SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel

Lukas Maar
Graz University of Technology

Stefan Gast
Graz University of Technology

Martin Unterguggenberger
Graz University of Technology

Mathias Oberhuber
Graz University of Technology

Stefan Mangard
Graz University of Technology

Abstract

While the number of vulnerabilities in the Linux kernel has increased significantly in recent years, most have limited capabilities, such as corrupting a few bytes in restricted allocator caches. To elevate their capabilities, security researchers have proposed software cross-cache attacks, exploiting the memory reuse of the kernel allocator. However, such cross-cache attacks are impractical due to their low success rate of only 40 %, with failure scenarios often resulting in a system crash.

In this paper, we present SLUBStick, a novel kernel exploitation technique elevating a limited heap vulnerability to an arbitrary memory read-and-write primitive. SLUBStick operates in multiple stages: Initially, it exploits a timing side channel of the allocator to perform a cross-cache attack reliably. Concretely, exploiting the side-channel leakage pushes the success rate to above 99 % for frequently used generic caches. SLUBStick then exploits code patterns prevalent in the Linux kernel to convert a limited heap vulnerability into a page table manipulation, thereby granting the capability to read and write memory arbitrarily. We demonstrate the applicability of SLUBStick by systematically analyzing two Linux kernel versions, v5.19 and v6.2. Lastly, we evaluate SLUBStick with a synthetic vulnerability and 9 real-world CVEs, showcasing privilege escalation and container escape in the Linux kernel with state-of-the-art kernel defenses enabled.

1 Introduction

Operating system kernels, such as Linux, are susceptible to memory safety vulnerabilities due to their size and complexity. However, most of these vulnerabilities have limited capabilities, such as corrupting a few bytes in restricted allocator caches. These limitations make exploitation difficult in practice. To make these vulnerabilities even more difficult to exploit, researchers and kernel developers have included defenses such as SMAP, KASLR, and kCFI [37]. In addition, the kernel's allocator is designed to restrict exploits that propagate from heap vulnerabilities. One particular hardening strategy is to enforce coarse-grained heap separation. This separation places objects in distinct allocator caches that maintain blocks of adjacent pages, called slabs, and separate security-critical objects from frequently used objects. Hence, vulnerabilities in frequently used caches cannot be directly exploited to manipulate security-critical objects, such as credentials.

To circumvent coarse-grained heap separation, security researchers [50] presented software cross-cache attacks, which have been used by several kernel exploits [2, 13, 17, 19, 28–30, 47]. Software cross-cache attacks exploit the memory reuse of the kernel allocator as follows: Initially, an adversary triggers a heap vulnerability to obtain and hold on to a write capability for a victim object. They then free all memory slots on the slab page containing the victim object and allocate a different (sensitive) object type. This triggers the allocation of new slab pages, presumably reclaiming the previously freed and recycled slab page. The adversary then continues to overwrite the victim object, which now resides in the same memory location as the newly allocated sensitive object, corrupting it.

The Linux kernel has two types of allocator caches: dedicated and generic caches. While dedicated caches can be reliably exploited for cross-cache attacks [2, 17, 19, 28, 47], generic caches cannot [28, 50]. In particular, exploitation of generic caches has a success rate of only 40 % [50], with failure scenarios often leading to system crashes. To increase the reliability of generic cache exploitation, security experts [13, 29, 30, 47] have used stabilization objects, e.g., `msg_msg` or `pipe_buffer`. However, these objects cannot be used in newer kernel versions due to more refined heap separation, i.e., v5.14 introduced `kmalloc-cg-*`. Therefore, for newer kernel versions, cross-cache attacks on generic caches do not provide the reliability required in practice [28, 50].

In this paper, we present SLUBStick, a novel kernel exploitation technique that converts a limited kernel heap vulnerability into an arbitrary read-and-write primitive. At its core, SLUBStick exploits timing side-channel leakage of the kernel's allocator to reliably trigger the recycling and reclaiming process for a specific memory target. Exploiting this side-channel leakage significantly enhances the success rate of soft-

ware cross-cache attacks, exceeding 99 % for generic caches with a single slab page and 82 % for multiple slab pages. With this substantial increase, our approach overcomes the prior unreliability and makes cross-cache attacks practical for exploitation. Using our reliable side-channel supported approach, SLUBStick performs a cross-cache attack to recycle a slab page that contains a write capability. SLUBStick then reclaims the slab page as a page table, i.e., Page Upper Directory (PUD), used for userspace address translation. By triggering the write capability, SLUBStick overwrites page table entries, obtaining arbitrary read and write capabilities.

To perform SLUBStick, we overcome the following technical challenges: First, we present reliably exploitable primitives for our timing side channel that are accessible to unprivileged users. Second, hardly any kernel heap vulnerabilities provide the capability to modify kernel data directly. Therefore, we present techniques that exploit code patterns prevalent in the Linux kernel. These techniques convert heap vulnerabilities before the recycling phase to allow a write capability after reclamation as a page table. Third, manipulating page table entries to obtain an arbitrary memory read-and-write primitive is challenging because the physical memory layout is randomized due to KASLR, and we do not assume address information leakage. Hence, we introduce a reliable solution that obtains such a primitive from an overwrite.

We conduct a systematic analysis for two Linux kernel versions, v5.19 and v6.2, providing a comprehensive list of primitives to successfully execute SLUBStick for all generic caches from `kmalloc-8` to `kmalloc-4096`. We also evaluate SLUBStick with a synthetic vulnerability as well as with 9 real-world CVEs for both kernel versions on x86_64 as well as aarch64, demonstrating its architecture and kernel version independence. Based on these findings, we conclude that SLUBStick poses a significant threat to kernel security.

Contributions. The main contributions of SLUBStick are:

(1) Side-Channel Supported Recycling and Reclaiming: We present a novel approach to reliably trigger the recycling process of a specific memory target and reclaim it by using a software timing side channel. Our approach shows success rates exceeding 99 % for frequently used generic caches, making cross-cache attacks practical.

(2) Novel Exploitation Method: Leveraging our reliable side-channel supported recycling and reclaiming approach, we present a novel exploitation technique to convert kernel heap vulnerabilities with limited capabilities into an arbitrary memory read-and-write primitive with state-of-the-art kernel defenses enabled.

(3) Comprehensive Analysis and Attack Evaluation: We systematically analyze two Linux kernel versions, v5.19 and v6.2, showing that SLUBStick can be executed for generic cache from `kmalloc-8` to `kmalloc-4096`. We also evaluate SLUBStick using a synthetic vulnerability and 9 real-world CVEs to escalate privileges.

Outline. Section 2 describes the background and threat model. Section 3 presents SLUBStick. Section 4 introduces our reliable recycling and reclaiming process. Section 5 describes pivoting heap vulnerabilities. Section 6 details how to gain arbitrary read-and-write capabilities. Section 7 comprehensively evaluates our attack. Section 8 discusses valuable insights and kernel defenses. Section 9 concludes this work.

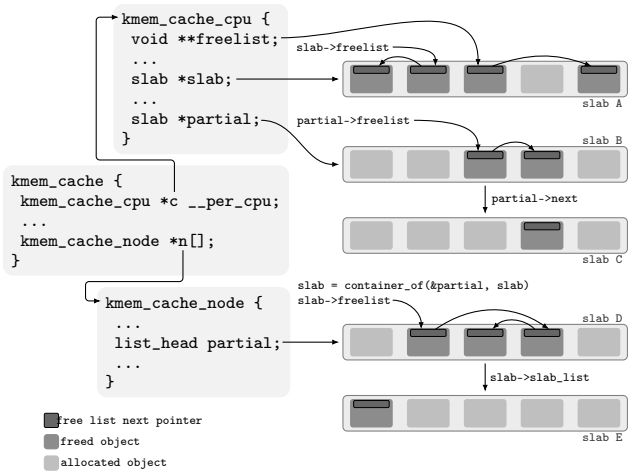


Figure 1: `kmem_cache` layout for the SLUB implementation.

2 Background and Threat Model

2.1 Buddy and SLUB Allocator

Linux's page allocator is based on the *Binary Buddy Allocator* [23], mainly referred to as buddy allocator. It allocates physical contiguous memory in chunks of page order size, i.e., $2^{\mathtt{n}} \cdot \mathtt{PAGE\_SIZE}$, where `n` is the page order. Moreover, it combines this page-order allocation with free chunk merging.

SLUB allocator. As the buddy allocator only provides page-order allocations, the slab allocator caches available objects with a predefined size in a multi-level free-list hierarchy, using pages obtained from the buddy allocator. There are three main implementations: SLUB is the default choice for several Linux distributions [22], while SLOB has become obsolete, and SLAB will be deprecated soon [8].

In the Linux kernel, the SLUB allocator provides two primary types of allocator caches: dedicated and generic caches. Dedicated caches are employed for frequently used fixed-size objects, such as `cred` or `task_struct`. Generic caches are utilized for generic object allocation and deallocation or for objects whose sizes are not known during compile-time, e.g., elastic objects [5]. Both types of caches utilize `kmem_cache`, with each dedicated cache having its own `kmem_cache`, while generic caches have multiple `kmem_caches` matched to different sizes. When allocating memory from a generic cache, the kernel matches the requested size to one of these caches and allocates an object from the corresponding `kmem_cache`.

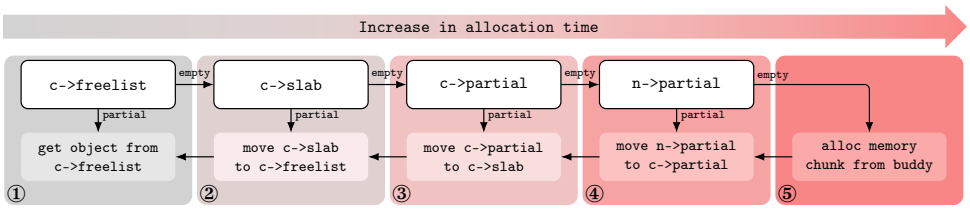


Figure 2: SLUB allocation of an object, where the terms `c` and `n` refer to the `kmem_cache_cpu` and `kmem_cache_node`, respectively. The free lists (i.e., `c->freelist` and `c->slab->freelist`) and slab lists (i.e., `c->partial` and `n->partial`) are checked to be either `empty` or `partial`.

