ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# WHEN GOOD KERNEL DEFENSES GO BAD: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks

Lukas Maar
*Graz University of Technology*

Lukas Giner
*Graz University of Technology*

Daniel Gruss
*Graz University of Technology*

Stefan Mangard
*Graz University of Technology*

## Abstract

Over the past decade, the Linux kernel has seen a significant number of memory-safety vulnerabilities. However, exploiting these vulnerabilities becomes substantially harder as defenses increase. A fundamental defense of the Linux kernel is the randomization of memory locations for security-critical objects, which greatly limits or prevents exploitation.

In this paper, we show that we can exploit side-channel leakage in defenses to leak the locations of security-critical kernel objects. These location disclosure attacks enable successful exploitations on the latest Linux kernel, facilitating reliable and stable system compromise both with re-enabled and new exploit techniques. To identify side-channel leakages of defenses, we systematically analyze 127 defenses. Based on this analysis, we show that enabling any of 3 defenses – enforcing strict memory permissions or virtualizing the kernel heap or kernel stack – allows us to obtain fine-grained TLB contention patterns via an Evict+Reload TLB side-channel attack. We combine these patterns with kernel allocator massaging to present location disclosure attacks, leaking the locations of kernel objects, i.e., heap objects, page tables, and stacks. To demonstrate the practicality of these attacks, we evaluate them on recent Intel CPUs and multiple kernel versions, with a runtime of 0.3 s to 17.8 s and almost no false positives. Since these attacks work due to side-channel leakage in defenses, we argue that the virtual stack defense makes the system less secure.

## 1 Introduction

The security of modern systems relies on privilege levels. While user programs have lower privileges, the operating system kernel has higher privileges and can typically access all system memory. Thus, the system's security depends directly on the security of the kernel. However, kernels – such as Linux – are also complex, which often unintentionally introduces software vulnerabilities, many of which remain unknown for years. To generically limit the impact of these vulnerabilities, kernels employ several defenses that limit what a bad actor can do after exploiting a software bug. A fundamental defense in the Linux kernel is the randomization of memory locations for security-critical objects. This strategy prevents exploitation outright or forces the bad actor to locate these randomized locations before achieving system compromise.

Side-channel attacks offer a promising approach to circumvent these randomization-based defenses and have been studied extensively. Numerous works [1, 2, 15, 20, 29, 36] have exploited a side channel in the Translation Lookaside Buffer (TLB), a CPU buffer that stores virtual-to-physical address translations. These TLB side channels have allowed partial bypasses of Kernel Address Space Layout Randomization (KASLR) [11], which randomizes parts of the kernel, e.g., the kernel code, module code, and Direct-Physical Map (DPM). Specifically, these works break code or physical KASLR, which refers to leaking the base address of randomized code sections or the DPM. Multiple kernel exploits have used these KASLR breaks [13, 38, 52], including those from Google Project Zero and Google's bug bounty program.

However, existing techniques do not reveal the locations of security-critical kernel objects, a requirement for many attacks to compromise the system. At the same time, as defenses become more integrated – particularly in the memory mapping subsystem – the risk increases that these protections, while intended to enhance security, may unintentionally expose the system to more precise leaks. As a result, it is unclear *which*, if any, of the defenses actually allow more precise leakage.

In this paper, we show that while kernel defenses are valuable in mitigating vulnerability exploitation, enabling any of 3 defenses allows for side-channel attacks to deduce the locations of security-critical kernel objects in a way that allows reliable and stable privilege escalation on modern Linux kernels. We refer to these as location disclosure attacks.

