel, 2022. URL: https://etenal.me/archives/1825.

[16] Thomas Garnier. mm: SLAB freelist randomization, 2016. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c7ce4f60ac199fb3521c5fcd64da21cee801ec2b.

[17] Jingquan Ge and Fengwei Zhang. FlushTime: Towards Mitigating Flush-based Cache Attacks via Collaborating Flush Instructions and Timers on ARMv8-A. In *CCS*, 2023.

[18] Google. Linux kernel sanitizers. URL: https://google.github.io/kernel-sanitizers/.

[19] Daniel Gruss, Michael Schwarz, and Moritz Lipp. Meltdown: Basics, Details, Consequences. In *BlackHat USA*, 2018.

[20] h0mbre. Escaping the Google kCTF Container with a Data-Only Exploit, 2023. URL: https://h0mbre.github.io/kCTF_Data_Only_Exploit#.

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

[22] Jann Horn. MITIGATION_README, 2022. URL: https://github.com/thejh/linux/blob/slub-virtual/MITIGATION_README.

[23] javierprtd. No CVE for this bug which has never been in the official kernel, 2023. URL: https://soez.github.io/posts/no-cve-for-this.-It-has-never-been-in-the-official-kernel/.

[24] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security*, 2014.

[25] The Linux Kernel. Kernel Electric-Fence (KFENCE), 2021. URL: https://docs.kernel.org/dev-tools/kfence.html.

[26] Dong-ok Kim, Juhyun Song, and Insu Yun. Cross-x: Generalized and stable cross-cache attack on the linux kernel. In *CCS*, 2025.

[27] 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. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[28] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In *USENIX Security*, 2023.

[29] Yoochan Lee, Hyuk Kwon, and Thorsten Holz. Dirtyfree: Simplified data-oriented programming in the linux kernel. In *NDSS*, 2026.

[30] Zhenpeng Lin. How AUTOSLAB Changes Kernel Self-Protection the Memory Unsafety Game, 2021. URL: https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game.

[31] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating Privilege in Linux Kernel. In *CCS*, 2022.

[32] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. Bad io_uring: A New Era of Rooting for Android, 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf.

[33] William Liu, Joseph Ravichandran, and Mengjia Yan. EntryBleed: A Universal KASLR Bypass against KPTI on Linux. In *HASP*, 2023.

[34] Waiman Long. [PATCH v5 2/3] mm: memcg/slab: Create a new set of kmalloc-cg-<n> caches, 2021. URL: https://lore.kernel.org/lkml/20210512145107.6208-1-longman@redhat.com/.

[35] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In *USENIX Security*, 2024.

[36] Marco Elver. kfence: add kfence.fault parameter. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da735962d05c4e7ffc68e02c0cb2459f837a0f51, 2026. Accessed: 2026-05-27.

[37] Marco Elver. kfence: add kfence.fault parameter. https://android.googlesource.com/kernel/common/+/da735962d05c4e7ffc68e02c0cb2459f837a0f51, 2026. Accessed: 2026-05-27.

[38] Jennifer Miller, Manas Ghandat, Kyle Zeng, Hongkai Chen, Abdelouahab Benchikh, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. System Register Hijacking: Compromising Kernel Integrity By Turning System Registers Against the System. In *USENIX Security*, 2025.

[39] Joao Moreira. Kernel FineIBT Support, 2022. URL: https://lwn.net/Articles/891976/.

[40] James Morse, 2015. URL: https://lwn.net/Articles/651614/.

[41] Alex Murray. What's new in security for Ubuntu 21.10? , 2021. URL: https://ubuntu.com/blog/whats-new-in-security-for-ubuntu-21-10.

[42] Lau Notselwyn. Flipping Pages: An analysis of a new Linux vulnerability in nf_tables and hardened exploitation techniques, 2024. URL: https://pwning.tech/nftables/.

[43] Phoronix. Phoronix Test Suite, 2022. URL: https://www.phoronix-test-suite.com/.

[44] Pimyn Girgis. mm/kfence: randomize the freelist on initialization. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=870ff19251bf3910dda7a7245da826924045fedd, 2026. Accessed: 2026-05-27.

[45] Pimyn Girgis. mm/kfence: randomize the freelist on initialization. https://android.googlesource.com/kernel/common/+/870ff19251bf3910dda7a7245da826924045fedd, 2026. Accessed: 2026-05-27.

[46] Alexander Popov. Linux kernel heap quarantine versus use-after-free exploits, 2020. URL: https://a13xp0p0v.github.io/2020/11/30/slab-quarantine.html.

[47] Zhiyun Qian, Jiayi Hu, Jinmeng Zhou, Qi Tang, and Wenbo Shen. PageJack: A Powerful Exploit Technique With Page-Level UAF, 2024. URL: https://i.blackhat.com/BH-US-24/Presentations/US24-Qian-PageJack-A-Powerful-Exploit-Technique-With-Page-Level-UAF-Thursday.pdf.

[48] Matteo Rizzo and Jann Horn. Prevent cross-cache attacks in the SLUB allocator, 2023. URL: https://lore.kernel.org/linux-mm/202309151425.2BE59091@keescook/T/.

[49] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privilege, 2015. URL: https://blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf.

[50] Sami Tolvanen. Control Flow Integrity in the Android kernel, 2018. URL: https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html.

[51] Linus Torvalds. kasan: add kernel address sanitizer infrastructure, 2015. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0b24becc810dc3be6e3f94103a866f214c282394.

[52] Linus Torvalds. Merge branch 'akpm' (patches from Andrew), 2021. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=245137cdf0cd92077dad37868fe4859c90dada36.

[53] Nicolas Wu. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel, 2023. URL: https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html.

[54] Dongchen Xie, Dongnan He, Wei You, Jianjun Huang, Bin Liang, Shuitao Gan, and Wenchang Shi. BridgeRouter: Automated Capability Upgrading of Out-Of-Bounds Write Vulnerabilities to Arbitrary Memory Write Primitives in the Linux Kernel. In *S&P*, 2025.

[55] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *CCS*, 2015.

[56] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[57] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In *CCS*, 2023.

[58] Hao Zhang, Jian Liu, Jie Lu, Shaomin Chen, Tianshuo Han, Bolun Zhang, and Xiaorui Gong. Reviving Discarded Vulnerabilities: Exploiting Previously Unexploitable Linux Kernel Bugs Through Control Metadata Fields. In *CCS*, 2025.