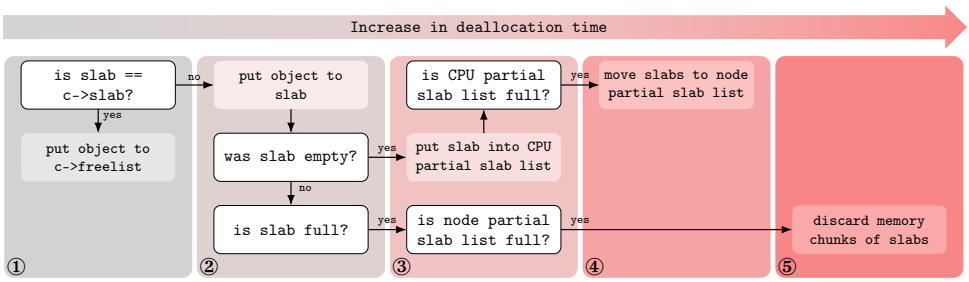


Figure 3: SLUB deallocation of an object, where the term `slab` refers to the slab that contains the object to be freed. The term `c` represents the `kmem_cache_cpu` associated with this slab. The `slab` is either active (i.e., slab stored as `c->slab`), or stored in the partial slab list (i.e., slab located within `c->partial`) or node partial slab list (i.e., slab located within `n->partial`).

The Linux kernel incorporates cache aliasing to optimize memory management. Cache aliasing merges freed objects stored in distinct `kmem_caches` with similar characteristics, e.g., object size and allocation properties. For security reasons, `kmem_caches` of dedicated or generic caches considered security-critical are marked as accounted to prevent aliasing. Essentially, these accounted `kmem_caches` separate accounted objects from the non-accounted ones. Security-critical caches include those that store sensitive information, such as `cred`, and objects commonly used for exploitation (allocated using `kmalloc-cg-*`), such as elastic objects [5].

Architecture. The architecture of a `kmem_cache` [22], shown in Figure 1, includes a `kmem_cache_cpu` for each logical CPU and an array of `kmem_cache_nodes`. The `kmem_cache_cpu` comprises various free lists: a CPU free list (`c->freelist`), a slab free list (`slab->freelist`), and additional free lists of partial slabs (`partial->freelist`, maintained as a single-linked list). Despite each slab having its free list, the separate CPU free list allows lockless allocation, improving performance. The `kmem_cache_node` has a double-linked list of slabs (`partial`) also containing freed objects. In the context of this work, we refer to a list (i.e., free list, and single- and double-linked list) as full when it reaches its capacity of containing objects. It is considered empty when no object is present in it. A list is classified as partial when it is neither full nor empty.

Allocation and deallocation. `kmem_cache` stores objects in a multi-level free-list hierarchy. As shown in Figure 2, the allocation process starts by searching for an available object in the lower free-list levels [22]. This process continues throughout the hierarchy until an available object is found. These levels include the CPU free list ①, slab free list ②, CPU partial slab list ③, and node partial slab list ④, with each level taking more allocation time. If no object is available in any of these free lists, the SLUB allocator falls back to the buddy allocator ⑤, which allocates a memory chunk.

When deallocating, the SLUB allocator attempts to place the object in the lower free-list levels, e.g., CPU free list ① [22], as shown in Figure 3. Upon deallocation, the kernel may check the number of free slabs, i.e., the number of slabs with full free list stored in the node partial slab list ③. If this number exceeds a particular capability (see Table 4), the SLUB allocator deallocates the slab's memory chunks ⑤, returning them to the buddy allocator. Memory chunks returned in such a recycling phase are reused for future allocations.

Timing attacks on allocation. Lee et al. [26] demonstrated with PSPRAY the feasibility of performing a timing side channel on the SLUB allocator. PSPRAY deduces when the allocator allocates a fresh memory chunk (see ⑤ in Figure 2). This insight increases the likelihood of successful kernel heap exploitation, which primarily relied on heap spraying, i.e., for Use-After-Free (UAF) and Double-Free (DF), or heap grooming, i.e., for Out-Of-Bounds (OOB) [4, 26, 53].

However, their method relies on a precise measurement primitive that is no longer available in recent kernel versions. Their primary proposed primitive uses `msg_msg`. Since it is allocated via the segregated `kmalloc-cg-*` for kernel versions v5.14 or higher, it is limited to scenarios where the vulnerable

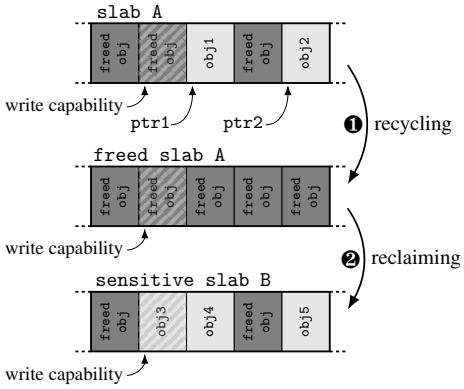


Figure 4: Software cross-cache attack with an initial state, where a write capability refers to a freed object. An attacker enforces a recycling ❶ of `slab A`'s memory chunk by freeing `obj1/2`. By allocating sensitive objects, the attacker presumably reclaims ❷ the chunk for a sensitive `slab B`, resulting in write capability referring `obj3`. Lastly, `obj3` is overwritten.

object is also allocated from the segregated generic cache. Other proposed primitives are limited because the computational overhead of non-allocation tasks primarily masks allocation timing (e.g., `read`). Furthermore, their approach fails to identify suitable measurement primitives, e.g., for `kmalloc-8/16`. For other identified primitives, we could not reproduce the allocation of a single data object (e.g., `fchown`), or the primitives are privileged and therefore unusable (e.g., `kexec_load`). We contacted the authors about the applicability of using their identified syscalls (apart from `msg_msg`) to determine the timing difference. They confirmed that the overhead of the syscalls they identified limits their applicability. In summary, while their work demonstrates feasibility, its applicability is limited to older kernel versions.

2.2 Software Cross-Cache Attacks

When the SLUB allocator frees memory chunks using the buddy allocator, as shown with ⑤ in Figure 3, these chunks are reused. Classic cross-cache attacks [2,17,19,28,29,47,50] exploit this reusing behavior. Initially, an adversary compels the SLUB allocator to recycle a memory chunk containing a write capability due to vulnerabilities. Subsequently, they allocate numerous sensitive objects from another allocator cache, hoping to reclaim the previously freed chunk. If successful, the memory that was previously occupied with the write capability will now be occupied by sensitive objects. Lastly, they trigger the write capability to this memory, corrupting a sensitive data object. The recycling ❶ and reclaiming ❷ phases are shown in a simplified setting in Figure 4.

Xu et al. [50] demonstrated the feasibility of cross-cache attacks, and subsequent research [28,29] has further explored their impact. However, executing such attacks is notably challenging, particularly for frequently used generic caches. One significant hurdle is the introduction of noise through uncontrolled allocations, complicating to achieve state ⑤ in Figure 3. For example, unknown allocations from a kernel thread can thwart the freeing of the slab's memory chunk, thereby preventing the reclamation [28]. Adding to the complexity, the unpredictable occurrence of both phases, recycling ❶ and reclaiming ❷, introduces instability to the exploit.

In summary, mounting cross-cache attacks is complex and fraught with challenges. Although these attacks are compelling, in practice, they have a limited success rate, as low as 40 % [50]. Importantly, this percentage only represents the success rate of the cross-cache attack, excluding additional stages of an end-to-end exploit, e.g., vulnerability triggering and memory manipulation, which further reduces the overall success rate. The process of repeatedly triggering vulnerabilities carries its risks. Traces left in the kernel often make it difficult to trigger the same vulnerability again. For instance, an OOB write may corrupt lists when triggered. Hence, repeated activation of the vulnerability can result in a crash, severely limiting the attack's practicality.

2.3 Threat Model

We assume that an unprivileged user has code execution. Additionally, we consider the presence of a heap vulnerability in the Linux kernel. We assume that the Linux kernel incorporates all defense mechanisms available in version 6.4, the most recent Linux kernel version when we started our work. These mechanisms include features such as `W^X`, KASLR, SMAP, and kCFI [37]. We do not assume any microarchitectural vulnerabilities, e.g., transient execution [24,31], fault injection [43], or hardware side channels [3,51].

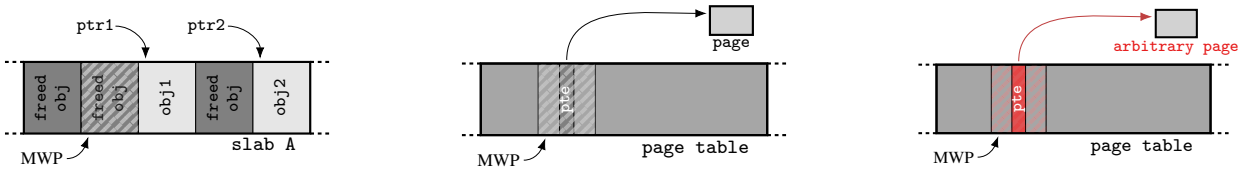
In this work, we primarily focus on heap vulnerabilities (most common type of software vulnerability according to Microsoft [36]) that result in a Double-Free (DF), or a Use-After-Free (UAF) or an Out-Of-Bounds (OOB) allowing for a limited writing capability. For instance, CVE-2023-21400 enables the double free of an object within the `kmalloc-32` generic cache, while CVE-2023-3609 permits a write operation at offset 0x18 on an object allocated from `kmalloc-64`.

3 Technical Overview and Challenges

This section outlines SLUBStick's capability to overcome several technical challenges when exploiting a limited heap vulnerability to obtain an arbitrary read-and-write primitive.

3.1 Overview

Obtaining an arbitrary read-and-write primitive with SLUBStick involves *three stages*, as depicted in Figure 5. In the *first stage* (see Figure 5a), SLUBStick exploits a heap vulnerability to acquire a Memory Write Primitive (MWP). This



(a) *Stage 1* deallocates objects `obj1` and `obj2` to trigger the recycling process of `slab A`'s memory chunk, with a Memory Write Primitive (MWP) referring to it.

(b) *Stage 2* reclaims the recycled memory chunk for a page table used by the userspace address translation, containing one entry, `pte`, which refers to the user-accessible `page`.

(c) *Stage 3* triggers the MWP to manipulate the page table entry `pte`. This manipulated entry indexes then the `arbitrary page`, allowing it to be overwritten from the userspace.

Figure 5: High-level overview of SLUBStick subdivided into three stages exploiting an MWP.

MWP provides an adversary with a write capability at an adversary-determined time, with the primitive referring to the memory of a freed object. Subsequently, SLUBStick triggers the recycling process of its slab's memory chunk by deallocating all objects of the slab. This chunk is then returned to the buddy allocator for future allocations. Crucially, even after the recycling, the MWP still refers to the recycled chunk.

In the *second stage* (see Figure 5b), SLUBStick allocates page tables to reclaim the recycled memory chunk for a page-table page used to translate a userspace address. The entries in this table refer to user-accessible pages, storing crucial information, e.g., their page frame number and access permissions.