To identify kernel defenses that allow these location disclosure attacks, we perform a systematic side-channel analysis. We analyze all 127 defenses recommended by the Kernel Self-Protection Project (KSPP) [60] or used within Google's KernelCTF [13] bug bounty program. These defenses include protection against various exploit techniques, such as cross-

cache reuses [31, 41, 64, 67] – that exploit the memory reuse by a kernel allocator – and kernel code tampering attacks [12]. We classify these defenses into 5 categories. We then observe that from one category, 3 defenses – enforcing strict memory permissions or virtualizing the kernel heap or kernel stack – modify the memory mapping to create exploitable access patterns in the TLB. These defenses change the mapping so that objects are accessed with a fine-grained mapping of 4 kB instead of 2 MB, which is how most kernel memory is accessed. This results in using 4 kB TLB entries, with contention patterns observable via a side channel.

We then present location disclosure attacks that leak the locations of kernel objects. Combining strategic kernel allocator massaging with TLB contention patterns allows the leak of page-aligned object locations and consequently deduces all sub-page granular object locations, all attacker-controlled. To perform these attacks, a bad actor must first load the object's address into the TLB. However, the kernel does not provide a way to load only one target kernel address into the TLB, as even the simplest syscall accesses and loads multiple addresses. Instead, we use a so-called access primitive multiple times with different arguments, which loads numerous addresses – including the target address – into the TLB. This creates multiple TLB contention patterns, which we leak via an Evict+Reload TLB side-channel attack using 2 known and 1 novel distinguishing primitive, i.e., distinguish 2 MB from 4 kB mappings. Due to strategic prior massaging, we use these patterns to deduce the locations of most security-critical kernel objects, i.e., `pipe_buffer`, `msg_msg`, `cred`, `file`, `seq_file`, page table (all levels), and kernel stack.

To demonstrate the practicality of these disclosure attacks, we leak object locations on recent Intel CPUs ranging from the 8th to the 14th generation and generic kernels ranging from v5.15 to v6.8. We evaluate on an idle and stressed system. For an idle system, these attacks require between 0.3 s to 17.8 s with almost no false positives. For a stressed system, the false positives increase to about 7 % despite being close to full CPU load with significant TLB pressure.

Using our disclosure attacks, we can perform privilege escalation on modern kernels (e.g., v6.8) without crashes and nearly 100 % reliability. We show 3 results: First, our disclosure attacks re-enable exploit techniques that have been largely prevented. Second, disclosure attacks enable a new exploit technique that was previously not possible due to the limited capabilities of most vulnerabilities. Third, we even argue that the virtual kernel stack defense reduces security.

Finally, we discuss the security implications of our disclosure attacks, e.g., that the exploitation becomes substantially more reliable and stable with already known security-critical kernel objects. We also discuss challenges inherent in fully mitigating location disclosure attacks, e.g., preventing the kernel from using 4 kB mappings for kernel objects.

**Contributions.** The main contributions of this work are:

**(1) Side-Channel Analysis of Kernel Defenses:** We systematically analyze all 127 defenses recommended by the KSPP or used within Google's bug bounty program for their side-channel leakage, showing that 3 leave fine-grained, exploitable TLB contention patterns.

**(2) Location Disclosure Attacks:** We present disclosure attacks combing allocator massaging with Evict+Reload-style TLB attacks to leak these fine-grained patterns and deduce the locations of security-critical kernel objects, i.e., heap objects, page tables, and stacks. We evaluate our attacks on recent Intel CPUs and kernel versions.

**(3) Reliable and Stable Kernel Exploitation:** We show that our disclosure attacks allow privilege escalation without crashes and nearly 100 % reliability on modern systems, e.g., Linux kernels v6.8. They re-enable a new exploit technique and previously prevented exploit techniques.

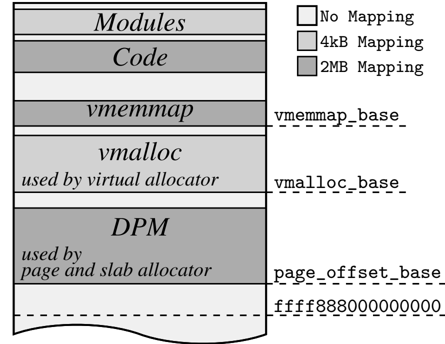


Figure 1: Virtual memory layout of the x86_64 Linux kernel.