In the *third stage* (see Figure 5c), SLUBStick triggers the MWP to overwrite the referenced memory of the page table. This manipulation allows an adversary to change the page frame number and permission bits, granting access to any physical page and adjusting user access permissions. As a result, the virtual address now refers to the selected page with permission to modify it from userspace. For example, by referring to kernel code, this capability allows the adversary to alter the kernel's behavior. As another example, the adversary can reference data of privileged files such as `/etc/passwd`, thus modifying it to bypass authentication checks.

SLUBStick does not violate the kernel's control flow or any of the existing kernel defenses outlined in Section 2.3. Hence, these existing kernel defenses cannot mitigate SLUBStick, highlighting its severe threat to current systems. Furthermore, efforts to separate generic caches [9] are ineffective in mitigating SLUBStick. We discuss the limitations of existing kernel defenses in Section 8.

3.2 Technical Challenges

This section briefly describes how SLUBStick overcomes several technical challenges to escalate privileges, while the subsequent sections discuss our solutions in more detail.

Triggering recycling/reclaiming. SLUBStick performs a cross-cache attack on generic caches. As described in Section 2.2, cross-cache attacks have been challenging to execute because they are unstable and unpredictable [28, 50]. Uncontrolled allocations and deallocations introduce noise, making it difficult to determine when the recycling and reclaiming phase will occur. However, we present a side-channel supported approach to reliably trigger the recycling process of a targeted memory chunk, which is then reclaimed. This significantly increases the success rate, exceeding 99 % for slabs with single page-size chunks and 82 % for multi-page-size chunks, which were previously considered impractical. In Section 4, we explain our novel approach, which exploits the side-channel timing leakage of the SLUB allocator.

Pivoting heap vulnerabilities. Pivoting a kernel heap vulnerability to establish an MWP presents two sub-challenges: First, SLUBStick obtains a dangling pointer from the vulnerability. DF vulnerabilities provide us with this capability, while OOB and UAF write vulnerabilities do not. Hence, SLUBStick requires an appropriate pivoting approach to utilize them fully. Second, in most cases, a dangling pointer does not directly allow for controlled overwriting with controlled timing. Hence, SLUBStick must pivot the dangling pointer, performing an adversary-controlled write, i.e., MWP. In Section 5, we explain the process of adequately pivoting capabilities from kernel heap vulnerabilities to obtain the MWP.

Arbitrary read-and-write. By triggering the MWP, SLUBStick can overwrite a page table used for address translations of a userspace address. As the kernel memory layout is randomized (for physical addresses) by KASLR, and we do not assume any address information leakage, pivoting an MWP to an arbitrary memory read-and-write primitive is not straightforward. In Section 6, we explain how we achieve this pivot to an arbitrary read-and-write primitive.

4 Triggering Recycling and Reclaiming

In this section, we present a novel approach that reliably triggers the recycling process of a targeted memory chunk and reclaims it, making cross-cache attacks practical for exploitation. Our approach leverages a software timing side-channel attack on the SLUB allocator. Through this side channel, we gain valuable information about when a memory chunk is allocated and deallocated using the buddy allocator. These insights allow us to reliably trigger the recycling process of a targeted chunk (see Section 4.1). To ensure the return of this

chunk during the reclamation phase, we massage the buddy allocator's internal state (see Section 4.2). By combining these two strategies, our novel approach triggers the recycling and reclaiming process reliably. To demonstrate the effectiveness, we provide a proof-of-concept (including experiments), offering comprehensive details on deploying generic caches from `kmalloc-8` to `kmalloc-4096` (see Section 4.3).

4.1 Side-Channel Supported Recycling

To trigger the recycling of a targeted memory chunk, our approach first groups allocated objects according to their slab by using a software timing side channel on the allocation. It then deallocates these grouped objects, prompting the kernel to recycle the slabs' chunks, including our targeted chunk.

The principal approach to grouping allocated objects involves timing measurements during standalone object allocation from userspace. A fast to medium timing indicates that the object was allocated from one of the slab's lists (①, ②, ③, or ④ in Figure 2). Hence, we group it with the previously allocated object in the same slab. Conversely, if the allocation time is notably higher (⑤), this indicates that the object originates from a newly allocated memory chunk and thus a new slab, which prompts us to group it separately. As a result, we obtain a list of grouped objects organized by their slabs.

One challenge with this approach is the lack of primitives in the Linux kernel to accurately measure standalone allocation times with minimal non-allocation tasks from userspace. This is because a single object allocation involves notable non-allocation tasks, e.g., allocating an object by opening its device. Using primitives with notable non-allocation tasks, as proposed by PSPRAY [26] for non-separated generic caches, leads to inaccurate and unusable results. To address this challenge, we separate the timing measurement from the persistent object allocation. While an allocation primitive persistently allocates an object, the timing measurement primitive is used to group this allocated object based on its slab.

Measurement primitive. We measure the timing of primitives that involve both allocation and deallocation within a single syscall while only performing minimal non-allocation tasks. During the deallocation process within this primitive, the previously allocated object is consistently set to the active CPU free list (state ① in Figure 3), which only minimally affects the measured timing. The minimal non-allocation tasks executed during the syscall exhibit high consistency, which also minimally affects the results. Thus, the measured timing depends primarily on the allocation. This allows us to determine when a new memory chunk is allocated from userspace, indicated by a notable slow allocation timing. To illustrate the notable difference between fast and slow allocation timings, we provide detailed experiments in Appendix B.

An example of a measurement primitive is the `add_key` syscall, shown in Listing 1. To ensure minimal interference from non-allocation tasks, our approach invokes the `add_key` syscall with a `_desc`, which fails the validation check at Line 8. As a result, the syscall primarily involves two privilege switches, allocation and deallocation, with the execution time mainly depending on the allocation. By measuring the timing of this syscall, our approach determines whether Line 4 allocates a new memory chunk. Crucially, while the `add_key` syscall is one example, measurement primitives are common in the Linux kernel, as we later illustrate in Section 4.3.

```
1  ssize_t __do_sys_add_key(const char __user *_desc) {
2    ssize_t ret;
3    size_t len = strnlen_user(_desc) + 1;
4    char *desc = kmalloc(len, GFP_KERNEL);
5    size_t n = copy_from_user(desc, _desc, len);
6    if (n) goto ERR;
7    desc[len - 1] = 0;
8    if (IS_INVALID(desc)) goto ERR;
9    ... /* add_key code execution */
10 ERR:
11   kfree(desc);
12   return ret;
13 }
```

Listing 1: The `add_key` syscall serves as a measurement primitive to determine whether Line 4 allocates a new chunk.

```
1  size_t timed_alloc(int *time) {
2    /* allocate the 64 byte struct snd_ctl_file */
3    int fd = open("/dev/snd/controlC0", O_RDONLY);
4    /* timed allocation with invalid add_key */
5    char _desc[64] = INVALID_DESC;
6    size_t t0 = rdtsc_begin();
7    add_key(_desc);
8    size_t t1 = rdtsc_end();
9    *time = t1 - t0;
10   /* return allocated object */
11   return fd;
12 }
```

Listing 2: An example of a 64 byte allocation from userspace, where Line 3 allocates a `snd_ctl_file` object. Between Lines 6 and 8, we use the `add_key` measurement primitive to determine whether a new chunk was allocated.

Persistent allocation primitive. A persistent allocation refers to a situation where the allocated kernel object remains allocated until the corresponding deallocation counterpart is executed. Crucially, the deallocation of the object must occur within the deallocation syscall, not at a later time. Therefore, objects released inside a Read-Copy-Update (RCU) [35] callback function cannot be used for persistent allocation. Also, the de-/allocation syscalls must be unprivileged.

Triggering the recycling process. Using our method of accurately measuring allocation timing and persistently allocating objects, our approach to reliably triggering the recycling process consists of *two stages*.

In the *first stage*, we allocate and group kernel objects based on their slabs. We start emptying all free lists within the cache's multi-level hierarchy by allocating many objects. Next, we persistently allocate objects to fill the free memory slots of the slab while using our measurement primitive to

exploit the timing side channel for grouping the allocated objects. Listing 2 shows an example of combining a measurement primitive (e.g., `add_key`) with a persistent allocation primitive (e.g., `snd_ctl_file`). The `timed_alloc` function provides two outputs: the measured timing at Line 9 and the allocated object as an identifier at Line 11. Using the measured `*time`, we determine whether the object belongs to the same slab as the previously allocated object (i.e., indicated by a low value) or whether it originates from a new memory chunk and hence a new slab (i.e., indicated by a notably high value). This allows us to group objects identified with `fd` by their slabs.

In the *second stage*, we proceed to free the grouped objects, prompting the Linux kernel to deallocate their slabs via the buddy allocator, as denoted by the state ⑤ in Figure 3. The release of slabs occurs when the number of free slabs exceeds the capacity of the node's partial slab list ③, detailed in Table 4. Since we know the objects of the slabs and the capacity of the partial slab list, we retain control over when the memory chunks, including our target, are recycled. While uncontrolled deallocations may introduce noise, we show remarkable noise resilience in our experiments in Section 4.3.

4.2 Massaging the Buddy Allocator

This section determines the reclaiming phase of a targeted memory chunk based on the recycling phase, where our approach is tailored to the generic cache size. A memory chunk can consist of a single or multiple pages, as shown in Table 4. Our goal is to ensure the reliable reclaiming of either the recycled chunk or a page-size part of it. Notably, the reclamation chunk's size is required to be one page, as SLUBStick reclaims it as a page table, as we later show in Section 6.

For generic caches from `kmalloc-8` to `kmalloc-256`, the memory chunk consists of a single page. When releasing slabs' memory chunks, the buddy allocator follows a Last In First Out (LIFO) principle to return recycled memory chunks. Therefore, to obtain a targeted chunk, one can request a page-size chunk shortly after the recycling process in the opposite order of the deallocation process.

In contrast, for generic caches from `kmalloc-512` to `kmalloc-4096`, the memory chunk consists of multiple pages. To ensure the reclaiming of the recycled memory chunk as page size, we need to put the allocator in a state where it splits the targeted memory chunk into smaller ones. To accomplish this, we take the following strategy: First, we empty smaller memory chunks, reducing the number of available smaller chunks. Next, we perform the recycling process, preparing for subsequent chunk splitting and reclaiming. Finally, we allocate multiple page-size chunks. This compels the kernel to split the targeted memory chunk into smaller ones, making it available for allocation. Through these steps, we ensure that the buddy allocator splits the targeted memory into smaller chunks and returns them for page-size allocation.

4.3 Proof-of-Concept and Experiments

For our Proof-Of-Concept (POC), we first systematically analyze the Linux kernel to identify kernel objects suitable for a measurement and allocation primitive. We then combine our approaches discussed in Sections 4.1 and 4.2 to reliably trigger the recycling and reclamation phase. Our POC demonstrates the feasibility and effectiveness of our approach with a success rate of between 99.3 % to 99.9 % for single page-size chunks and 82.1 % to 93.5 % for multi-page-size chunks.