**Outline.** Section 2 provides the background. Section 3 presents the workflow. Section 4 discusses our side-channel analysis of the defenses. Section 5 combines massaging with exploitation of these side-channel leaks for disclosure attacks. Section 6 evaluates these attacks. Section 7 shows how the leakage can be used in kernel exploits. Section 8 provides a discussion. Section 9 concludes our work.

## 2 Background

This section provides the necessary background for this work.

**Kernel Allocators.** The Linux kernel provides 3 allocators (i.e., *page allocator*, *slab allocator*, and *virtual memory allocator*), where Figure 1 shows the memory areas used.

First, the *page allocator* [28] divides the Direct-Physical Map (DPM) [44] – a linear virtual mapping of (typically) the entire physical memory – into page-order memory chunks. This allocator combines memory allocation with free memory coalescing. Simply put, it provides global and per-CPU page-order free lists. Both free lists allow allocation of physically contiguous page-order memory chunks, where the kernel first allocates/deallocates chunks from the per-CPU lists. If the per-CPU lists are exhausted on allocation or exceed a maximum capacity on deallocation, the kernel resorts to the global lists.

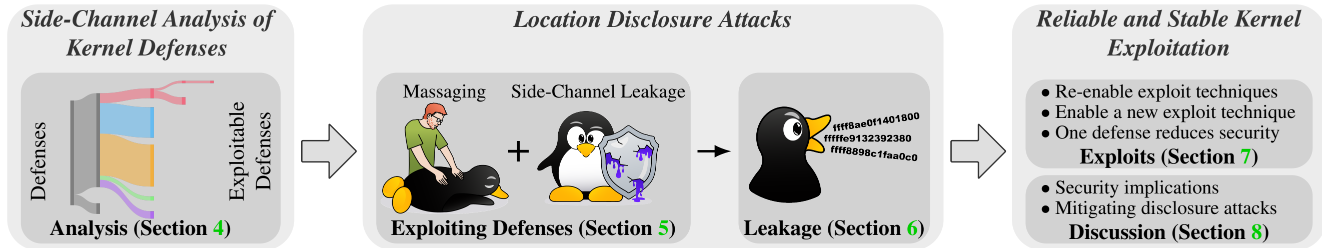


Figure 2: The high-level overview of our work.

Second, the *slab allocator* allocates page-order chunks – used as slabs – from the page allocator, where the slabs cache free and available memory slots [7, 27]. This allocator provides two types of caches, both of which use slab pages for allocating and deallocating objects: dedicated and generic caches. Dedicated caches are used for frequent object allocation with `kmem_cache_alloc`. Generic caches are used for allocating less frequently used objects and have multiple allocator caches matched to different sizes. When allocating objects from a generic cache with `kmalloc`, the kernel first matches the object size to one of these caches and then returns a free and available memory slot from that cache.

Third, the *virtual memory allocator* uses `vmalloc` to provide a virtually contiguous memory with a specific mapping.

**SLUBStick.** Maar et al. [41] introduced a timing side channel of the slab allocator to detect new slab page allocations. It works by timing object allocations from user space. Fast allocations indicate that the currently active slab page has returned the memory slot. Slow allocations indicate a new slab page allocation by the page allocator that returns a slot for the object. By grouping object allocations according to their timing, SLUBStick determines all objects on a slab page.

**Kernel Memory Mapping.** Figure 1 shows the virtual memory layout of the Linux kernel with 5 relevant areas for this work, all of which are randomized due to KASLR [11].

*Code* and *Modules* contain the instructions of the kernel binary and inserted modules. The *DPM* is a virtual memory area of (typically) the entire physical memory, mostly mapped with 2 MB pages. Crucially, the kernel heap allocated by the slab allocator accesses the DPM directly. On x86_64 systems, the kernel places the DPM at a random 1 GB-aligned address – the `page_offset_base` – between `ffff888000000000` and `ffffc88000000000`. The *vmalloc* area (mapped with 4 kB pages) contains memory slots allocated with the virtual allocator. This area is randomly located to a 1 GB-aligned address – the `vmalloc_base` – between the DPM's end and `vmemmap`'s start. The *vmemmap* area is a virtual memory mapping that stores metadata for each physical page. More specifically, this virtual mapping is an array of 64 B `page` objects that store this metadata and is indexed by the physical frame number. It is located at a 1 GB-aligned address – the `vmemmap_base` – between `vmalloc`'s end and `ffffffe0000000000`.

**Kernel Exploitation.** Most kernel exploits that exploit memory-corruption vulnerabilities work similarly: A bad actor initially triggers the vulnerability, such as Out-Of-Bounds (OOB) or Use-After-Free (UAF), to falsely put an object – often referred to as the victim object – in a free state. They then reuse the victim's memory slot for a different object. There are two variants of reuse attacks: First, they perform an in-cache reuse and reuse the victim object as another object from the same allocator cache, e.g., `kmalloc-*`. However, this limits reuse to objects with the same (or similar) size and allocation properties, as heap separation prevents direct reuse of the victim as a security-critical object. Second, they perform a cross-cache reuse by freeing all memory slots on the slab page that contains the victim object, causing page recycling. They then reclaim the page (typically) used for a security-critical purpose, e.g., DirtyCred [33] as a credential reference, CVE-2022-27666 [71] as a `msg_msg`, DirtyPage [34]/CVE-2020-29660 [23] as a `pipe_buffer`, or Dirty PageTable [65]/SLUBStick [41] as a page table.

**Translation-Lookaside Buffer.** The TLB is a per-core CPU cache that stores virtual-to-physical address translations. When an address translation is found in a TLB entry, it eliminates the need to perform a page-table walk for memory access, resulting in a speedup. There are typically two levels of TLBs, with the first level split between data (DTLB) and instructions (ITLB), while the second level (STLB) is shared between data and instructions. Each TLB is a set-associative cache, i.e., split into sets and ways. For example, an Alder Lake CPU might feature two STLBs of 128 sets with 8-way associativity each, one for 4 kB and 2 MB/4 MB pages, and one for 4 kB and 1 GB pages. When a virtual address is not found in any TLB, a page-table walk is performed, and an existing entry in the TLB is evicted to store the new translation.

## 3 High-Level Overview

Our work is based on 3 components, as shown in Figure 2.

*First*, we show that several kernel defenses leave exploitable, fine-grained TLB contention patterns (see Section 4). Initially, we analyze all 127 defenses recommended by the KSPP [60] or used by KernelCTF [13], which protects against techniques, e.g., code manipulation [12] and cross-

cache reuse [31, 41, 67]. We categorize them into 5 categories based on how they improve security. We then analyze whether they change the memory mapping to create fine-grained TLB patterns, resulting in 3 defenses of a particular category.

*Second*, we show that combining kernel allocator massaging with these patterns enables the leaking of the location of target objects (see Section 5). To achieve this, we perform an Evict+Reload TLB side-channel attack and obtain the leakage of each exploitable defense. Using these leaks, we show that due to the prior massaging, we can deduce the location of most security-critical objects, i.e., heap objects, page tables, and kernel stacks, which are all popular exploitation targets such as `msg_msg` [4, 10, 24, 37, 48, 71], `pipe_buffer` [34, 49, 62], `cred` [17, 33], `seq_file` [19, 25, 58], `file` [33, 56, 63, 68], page table [41, 47, 64, 65], and kernel stack [52, 66, 69]. We demonstrate the practicality of these attacks on recent Intel CPUs and multiple kernel versions (see Section 6).