Systematic analysis. Our approach requires a *measurement* (e.g., `add_key` in Listing 1) and an *allocation* (e.g., `snd_ctl_file` in Listing 2) primitive. We scan the kernel code for suitable code patterns using the query language CodeQL [15], as we explain in detail in Appendices C.2 and C.3.

Many kernel execution paths in the kernel allow *measurement* primitives, as any snippet that copies user data to a dynamically allocated buffer and subsequently performs validation checks can be leveraged. This includes functions that delegate the allocation and copying process to commonly used routines like `strndup_user` and `memdup_user`. Hence, our analysis yields a measurement primitive for each generic cache from `kmalloc-8` to `kmalloc-4096` (see Table 8).

Similarly, the kernel has many execution paths that allow persistent *allocation* primitives, as any unprivileged syscall that persistently allocates an object and has a freeing counterpart that instantly releases the object can be leveraged. As a result, our analysis also yields an allocation primitive for each of these generic caches (see Tables 6 and 7).

We conduct this analysis for Linux kernel versions v5.19 and v6.2. Our results, presented in Table 8 for measurement primitives and Tables 6 and 7 for allocation primitives, underline the high applicability of our approach across versions.

Experiments. We demonstrate the feasibility and effectiveness of our approach by showcasing how to reliably trigger the recycling process of a targeted memory chunk and reclaim it during the reclaiming phase. To perform this experiment, we use a read device driver `rdd`, shown in Listing 6. The driver provides object allocation (`ALLOC`), deallocation (`FREE`), and reading of the object's contents (`READ`). Allocation and deallocation are used to allocate and release an object on a target memory chunk while reading determines if the reclamation was successful. In this experiment, we use a page table as the object that reclaimed the recycled memory chunk. To allocate a page table, we map a page without all page table levels mapped. We consider the experiment successful if we can read the page table entries from our `rdd` after the reclaiming phase since this means that the page table successfully reclaimed our target memory chunk. The experiment is considered unsuccessful if we can not trigger the recycling process for the chunk containing the target object or if the recycled page is not reclaimed as a page table.

We ran this experiment 100 times to determine the success rate, which we repeated 10 times to compute the mean

Table 1: Success rate of triggering the recycling and reclamation process for generic caches.

Generic Cache	#Pages	Success Rate		
		Idle	No CPU pinning	External noise
		%	%	%
kmalloc-8	1	99.9 ± 0.1	99.9 ± 0.1	99.6 ± 0.7
kmalloc-16	1	99.4 ± 0.6	98.9 ± 1.2	99.9 ± 0.4
kmalloc-32	1	99.4 ± 0.9	99.7 ± 0.5	99.9 ± 0.3
kmalloc-64	1	99.2 ± 1.3	99.2 ± 0.9	81.0 ± 6.4
kmalloc-96	1	99.9 ± 0.4	99.9 ± 0.1	99.8 ± 0.6
kmalloc-128	1	99.9 ± 0.4	99.8 ± 0.5	99.9 ± 0.3
kmalloc-192	1	99.9 ± 0.4	99.8 ± 0.4	99.3 ± 1.2
kmalloc-256	1	99.9 ± 0.3	99.9 ± 0.3	99.7 ± 0.7
kmalloc-512	2	90.2 ± 5.4	87.2 ± 3.1	65.2 ± 2.8
kmalloc-1024	4	88.1 ± 7.2	79.5 ± 3.3	70.3 ± 8.1
kmalloc-2048	8	83.1 ± 9.2	70.5 ± 16	57.8 ± 5.7
kmalloc-4096	8	82.1 ± 3.4	73.3 ± 19	53.8 ± 10

and standard deviation. We tested under an idle system with *CPU pinning* and two noise conditions: *no CPU pinning* and external *noise*, demonstrating our noise resilience. Our experimental setup was Ubuntu 22.04 LTS with the generic Linux kernel v6.2 for x86, where v5.19 gives similar results. We ran this on a machine with Intel i7-1260P and 48 GB RAM.

CPU pinning is the method used to fix a process to a CPU, preventing CPU migration. Since both the slab and buddy allocator maintain per-CPU lists, CPU migration may introduce noise. To examine this effect, we included both scenarios: with and without CPU pinning. To introduce external *noise*, we ran `stress-ng` concurrently with two workers for each CPU, IO, and VM, saturating approximately 100 % usage on all CPUs. This aligns with prior noise evaluation [26].

Results. Under idle, our experiments (see Table 1) show remarkable results, with a more than 99.2 % success rate for generic caches with page-size chunks, i.e., `kmalloc-8` to `kmalloc-256`. The failure scenarios are primarily due to the targeted chunk that can not be reclaimed as a page table. The results of generic caches with multiple page-sized chunks decrease with increasing size from 90.2 % to 82.1 %. This is primarily due to the failure in reclaiming the targeted page from the split chunk. The larger the original chunk, the more error-prone the reclaiming is, as seen by the lowest success rate of about 83 % for generic caches with 8 pages.

Our results demonstrate remarkable noise resilience for almost all generic caches with page-size chunks with more than 98 % for no CPU pinning and external noise. One exception is `kmalloc-64` for the noise evaluation since `stress-ng` performs frequent allocation directly following deallocation for `kmalloc-64`. Hence, it appends a memory chunk to the node partial list, which makes it confusing to reclaim the memory chunk before the actual target chunk as the target chunk. For multiple page-sized chunks, no CPU pinning decreases the success rate to 70.5 %, while external noise decreases it to 53.8 %. Crucially, despite the induced noise, our approach is still notably better than the prior success rate of 40 % with no noise [50].

5 Pivoting Kernel Heap Vulnerabilities

SLUBStick leverages a kernel heap vulnerability to gain a Memory Write Primitive (MWP). This primitive provides an adversary with a write capability to previously freed memory at a controlled time. To achieve this, SLUBStick initially obtains a dangling pointer (see Section 5.1), i.e., a pointer referencing an already freed object from a heap vulnerability. SLUBStick then establishes an MWP from this dangling pointer (see Section 5.2). Crucially, we must extend the time window between converting the dangling pointer to an MWP (i.e., before the recycling) and when it is triggered (i.e., after the reclaiming). These steps provide the adversary with an MWP that can be triggered at the desired time.

5.1 Obtaining a Dangling Pointer

In this section, we pivot DF, and UAF and OOB write vulnerabilities to obtain a dangling pointer.

Double-Free. By exploiting a DF vulnerability and freeing an object twice, we induce a situation where the pointer to an object becomes dangling. However, freeing the same object twice will cause it to reside twice in the CPU free list, corrupting this free list and causing a system crash. Hence, to leverage a DF vulnerability for obtaining a dangling pointer, SLUBStick allocates an object after its initial freeing and then utilizes the subsequent double free to create a dangling pointer pointing to this newly allocated object. Reallocating the object prevents the duplicate free list entry, avoiding free-list corruption and a subsequent system crash.

Out-Of-Bounds and Use-After-Free write. While UAFs that allow write control over dangling pointers might work, they often fall short in practice. For direct use in SLUBStick, a UAF vulnerability would need specific criteria: First, they must retain overwriting capability post-zeroing during recycling. Second, they must be triggerable after reclaiming as a page table. Third, they require the ability to overwrite specific values, i.e., malicious page-table entries. However, most UAFs [2, 19, 40, 52] provide limited write primitives and small race windows, thus failing these criteria.

Hence, SLUBStick leverages significantly weaker write capabilities offered by UAF (and OOB) vulnerabilities commonly found in practice to enforce a double free of an object. It does so by using this capability to manipulate either a *reference counter* or an *object pointer* within an object of the same cache. By corrupting these members, SLUBStick forces an object into a DF state (see **Double-Free** above).

In Linux, *reference counters* manage the lifetime of an object. At its core, the counter increases with each new reference and decreases with each release. When the counter reaches zero, all the object's resources are released. For exploitation, SLUBStick identifies a victim object within the same cache containing a reference counter located at the write capability. Using heap manipulation [6, 26, 50], it aligns this victim

```
1  int ipmi_open(void) {
2    ipmi_file_private *priv;
3    /* allocate object */
4    priv = kmalloc(sizeof(*priv));
5  }
6  long ipmi_ioctl(file *f, u64 data) {
7    ipmi_file_private *priv;
8    ipmi_timing_parms parms;
9    /* copy data from user */
10   copy_from_user(&parms, data);
11   priv->parms = parms;
12 }
```

Listing 3: Persistent code pattern 1.

```
1  u64 netlink_sendmsg(msghdr *msg, u64
       len) {
2    /* allocate object */
3    sk_buff *skb = kmalloc(len);
4    /* copy data from user */
5    copy_from_user(obj, msg, len);
6  }
```

Listing 4: Persistent code pattern 2.

```
1  u64 keyctl_pkey_verify(void *uaddr, void *
       uaddr2, u64 size, u64 size2) {
2    /* allocate and copy data from user */
3    void *in = kmalloc(size);
4    copy_from_user(in, uaddr, size);
5    /* second copy for extending time window */
6    void *in2 = kmalloc(size2);
7    copy_from_user(in2, uaddr2, size2);
8    /* free obj */
9    kfree(in2);
10   kfree(in);
11 }
```

Listing 5: Temporal code pattern.

Figure 6: Examples of code patterns allowing to be exploited as a Memory Write Primitive (MWP).

object in memory with the write capability. SLUBStick then utilizes the write capability to corrupt the counter, forcing the release of the object still in use and placing it in a DF state.

To demonstrate the applicability of this approach, we analyze the kernel for the available objects with a reference counter located at each potential 8 byte write capability. We found 873 objects with a reference counter. They cover 100 % of overwrite locations for generic caches up to `kmalloc-32` and more than 80 % for generic caches between `kmalloc-64` and `kmalloc-256`. For caches between `kmalloc-512` and `kmalloc-4096` the availability ranges from 73.4 % to 2.7 %.

Regarding *pointer* corruption, SLUBStick identifies a victim object within the same cache that contains a pointer to an object from a separate cache. SLUBStick then places the victim object to align the overwrite capability and triggers it to corrupt the pointer by zeroing out the two least significant bytes. This creates an additional reference to another object in the separate cache. When the kernel releases this object, SLUBStick obtains a dangling pointer to the other object stored in the separate cache without leaking KASLR.

The uncorrupted pointer must refer to an object in a different slab. Otherwise, the slab that now contains a double-referenced object cannot be recycled, as the object that was originally pointed to has lost its reference and cannot be freed. To ensure that the uncorrupted pointer refers to an object in a different slab, SLUBStick employs our side channel (see Section 4.1). This technique allows for detecting and preventing failure scenarios, enhancing the success rate of pivoting.

Primitive conversion. The success rate of converting a UAF (including DF) vulnerability to a dangling pointer relies on the time window between the free and use stages. Techniques like ExpRace [27] can enhance exploitability by widening the time window. Additionally, UAF and OOB write vulnerabilities require an object with a reference counter or pointer located at the write capability.

5.2 Establishing a Memory Write Primitive

In this section, we detail how SLUBStick establishes a Memory Write Primitive (MWP) from a dangling pointer, allowing an adversary to overwrite memory reclaimed for a page table (see Section 6). The main challenge is to extend the time window from converting the dangling pointer into an MWP before recycling, while triggering must be done after the reclaim as a page table. To address this challenge, we exploit three distinct code patterns prevalent in the Linux kernel, as demonstrated in our **systematic analysis**.

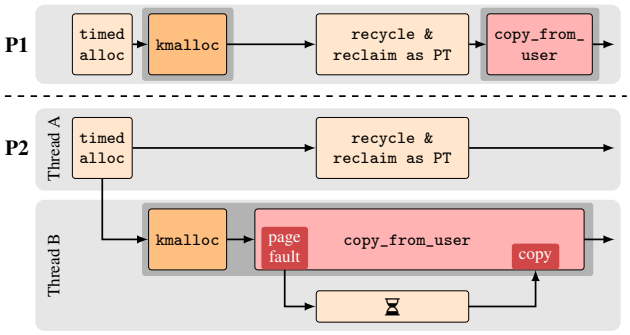


Figure 7: The execution timeline of persistent code patterns 1 (**P1**) and 2 (**P2**), where `timed alloc` performs persistent allocations combined with our timing side channel. `kmalloc` allocates the memory slot referenced by the dangling pointer. Since this slot resides on the page table after recycling and reclamation, `copy_from_user` overwrites page table entries.

Persistent code pattern 1. The pattern shown in Listing 3 grants SLUBStick an MWP with control over the timing. The process involves allocating the memory slot referred to by the dangling pointer, e.g., as an `ipmi_file_private` object at Line 4, before recycling. After reclamation, SLUBStick triggers the MWP with `ipmi_ioctl` to overwrite page table entries at Line 11. Figure 7 shows **P1** with its timeline.

While this pattern allows the MWP to be triggered arbitrarily, it has limitations as the allocated object, e.g., `ipmi_file_private`, is zeroed during recycling. If other code parts access the (unexpectedly) zeroed data, this could lead to a crash. Notably, this limitation does not affect `ipmi_file_private`, but others like `timerfd_ioctl`.

Persistent code pattern 2. The second pattern (see List-

ing 4) allows SLUBStick to trigger an MWP at a controlled time as follows: It allocates the memory slot referred to by the dangling pointer, e.g., as `sk_buff` at Line 3, before the recycling. Within the same syscall, the overwriting step at Line 5 follows after memory reclamation. To extend the time window between allocation and overwriting, SLUBStick can use techniques such as `userfaultfd`, Filesystem for USErspace (FUSE) [16], or slow page fault [20], controlling the duration of a user page fault, e.g., from of `msg` at Line 5.

While prior work [29, 40, 55] controlled the duration of copying from userspace with `userfaultfd`, recent systems restrict `userfaultfd` to privileged users for handling user page faults while in kernel mode [7]. Alternatively, researchers [16] showed that FUSE also allows control of the duration of copying from userspace. FUSE is a framework for userspace filesystems and can be utilized by an unprivileged user [16, 21]. We hence use FUSE to extend the time window between allocation and overwriting, allowing us to perform the recycling and reclaiming in between. Besides FUSE, SLUBStick can leverage the slow page fault approach [20].

We show **P2** with its execution timeline in Figure 7, where thread B's page fault starts before recycling and ends after reclaiming. The hourglass illustrates this time window extension, e.g., with FUSE. Hence, SLUBStick successfully overwrites entries in the reclaimed page table.

Temporal code pattern. Beyond persistent patterns, SLUBStick can also utilize temporal ones, as exemplified in Listing 5. This approach demands the management of two specific timing windows. The first ensures that object allocation at Line 3 and subsequent overwriting at Line 4 occur during the intended phases. The second window is crucial to avert premature object deallocation, leading to a potential failure scenario since the object is concurrently accessed as a page table. To extend these time windows, SLUBStick uses FUSE files as described for persistent pattern 2.

Systematic analysis. We use CodeQL [15] to identify suitable code patterns. We detail this approach in Appendix C.4. Our results for Linux kernel versions v5.19 and v6.2 are shown in Table 9. They indicate that persistent code pattern 1 is with 3 instances (and limited sizes) rare. For persistent code pattern 2, we identify 5 instances with sizes that cover generic caches from `kmalloc-8` to `kmalloc-4096`. A similar stands true for temporal code patterns with 7 identified. These findings underline the portability of the existing patterns.

6 Arbitrary Memory Read-and-Write

This section describes the *three-stage* process to obtain an arbitrary memory read-and-write primitive by triggering an MWP once, with an example shown in Figure 8. The single triggering of the MWP is crucial since both the persistent pattern 2 and the temporal pattern only allow a single trigger.

Initially, SLUBStick starts with a mapped userspace page using a Page Upper Directory (`PUD`) for address translation, with an MWP associated with it. The goal is to corrupt the content of a targeted page. Figure 8a shows an example where the userspace page utilizes the `cr3->pgde->pude->pmde->pte->page` translation, and SLUBStick aims to corrupt the targeted page containing data of `/etc/passwd`. While `/etc/passwd` is the target in this example, any page can be targeted, including kernel code.

In the *first stage* (Figure 8b), SLUBStick utilizes the MWP to partially overwrite the `PUD` page with `pude'` entries, each comprising a page frame number of zero and set user-accessible and size bit. As a result, userspace addresses using these entries for the page-table walk (`cr3->pgde->pude'`) are granted read and write access over the first physical GB.

In the *second stage* (Figure 8c), SLUBStick maps userspace pages, prompting the kernel to map page tables. It continues this process until a page table (`PT'` in the case of Figure 8c) is mapped in the first GB of physical memory. Given SLUBStick's control over reading and writing the first GB, the content of this page table `PT'` can be modified.

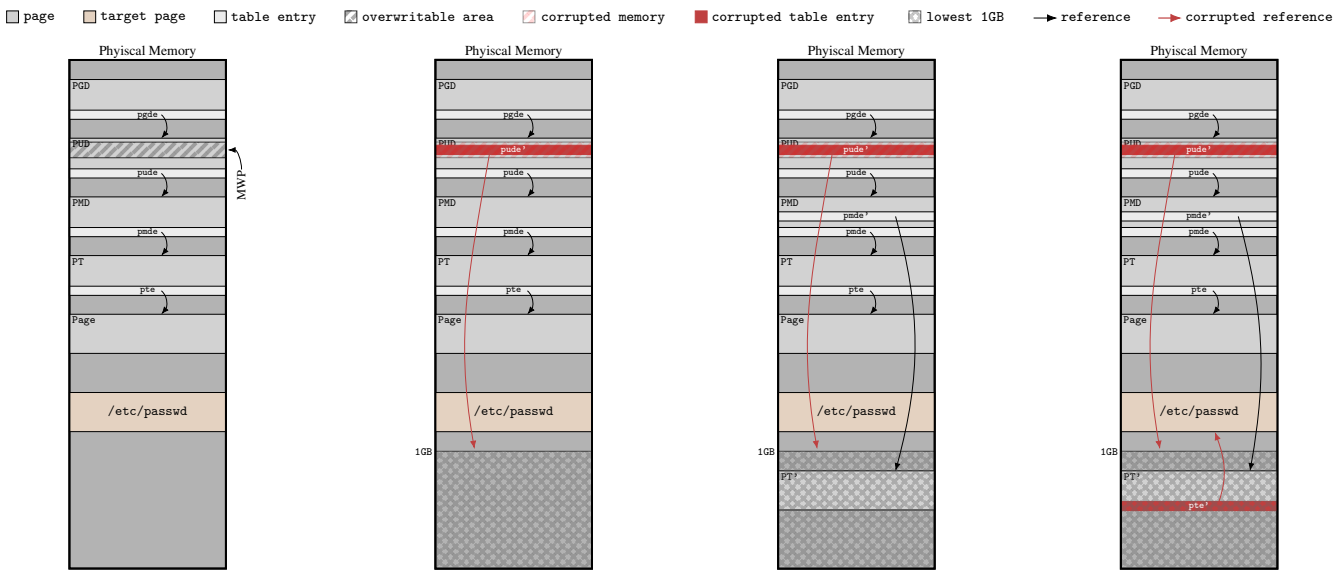
In the *third stage* (Figure 8d), SLUBStick overwrites an entry in the page table `PT'` with `pte'`. This grants an arbitrary physical memory read and write, using the userspace page with the `cr3->pgde->pude->pmde'->pte'` address translation. Since `kpti` is enabled by default on Ubuntu, SLUBStick uses an invalid syscall to flush this translation from the TLBs and force a page-table walk. Other methods, such as using a TLB eviction set referring to zero pages, would also work. With this primitive, SLUBStick can locate the targeted page and tamper with its content to escalate privileges, e.g., adding a privileged user with no authentication required.

This example uses the MWP to overwrite a PUD page but can also be used to overwrite a Page Middle Directory (PMD) page. By tampering with a PMD page, SLUBStick has an access space to find `PT'` of 2 MB instead of 1 GB.

In fact, we can use our MWP to overwrite not just a single page table entry but the entire memory slot previously used by the object. For instance, the function `netlink_sendmsg` allows an adversary to overwrite `len` (see Line 5 of Listing 4) bytes of the page table page. Thus, to increase the access space after the first stage to find `PT'`, SLUBStick overwrites the page table with entries of increasing page frame number. This approach increases the access space to $\frac{\texttt{sizeof(cache)}}{8} \cdot 1\,\text{GB}$ and $\frac{\texttt{sizeof(cache)}}{8} \cdot 2\,\text{MB}$ for PUD and PMD, respectively.

Reliability of obtaining an MWP to a PMD/PUD page. In order to reliably obtain an MWP primitive to a PMD or PUD page, SLUBStick employs the following strategies.

For generic caches from `kmalloc-8` to `kmalloc-256`, SLUBStick maps a single PUD page, as explained in Section 4.2, with an MWP associating the PUD, as follows: Exploiting the LIFO approach inherent to the buddy allocator for returning recycled memory chunks, SLUBStick initiates the mapping of a user page with an unmapped PUD, PMD, and PT. This procedure guarantees the reliable reclamation of the recycled memory chunk for the PUD page.



(a) SLUBStick has an MWP to a PUD page used for a user address translation. It aims to tamper with the targeted page containing data of /etc/passwd.

(b) SLUBStick leverages the MWP to write pude' (i.e., pfn=0, size=1, and user=1) to the PUD page. As a result, user addresses using pude' now reference the first GB of the physical memory.