*Third*, we show that by using our disclosure attacks, we can build reliable and stable exploits (see Section 7): First, they re-enable exploit techniques that have been largely prevented. Second, they enable a new technique that was not previously possible due to the limited capabilities of most vulnerabilities in their initial state. Third, for one defense, we even argue that the defense provides less security due to disclosure attacks. We then discuss (see Section 8) the security implications of the disclosure attacks and the challenges of full mitigation.

**Threat Model.** We assume an unprivileged user with code execution, a typical scenario for the last stage of a full-chain exploit [57]. We also consider the presence of an exploit primitive, such as kernel UAF or OOB write, due to a kernel heap bug. We assume that all upstream defenses available in v6.9 (i.e., the latest version when we started our work) are enabled and exclude defenses that require paid subscriptions (e.g., AUTOSLAB [31]), in line with prior work [17, 33, 41, 69]. In line with our evaluation CPUs, while KPTI is included in the kernel binary, it is disabled by the CPU.

## 4 Side-Channel Analysis of Kernel Defenses

In this section, we detail our systematic analysis of all 127 kernel defenses recommended by the KSPP [60] or used within Google's KernelCTF [13] bug bounty program. We aim to determine which kernel defenses introduce exploitable fine-grained TLB contention patterns. We first organize all kernel defenses into 5 categories. We then show that 3 defenses of one category introduce exploitable fine-grained leakage.

### 4.1 Requirements

There are two requirements for defenses to leave exploitable contention patterns in the TLB. We first discuss these and then describe how kernel defenses satisfy these requirements.

**R1.** The identified kernel defenses must produce different TLB contention patterns when applied than when not applied. Thus, the first requirement is that applying a defense must change the memory mapping. We refer to a mapping change in *where* or *how* kernel objects are mapped, potentially creating exploitable TLB contention patterns on object access.

**R2.** Since most of the accessed kernel memory is mapped as 2 MB pages [6], TLB contention patterns mainly occupy 2 MB entries. Even if we leak which 2 MB entries the target object occupies, the offset within that 2 MB page remains unknown. Thus, as a second requirement, the defense must alter the mapping of objects from 2 MB to 4 kB pages, creating contention patterns in 4 kB TLB entries, so that we can leak the 4 kB location of the object first and narrow it down to its sub-page granular locations subsequently. This can be done either by statically switching from a 2 MB to a 4 kB mapped region or by dynamically changing the mapping.

Since defenses satisfying **R2** are a subset of defenses satisfying **R1**, **R1** helps to filter out defenses that do not satisfy **R2**, allowing for efficient and accurate kernel defense analysis.

**Unexploitability of 2 MB Mappings.** As we will show in Section 5, our disclosure attacks require full control over the smallest memory granularity leaked by TLBs. With 2 MB mappings, adversaries must ensure all controlled objects occupy and map exclusively to an entire 2 MB memory chunk, which is infeasible because: First, the largest slab size is 32 kB and, second, slab adjacency to fill exclusively only to a 2 MB memory chunk is almost impossible, e.g., due to `CONFIG_SHUFFLE_PAGE_ALLOCATOR`.

### 4.2 Classification of Kernel Defenses

From the 127 defenses, we filter out those that do not apply to any of our evaluated kernels (i.e., v5.15, v6.5, v6.6, and v6.8), e.g., `CONFIG_RANDOM_TRUST_BOOTLOADER` is not available in these versions. This leaves us with 114 defenses (see Figure 3), which we divide into the *Memory Mapping Change*, *Reduce Attack Surface*, *Add Checks*, *Poisoning/Cleanup*, and *Others* categories. We then analyze whether these categories have the potential to satisfy our requirements and find that *Memory Mapping Change* satisfies **R1**. Finally, we show that multiple defenses within this category satisfy **R2**, leaving exploitable contention patterns in 4 kB TLB entries.

**Non-Exploitable Categories.** To filter the non-exploitable categories from the *Memory Mapping Change* category that satisfy **R1**, we automatically identify all files containing an `#ifdef` of a kernel defense. We then automatically determine whether the file is directly responsible for the memory mapping, e.g., `vmlinux.lds.S`, or is located in a directory that handles mappings, e.g., `arch/*/mm/`. In these cases, we manually analyze the files and determine whether they contribute to memory mapping changes. Several automatically identified defenses turned out as false negatives on manual analysis, e.g., `CONFIG_STRICT_DEVMEM` includes checks in `arch/x86/mm/pat/memtype.c` but does not contribute to mapping changes. After the analysis, we are left

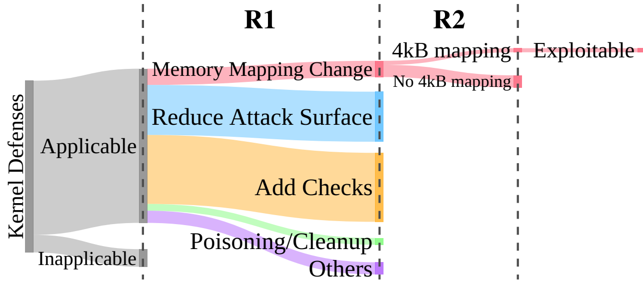


Figure 3: Our classification of kernel defenses shows the *Memory Mapping Change* defenses inducing exploitable TLB contention patterns that leak the precise location of objects.

with 12 defenses contributing to mapping changes.

For completeness, we briefly describe the other non-exploitable categories *Reduce Attack Surface*, *Add Checks*, *Poisoning/Cleanup*, and *Others*, where we use manual classification based on the following definitions: *Reduce Attack Surface* limits the amount of kernel code accessible to an application [22, 54], e.g., resetting `CONFIG_X86_VSYSCALL_EMULATION` removes virtual syscalls handling. *Add Checks* defenses insert security checks, ensuring the integrity of parts of the kernel, e.g., `CONFIG_SLAB_FREELIST_HARDENED` protects slab metadata. We classify defenses as *Poisoning/Cleanup* if they poison or cleanup registers (e.g., `CONFIG_ZERO_CALL_USED_REGS`) or memory (e.g., `CONFIG_PAGE_POISONING` or `CONFIG_INIT_ON_FREE_DEFAULT_ON`). Finally, *Others* defenses do not fit in the other categories, e.g., `CONFIG_SCHED_CORE` permits core scheduling. To summarize, none of these categories contribute to memory mapping changes, and, thereby, they do not satisfy **R1**.

**Defenses that Change the Memory Mapping.** *Memory Mapping Change* defenses can be divided into those that change the object's mapping to 4 kB pages (satisfying **R2**) and those that do not. For those that do not, several of them change the object's location by separating them according to their security context. Examples include `CONFIG_KMALLOC_CG` with coarse-grained separation, `CONFIG_KMALLOC_SPLIT_VARSIZE` to separate elastic objects [4], and `CONFIG_RANDOM_KMALLOC_CACHES` to randomly assign dedicated caches to allocation sites. Other defenses that also do not change the mapping are `CONFIG_SHUFFLE_PAGE_ALLOCATOR` and `CONFIG_RANDSTRUCT_FULL`, which randomize page allocations or the members within a struct. Another example is `CONFIG_STRICT_KERNEL_RWX`, which only sets code permissions on a 2 MB granularity. Since none of these defenses change the object's mapping to 4 kB pages, they do not satisfy **R2**.

This leaves 3 defenses that change the object's mapping to 4 kB pages. `CONFIG_STRICT_MODULE_RWX` satisfies **R2** as follows: It must set the permission of the virtual memory and the DPM of the module code to non-writable [12, 60]. To achieve permission settings in the DPM, this defense splits the 2 MB pages of the DPM into 4 kB, allowing legitimate use of other pages from the former 2 MB page, e.g., for the heap. `CONFIG_SLAB_VIRTUAL` and `CONFIG_VMAP_STACK` satisfy **R2** as they use virtual memory mapped with 4 kB pages.