(c) SLUBStick maps alot of page tables, till a page table (i.e., PT') is mapped within the first GB of the physical memory.

(d) SLUBStick uses first GB mapping to tamper with the entry pte' of page table PT'. As a result, the user address using pte' now references an arbitrary page, e.g., /etc/passwd.

Figure 8: Converting an MWP to an arbitrary read-and-write primitive to find and tamper with the targeted page /etc/passwd.

Conversely, achieving an MWP to a PMD page involves another strategy. For generic caches from kmalloc-512 to kmalloc-4096, where the memory chunk size exceeds a single page, SLUBStick initially prompts the kernel to split the recycled memory chunk. To reliably reclaim the split part of the memory chunk containing the MWP, SLUBStick employs an extensive mapping strategy for PMD pages, accomplished by mapping 2 MB huge pages. This extensive mapping tactic compels the buddy allocator to predominantly allocate PMD pages, including the page associated with the MWP. There are two options for using huge pages: Transparent Huge Pages (THP) or hugetlbfs. Albeit hugetlbfs offers more stable exploitation results, these huge pages must be pre-allocated. However, due to the widely spread application of hugetlbfs [25], such as databases [38], virtualization, and even Java [18], SLUBStick utilizes hugetlbfs.

7 Attack Evaluation

In this section, we evaluate the effectiveness of SLUBStick. We initially exploit a synthetic DF vulnerability (see Section 7.1) to show its applicability across generic caches from kmalloc-8 to kmalloc-4096. We then exploit 9 CVEs (see Section 7.2), consisting of DF, UAF, and OOB vulnerabilities.

Evaluation setup. For our experimental setup, we run the exploits in a virtual machine via QEMU 6.2.0, equipped with 4 cores and 16 GB RAM. We ran Ubuntu 22.04 LTS, with Linux kernels v5.19 and v6.2 for x86, and v6.2 for aarch64, to demonstrate architecture and version independence. These kernels were the unmodified generic ones, which presents a considerable challenge for exploitation.

7.1 Synthetic Vulnerability

To evaluate SLUBStick for all generic caches, we create a synthetic DF vulnerability in the Linux kernel, integrated into a module, i.e., read device driver rdd. Its simplified code is seen in Listing 6. For the exploitation processes, we need this driver code's allocation (ALLOC) and release (FREE) functionalities. We compile and then include this driver code during the operating system's startup for our three kernels.

With the synthetic DF vulnerability in place, we execute SLUBStick on all generic caches from kmalloc-8 to kmalloc-4096. For the measurement primitive, we use add_key with the invalid argument _descr (see Section 4.1). As allocation primitive, we utilize objects retrieved via our systematic analysis (see Section 5.2). For the MWP, we accomplish the exploitation with all three types, persistent 1 and 2 as well as temporal, using do_signalfd4, replace_user_tlv, and key_ctl_key_verify, respectively. Table 2 shows all primitives used for this evaluation. Equipped with these primitives, we successfully execute SLUBStick, exploiting the DF for privilege escalation.

Table 2: Primitives used for our attack evaluation.

Generic Cache	MP	AP	MWP
kmalloc-8	add_key	signalfd_ctx	do_signalfd4
kmalloc-16	add_key	aa_revision	key_ctl_key_verfify
kmalloc-32	add_key	anon_vma_name	key_ctl_key_verfify
kmalloc-64	add_key	snd_ctl_file	key_ctl_key_verfify
kmalloc-96	add_key	vfio_container	key_ctl_key_verfify
kmalloc-128	add_key	dlm_user_proc	key_ctl_key_verfify
kmalloc-192	add_key	pp_struct	key_ctl_key_verfify
kmalloc-256	add_key	snd_compr_file	replace_user_tlv
kmalloc-512	add_key	tls_context	replace_user_tlv
kmalloc-1024	add_key	pipe_buffer	replace_user_tlv
kmalloc-2048	add_key	key.description	replace_user_tlv
kmalloc-4096	add_key	net_device	replace_user_tlv

MP: Measurement Primitive **AP**: Allocation Primitive **MWP**: Memory Write Primitive.

Table 3: Exploitability shown on real-world vulnerabilities.

CVE	Capability	Cache
CVE-2023-21400	DF	kmalloc-32
CVE-2023-3609	UAF	kmalloc-96
CVE-2022-32250	UAF	kmalloc-64
CVE-2022-29582	UAF	files_cachep
CVE-2022-27666	OOB	kmalloc-4096
CVE-2022-2588	DF	kmalloc-192
CVE-2022-0995	OOB	kmalloc-96
CVE-2021-4157	OOB	kmalloc-64
CVE-2021-3492	DF	kmalloc-4096

7.2 Real-World Vulnerabilities

In line with prior research [6,29,46,49,53], we evaluate SLUBStick with a selection of real-world vulnerabilities. We port these vulnerabilities to our kernels to demonstrate the version, architecture, and kernel binary independence of SLUBStick. Our evaluation includes 9 vulnerabilities that are systematically classified based on the capability they provide. If the root cause of a vulnerability is a race condition that allows an adversary to derive a kernel heap vulnerability, we classify it as a heap vulnerability. For instance, since both race conditions, CVE-2023-21400 and CVE-2022-29582, can be pivoted to a UAF and DF, respectively, we classify them as a UAF and DF. In total, our selection consists of 3 DF vulnerabilities that enable SLUBStick directly, as well as 3 OOB and 3 UAF that we pivot to achieve a DF state.

The feasibility of pivoting an OOB and UAF depends on the write capability provided. We demonstrate with our results that the capability to overwrite as low as two bytes is enough to pivot to a DF state. Examples include corrupting a reference counter (as seen in CVE-2022-29582 and CVE-2023-3609) or nullifying the two least significant bytes of a pointer (like in CVE-2022-27666). For pivoting, SLUBStick requires an exploitable object containing a data pointer or reference counter at the overwrite point. The purely manual identification of suitable objects is time-consuming due to the extensive nature of the Linux kernel. For this reason, we utilize CodeQL to assist in identifying suitable objects for OOB and UAF write vulnerabilities, described in Appendix C.1.

Exploitability results. Table 3 shows the exploitability of SLUBStick, showcasing its functionality across kernel heap vulnerabilities and cache sizes. Specifically, we can free objects twice for the CVE2023-21400, CVE-2022-2588, and CVE-2021-3492 vulnerabilities, allowing SLUBStick to exploit these vulnerabilities and directly escalate privileges.

Additionally, the CVE-2022-0995 vulnerability allows overwriting out-of-bounds originating from the `watch_queue` object. We exploit this write capability to corrupt the reference counter of an `anon_vma_name` object, both located within the same generic cache, i.e., `kmalloc-96`. With the corrupted reference, we create a DF state of an `anon_vma_name` object, which enables the execution of SLUBStick. For CVE-2022-27666, the OOB allows overwriting memory adjacent to the `page_frag` object (allocated from generic cache `kmalloc-4096`). We perform a cross-cache overflow to zero the `next` pointer's two least significant bytes of a `msg_msg` object positioned on the adjacent memory chunk. This allows us to free the `next` twice so we can execute SLUBStick. As a third OOB write vulnerability, CVE-2021-4157, we exploit the overwriting capability within `kmalloc-64` to tamper with the reference counter `kref` of an `eventfd_ctx` object. As a result, we can pivot this to a DF and perform SLUBStick.

Furthermore, the CVE-2022-29582 vulnerability allows a `file` UAF, specifically using the `file` object after it has been invalidly freed. By strategically pivoting this vulnerability using cross-cache, we can reclaim the memory chunk for the generic cache `kmalloc-256`. Subsequently, we reclaim the invalidly freed `file` object to corrupt its reference counter. This allows us to free the `file` a second time, creating a DF state to run SLUBStick. Similarly, CVE-2023-3609 allows to use a `tc_u_hnode` object after it has been freed. Reclaiming the memory within the same generic cache and overwriting the reference counter, we pivot this vulnerability to a DF state of `tc_u_hnode` to perform SLUBStick. Lastly, the CVE-2022-32250 vulnerability provides a write capability within the generic cache `kmalloc-64`, which we use to corrupt a data pointer within `fdtable` for a DF state.

8 Discussion

Impact on existing attacks. Our side-channel supported approach presented in Section 4 greatly enhances the reliability of cross-cache attacks from generic caches and makes them practical for exploitation. Thus, it amplifies the effectiveness of exploitation methods employing cross-cache attacks.

For instance, DirtyCred [29] relies on the cross-cache approach to exploit DF vulnerabilities. This exploitation entails pivoting DF vulnerabilities to trigger an invalid free on a credential object. With our enhancement of the cross-cache attack, DirtyCred's success rate in pivoting DF vulnerabilities from generic caches also significantly increases, making DirtyCred an even more significant threat to kernel security.

Moreover, SLUBStick is more versatile than DirtyCred

as it does not rely on an invalid free on a credential object. Instead, an invalid free operation on any object is sufficient.

Container escape. In addition to privilege escalation, our research demonstrates that SLUBStick can directly enable container escapes such as Docker. For containers permitting FUSE, SLUBStick uses FUSE to extend the time window in persistent code pattern 2 and the temporal pattern. For (hardened) containers prohibiting FUSE, SLUBStick uses persistent code pattern 1 or the slow page fault approach [20] instead of FUSE. It then modifies kernel code accessible from userspace via a syscall, i.e., `sys_setresuid`.

Noise. With SLUBStick, we present a reliable approach for privilege escalation, leveraging a software timing side channel on the SLUB's object allocation. To maximize the reliability of SLUBStick, we minimize noise as follows:

CPU migration, the process of moving a task to another CPU, can introduce instabilities as both the slab and buddy allocator maintain per-CPU lists. To mitigate migration and, hence, increase reliability, SLUBStick may pin all running processes used in exploitation to a single CPU. However, it is not essential as illustrated in Table 1.

Due to the preemptive nature of the Linux kernel, preemption may occur during the timing measurement of a measurement primitive, potentially corrupting the result. To limit this noise source, SLUBStick performs timing measurements and keeps track of the number of objects associated with the current slab. If SLUBStick observes slow timing, indicating the allocation of a new memory chunk, it validates this by checking the object number of the current slab. This validation prevents misinterpretation of slow allocations, ensuring that previous and current objects reside in the same slab.

Manual vs automated components. We use the CodeQL [15] static analyzer to generate a list of possible objects or primitives automatically. Through manual inspection, we then refine this list by selecting suitable candidates. To provide a representation of the effectiveness of our automated part, our evaluated kernel contains 66787 structures, where our automated tool outputs 8601 potential as allocation and memory write primitives. After manual inspection, we get 36 suitable structures for allocation primitives. For memory write primitives, the tool outputs 2046 copy locations, where manual inspection yields 15 memory write primitives. For measurement primitives, the tool outputs 195 primitives, with 14 after inspection. Tables 6 to 9 show our findings. Further details on this process are provided in Appendix C.

Cross version/architecture dependencies. Kernel exploitation often depends on the specific kernel version and system architecture, as various exploitation techniques are closely tied to these factors. For example, KASLR is bypassed either with a read primitive [5] or a hardware side channel [3]. Read primitives are closely tied to specific kernel versions and configurations, while hardware side channels depend on the system architecture. Additionally, numerous exploits rely on constructing ROP chains [48,54], which involves a detailed inspection of the kernel binary, a process connected with system architecture, Linux kernel version, and configuration. SLUBStick distinguishes itself by not relying on bypassing KASLR or constructing a ROP chain, nor does it use architecture-specific data. As a result, SLUBStick is resilient to variations in kernel versions and architecture dependencies.

Kernel defenses. The last decade has seen a surge of proposals to improve kernel security. We briefly discuss the defenses, particularly considering SLUBStick.

KASLR is designed to enhance kernel security by randomizing the memory layout, making it more challenging to exploit vulnerabilities to perform Code Reuse Attacks (CRA) and data-only attacks. To further complicate CRA, researchers have introduced Control-Flow Integrity (CFI) [1], which has been adapted to kernel software [10, 14, 32, 45]. CFI helps ensure the integrity of the kernel control flow by restricting it to an approximated control-flow graph. Moreover, the Linux kernel has incorporated various hardening strategies to make the allocator more resistant to memory corruption vulnerabilities. These include slab list randomization, protection of slab meta-data, and slab quarantine [39]. However, researchers have identified bypassing attacks [26, 39, 53] for these hardening strategies. Additionally, heap separation is a defense integrated into the Linux kernel to separate caches containing security-critical data, e.g., `cred`, or objects often used for exploitation [29, 40, 55], e.g., `msg_msg`. To further enhance the separation of kernel objects, researchers [9] have proposed to increase the separation granularity. This involves randomly assigning each allocation site to one of these separated caches, making heap spraying attacks more challenging. However, SLUBStick leverages common code patterns, allowing it to circumvent the separation attempt of generic caches. Going a step further, our timing side channel on the allocator can even be utilized to determine whether two allocation sites share the same cache. In summary, the described countermeasures, while valuable, appear ineffective in mitigating SLUBStick.

AUTOSLAB [28] is a defense provided by grsecurity via paid subscription. Since AUTOSLAB separates allocator caches based on their allocation types, it restricts that different allocation types share the same allocation cache. However, this granularity is still too coarse-grained, as, e.g., the type `char *` is used by multiple allocation, measurement, and memory write primitives. As a result, the allocation cache with type `char *` is still exploitable with SLUBStick.

SLAB_VIRTUAL [42], which is currently under development, is designed to mitigate cross-cache attacks by allocating kernel objects via virtual addresses rather than direct physical memory. However, Torvalds and Molnar noted that it has drawbacks like significant overhead and incompatibility with DMA [42]. If merged, DMA-allocated memory will most likely be excluded [42]. This leaves more than 350 allocation sites unprotected, exploitable to obtain all necessary code patterns (allocation, measurement, and memory write primitives, such as `sun8i_ce_aes_setkey`, `monwrite_new_hdr`,

and `sti_hqvdp_vtg_cb`). Hence, SLUBStick can exploit a vulnerability, e.g., CVE-2023-2194, of a DMA-allocated object. Since DMA-allocated memory is mainly used in drivers, known to be vulnerable [34], SLUBStick still poses a threat.

Lastly, prior academic works [11, 33, 41, 44] proposed defenses to protect page tables to mitigate data-only attacks.

While existing defenses such as AUTOSLAB and SLAB_VIRTUAL demonstrate promise in mitigating SLUBStick, none provide comprehensive protection. This underscores the threat SLUBStick poses to kernel security.

Public exploits. Recently, multiple non-academic exploits have been proposed [2, 12, 13, 17, 19, 30, 47, 52]. Some [2, 17, 19, 47] leverage cross-cache attacks using the dedicated `file` cache. In contrast, we focused on generic caches, which are considerably more challenging to exploit. This is primarily because generic caches have numerous allocation sites, resulting in significant allocation noise. Specifically, while the dedicated `file` cache has one allocation site (`dup_f`), generic caches have from 354 (`kmalloc-4096`) to 2250 (`kmalloc-64`) on systems like our evaluated Ubuntu. Hence, we proposed our side-channel supported approach to perform cross-cache attacks on generic caches reliably.

Two exploits, bad io_uring [30] and the exploit demonstrated by Wu et al. [47], highlight the exploitation of cross-cache attacks on older kernel versions (i.e., v5.10), targeting Android devices. Bad io_uring leverages a cross-cache attack to misuse `pipe_buffer` as an arbitrary read and write. Wu et al. (concurrently to our work) demonstrated page-table manipulation by misusing `signalfd_ctx`. Both exploits rely on stabilization objects in conjunction with multiple retriggers of cross-cache reuse. Specifically, bad io_uring utilized additional `pipe_buffer` objects for stabilization, while Wu et al. relied on `seq_operations`. However, kernel advances beyond v5.14 introduced heap separation (i.e., `kmalloc-cg-*`), rendering the reuse of these stabilization objects ineffective for exploitation. Consequently, these exploitation strategies are thwarted by newer kernel versions. In contrast, SLUBStick exploits more recent systems, including v5.19 and v6.2, for a wide variety of heap vulnerabilities.

9 Conclusion

This paper presented SLUBStick, a novel kernel exploit technique that enables arbitrary memory read-and-write primitives through a practical software cross-cache attack. For our cross-cache attack, we used a software timing side channel on the SLUB allocator, significantly enhancing the success rate for frequently used generic caches to over 99 %. Moreover, using page-table manipulation, SLUBStick effectively converts a limited kernel heap vulnerability into arbitrary read-and-write capabilities. We demonstrated privilege escalation in the Linux kernel using a synthetic vulnerability and 9 real-world CVEs, showcasing its serious threat.

Acknowledgements

We thank Daniel Gruss, the anonymous reviewers, and our shepherd for their valuable feedback. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE and AWARE project (FFG grant numbers 888087 and 891092), the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *CCS*, 2005.

[2] 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/.

[3] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*, 2020.

[4] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *USENIX Security*, 2020.

[5] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A Systematic Study of Elastic Objects in Kernel Exploitation. In *CCS*, 2020.

[6] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *CCS*, 2019.

[7] Jonathan Corbet. Blocking userfaultfd() kernelfault handling, 5 2020. URL: https://lwn.net/Articles/819834/.

[8] Jonathan Corbet. A slab allocator (removal) update, 5 2023. URL: https://lwn.net/Articles/932201/.

[9] Jonathan Corbet. Randomness for kmalloc(), 7 2023. URL: https://lwn.net/Articles/938637/.

[10] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *S&P*, 2014.

[11] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *NDSS*, 2017.

[12] Devil. CoRJail: From Null Byte Overflow To Docker Escape Exploiting poll_list Objects In The Linux Kernel, 2022. URL: https://syst3mfailure.io/corjail/.

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

[14] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. *RAID*, 2023.

[15] GitHub. CodeQL, 2021. URL: https://codeql.github.com/.

[16] Luke Gix. FUSE for Linux Exploitation 101, 2022. URL: https://exploiter.dev/blog/2022/FUSE-exploit.html.

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

[18] IBM. Working with hugetlbfs huge-page support, 2023. URL: https://www.ibm.com/docs/en/linux-on-z?topic=hps-working-huge-pages-3.

[19] 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/.

[20] Choo Yi Kai. A new method for container escape using file-based DirtyCred, 2023. URL: https://starlabs.sg/blog/2023/07-a-new-method-for-container-escape-using-file-based-dirtycred/.

[21] The Linux Kernel. FUSE, 2023. URL: https://www.kernel.org/doc/html/next/filesystems/fuse.html.

[22] Imran Khan. Linux SLUB Allocator Internals and Debugging, 2022. URL: https://blogs.oracle.com/linux/post/linux-slub-allocator-internals-and-debugging-1.

[23] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 1965.

[24] 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.

[25] Mike Kravetz. hugetlbfs: Not just for databases anymore!, 2015. URL: https://blogs.oracle.com/linux/post/hugetlbfs-not-just-for-databases-anymore/.

[26] 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.

[27] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *USENIX Security*, 2021.

[28] Zhenpeng Lin. How AUTOSLAB Changes the Memory Unsafety Game, 2021. URL: https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game.

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

[30] 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.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[32] Lukas Maar, Pascal Nasahl, and Stefan Mangard. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In *AsiaCCS*, 2024.

[33] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In *ACSAC*, 2023.

[34] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In *NDSS*, 2022.

[35] Paul McKenney. What is RCU, Fundamentally?, 12 2007. URL: https://lwn.net/Articles/262464/.

[36] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. *Bluehat IL*, 2019.

[37] Joao Moreira. Kernel FineIBT Support, 4 2022. URL: https://lwn.net/Articles/891976/.

[38] MySQL. Enabling Large Page Support, 2023. URL: https://dev.mysql.com/doc/refman/8.0/en/large-page-support.html.

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

[40] Alexander Popov. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel, 2021. URL: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.

[41] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *S&P*, 2020.

[42] 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/.

[43] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. In *Black Hat USA*, 2015.

[44] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *NDSS*, 2016.

[45] Yoo Sungbae, Park Jinbum, Kim Seolheui, Kim Yeji, and Kim Taesoo. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *USENIX Security*, 2022.

[46] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects. In *USENIX Security*, 2023.

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

[48] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *USENIX Security*, 2019.

[49] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *USENIX Security*, 2018.

[50] 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.

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

[52] Wang Yong. KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features, 2018. URL: https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf.

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

[54] 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.

[55] Xiaochen Zou and Zhiyun Qian. Exploit esp6 modules in Linux kernel, 2022. URL: https://etenal.me/archives/1825.

Table 4: Detailed information of each generic cache, where ✶ denotes the number of slabs until the node partial slab list is full, prompting to discard the slabs' memory chunks.

Generic Cache	Memory Chunk	Number of Pages	Number of Objects	Node Partial Slab List Capacity✶
kmalloc-8	4096	1	512	6
kmalloc-16	4096	1	256	6
kmalloc-32	4096	1	128	6
kmalloc-64	4096	1	64	8
kmalloc-96	4096	1	42	12
kmalloc-128	4096	1	32	8
kmalloc-192	4096	1	21	12
kmalloc-256	4096	1	16	7
kmalloc-512	8192	2	16	6
kmalloc-1024	16384	4	16	6
kmalloc-2048	32768	8	16	6
kmalloc-4096	32768	8	8	6

A Generic Cache Information

To trigger the recycling process reliably, we provide insights (see Table 4) into how a generic cache manages memory chunks and stores available objects. Generic caches from `kmalloc-8` to `kmalloc-256` use one page per slab, with the slab storing 512 to 16 objects per memory chunk. Caches larger than `kmalloc-256` use multiple pages per slab, storing between 16 and 8 objects per chunk. The **Node Partial Slab List Capacity** column indicates when the SLUB allocator releases the chunks of the slabs, e.g., for `kmalloc-256`, if this cache reaches 6 free slabs, the SLUB allocator prompts the buddy allocator to discard and recycle the slab's chunks.

Table 5: Measured timing in timestamps for fast ① and slow ⑤ allocation (see Figure 2) using the measurement primitive `add_key` with an invalid `_descr` argument.

Generic Cache	Fast Allocation ①	Slow Allocation ⑤
kmalloc-8	1086 ± 53	> 4401
kmalloc-16	1114 ± 69	> 3224
kmalloc-32	1133 ± 58	> 2510
kmalloc-64	1130 ± 69	> 2468
kmalloc-96	1117 ± 44	> 2169
kmalloc-128	1250 ± 126	> 2470
kmalloc-192	1197 ± 86	> 2121
kmalloc-256	1359 ± 34	> 2439
kmalloc-512	1685 ± 84	> 3759
kmalloc-1024	1601 ± 30	> 3589
kmalloc-2048	1561 ± 23	> 3415
kmalloc-4096	1925 ± 33	> 3758

```c
typedef struct { size_t uaddr, size; } msg_t;
void *obj;
long rd_ioctl(struct file *_, unsigned num, size_t param) {
  msg_t msg;
  copy_from_user(&msg, param, sizeof(msg_t));
  switch (num) {
    case ALLOC:
      obj = kmalloc(msg->size);
      return 0;
    case FREE:
      kfree(obj);
      return 0;
    case READ:
      copy_to_user(msg->uaddr, obj, msg->size);
      return 0;
    default:
      return -1;
  }
}
```

Listing 6: Read device driver `rdd`, supporting an allocation (`ALLOC`), deallocation (`FREE`), and read (`READ`).

B Timings of Measurement Primitives

We perform experiments to verify that the slow allocation time ⑤ is higher than the fast allocation time ① (see Figure 2). We use `add_key` as the measurement primitive with an invalid `_descr` argument to fulfill the constraints described in Section 4.1, while others (e.g., `mount` with an invalid `dev_name` address) yield similar results. Our experimental environment is Ubuntu 22.04 LTS with a Linux kernel v6.2, running on a machine with Intel i7-1260P and 48 GB RAM. We allocate 16384 objects as a warm-up to ensure that the measured timing of the subsequent allocations is either from the CPU free list ① or from a new memory chunk using the buddy allocator ⑤. We then perform 4096 allocations and distinguish them between **Fast** and **Slow Allocation**. As shown in Table 5, the results demonstrate a notable timing difference between these allocation paths, with an average fast allocation from 1086 to 1925 timestamps and a minimum slow allocation from 4401 to 3758.

Table 6: Allocation primitives allocating a single fixed-size object during the syscall, with ✶ new objects identified.

Generic Cache	Object	Constraints
kmalloc-8	pci_filp_private✶ signalfd_ctx	
kmalloc-16	afs_file✶ aa_revision✶	
kmalloc-32	vmci_host_dev✶ seq_operations coda_file_info✶ shm_file_data	cg cache
kmalloc-64	snd_info_private_data✶ snd_ctl_file✶	
kmalloc-96	subprocess_info watch_queue vfio_container✶	
kmalloc-128	dlm_user_proc✶	
kmalloc-192	loopback_pcm✶ snd_timer_user✶ pp_struct✶	
kmalloc-256	vhci_data✶ snd_compr_file✶ msg_queue	cg cache
kmalloc-512	tls_context mousedev_client✶	input group
kmalloc-1024	pipe_buffer tty_struct sock xfrm_policy nouveau_cli✶	
kmalloc-2048	super_block perf_event✶	SELinux disabled
kmalloc-4096	net_device	

Table 7: Allocation primitives allocating a single variable-size object during the syscall, with ✶ new objects identified.

Elastic Object	Generic Caches	Constraints
user_key_payload	kmalloc-[32,32767)	only 200 allocation
anon_vma_name✶	kmalloc-[8,96)	
msg_msg	kmalloc-[64,4096)	cg cache
msg_msgseg	kmalloc-[8,4096)	cg cache
drm_property_blob	kmalloc-[96,INT_MAX)	
key.description	kmalloc-[8,4096)	

C Systematic Analysis Detailed

In this section, we present our systematic analysis in more detail. Our principal approach is first to identify a list of possible objects or primitives using the CodeQL [15] static analyzer. We then use these results to manually identify suitable results or discard those results that violate the constraints of the respective primitive. For each primitive, our systematic analysis results in a comprehensive list of suitable objects and functions, demonstrating the effectiveness of our approach.

CodeQL aims to find code patterns that cause vulnerabilities in software. At its core, it creates a database where essential meta-information about the examined software is stored. With this database, CodeQL uses queries as input to analyze the software and interpret the query results. We retrofit CodeQL to find allocation and measurement primitives (see Section 4.1). We also use CodeQL to find suitable victim objects for UAF and OOB write vulnerabilities (see Section 5.1) and suitable code patterns to exploit for an MWP

Table 8: Measurement primitives, where ✶ denotes allocation via the separated `kmalloc-cg-*` caches and ☆ denotes depending allocation size whether `msg_msg`/`msg_msgseg` is allocated.

Syscall	File	Argument	Allocation Size	Condition
add_key	security/keys/keyctl.c	_descr	[1,4096)	*(char *)_descr = '.'
request_key	security/keys/keyctl.c	_descr	[1,4096)	_callout_info bad address
keyctl$KEYCTL_JOIN_SESSION_KEYRING	security/keys/keyctl.c	arg2	[1,4096)	*(char *)arg2 = '.'
keyctl$KEYCTL_SEARCH	security/keys/keyctl.c	arg4	[1,4096)	arg2 invalid keyid
keyctl$KEYCTL_PKEY_QUERY	security/keys/keyctl.c	arg5	[1,4096)	arg2 invalid keyid
mount	fs/namespace.c	type	[1,4096)	dev_name bad address
fsopen	fs/fsopen.c	_fs_name	[1,4096)	_fs_name not existing
fsl_hv_ioctl$FSL_HV_IOCTL_SETPROP	drivers/virt/fsl_hypervisor.c	*(size_t *)arg	[1,4096)	*((size_t *)arg+1) bad address
perf_ioctl$PERF_EVENT_IOC_SET_FILTER	kernel/events/core.c	arg	[1,4096)	!has_addr_filter(event)
joydev_ioctl_common$JSIOCSAXMAP	drivers/input/joydev.c	argp	[64,UINT64_MAX)	*(char *)argp > 0x3f
fsconfig$FSCONFIG_SET_FD	fs/fsopen.c	_key	[1,256)	fget(aux) not existing
prctlPR_SET_VMAPR_SET_VMA_ANON_NAME	kernel/sys.c	arg5	[1,80)	*(char *)arg5 = 1
io_uring_register$IORING_REGISTER_RESTRICTIONS	fs/io_uring.c	arg	[16,UINT32_MAX)	*(short *)arg = 5
msgsnd$alloc_msg✶	ipc/msgutil.c	len	[64/8☆,4096)	mtext bad address

Table 9: Code patterns allowing for an MWP, where ✶ denotes the same function for allocation and triggering the MWP of the persistent object, and ☆ and ✢ denote distinct function for allocation and triggering the MWP.

Function	Type	Size	Constraints
ipmi_open☆/ipmi_ioctl✢	P1	8	
do_signalfd4✶	P1	8	
joydev_ioctl✶	P1	5680	
replace_user_tlv	P2	[1,131072)	
atmel_ioctl	P2	[1,32)	CAP_NET_ADMIN
netlink_sendmsg	P2	[1,INT_MAX)	
tun_sendmsg	P2	[1,INT_MAX)	
tap_sendmsg	P2	[1,INT_MAX)	
mount	T	[1,4096)	
key_ctl_key_verfify	T	[1,256)	
mtdchar_writeoob	T	[1,4096)	
mmc_blk_ioctl_copy_from_user	T	96	
ptp_ioctl	T	1216	
__cld_pipe_inprogress_downcall	T	[1,65536)	
__bpf_copy_key	T	[1,512)	bpf as unprivileged

P1/2: Persistent code pattern 1/2 T: Temporal code pattern.

(see Section 5.2).

C.1 Finding Suitable Objects to Pivot UAF and OOB Writes

To find suitable objects for pivoting UAF and OOB vulnerabilities with an overwriting capability, we need a victim object with a pointer to a dynamically allocated object or a reference counter at the location of the overwriting location. In the example of CVE-2022-32250, the UAF write vulnerability provides overwriting at offset 0x18 for objects allocated from the generic cache `kmalloc-64`. With these constraints, we use a CodeQL query to help find dynamically allocated objects with either a pointer or reference counter at an offset of 0x18. We find the `fdtable` object that satisfies these constraints with the pointer `open_fds` at an offset 0x18.

Our query takes the overwrite offset and object size as input and returns all matching objects available in the Linux kernel with default kernel configurations. From these possible objects, we manually identify a suitable object accessible from userspace as an unprivileged user, e.g., the `fdtable` for the CVE-2022-32250 vulnerability.

C.2 Finding Allocation Primitives

For allocation primitives, our CodeQL query identifies every persistent object allocation code location grouped by object size. From this list, we manually filter out those inaccessible from userspace as unprivileged users. The result is a list of fixed and variable-size objects, shown in Tables 6 and 7.

C.3 Finding Measurement Primitives

For measurement primitives, any code snippet that copies user data into a dynamically allocated buffer and then performs validation checks can be leveraged. This includes kernel functions that use `strndup_user`, `memdup_user` and `memdup_user_nul`. Therefore, we use a CodeQl query to obtain all the code locations where these functions are called. Next, we manually validate, e.g., by running test programs to execute the syscall that executes the function found, that these code locations are accessible from userspace as an unprivileged user. In the example of the `add_key` syscall, we execute this syscall with a `_desc` where the first byte is a `'.'`. All other conditions with additional information to execute the measuring primitives can be found in Table 8.

C.4 Finding Memory Write Primitives

For memory write primitives, our queries obtained all code locations that execute `copy_from_user`. With all these locations, we search for our three distinct code patterns, i.e., persistent code patterns 1 and 2, and temporal code patterns. We then filter out those that violate constraints, e.g., `timed_alloc` shown in Listing 2. As a result, we obtain functions that can be used as MWP, shown in Table 9.