**Discussion.** A false positive is when an identified defense leaves no exploitable TLB contention patterns. However, all of our identified defenses leave exploitable patterns that leak locations of target objects. A false negative is when a defense allows location leakage, but we missed it. To minimize false negatives, we strictly defined our requirements **R1** and **R2** and systematically analyzed the defenses. We also performed a bottom-up approach and tried to find defenses that include mapping changing functions such as `set_memory_ro`.

We reveal 3 exploitable and 124 non-exploitable defenses. This provides encouraging results for defenders, as only 3 defenses require hardening against our attacks. Researchers can focus on these instead of all 127 defenses.

**Takeaway 1**

Our analysis shows that 3 defenses change the memory mapping of kernel objects to 4 kB pages, which creates exploitable contention patterns in 4 kB TLB entries.

## 5 Exploitation of Kernel Defenses

To leak the location of a target object, we need to load its page-aligned address into the TLB. However, even the simplest syscall accesses multiple kernel addresses and loads them into the TLB. Instead, we call a so-called access primitive multiple times with different arguments. These access primitives load the addresses of a couple hundred to a thousand accessed objects – including the target address – and, therefore, create TLB contention patterns. By strategically massaging the kernel allocators beforehand, we use these patterns to infer the page-aligned location of the target object. We then deduce all sub-page granular object locations within the leaked page, all controlled by the attacker. Below, we set the challenges of exploiting **D1-3** and massaging the allocators for location leakage, while Sections 5.1 to 5.3 solve them.

**C1.** The first challenge is to ensure that our target object is located on 4 kB mappings. Therefore, on object access, its page-aligned address is now loaded into a 4 kB TLB entry, resulting in 4 kB TLB entries are occupied.

**C2.** The second challenge is to minimize TLB noise from uncontrolled accesses within the access primitives. Reducing the noise is critical, as otherwise, it could lead to the target TLB entry being misoccupied, resulting in misclassification or the inability to locate the target 4 kB TLB entry.

**C3.** The third challenge is to deduce the location of the target kernel object from the contention patterns in 4 kB TLB entries. To achieve this, we first need to determine the 4 kB TLB entry of our target kernel object (**C3.1**). For reference, when leaking `msg_msg`, we need to non-trivially reduce the 976 TLB entries of its access primitive to 1 target entry. Finding this TLB entry gives us the page-aligned kernel address

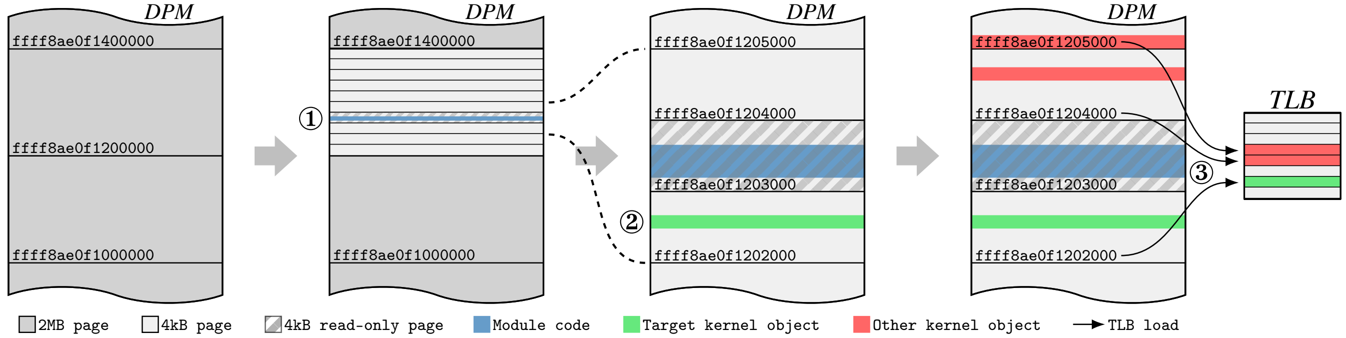


Figure 4: Exploiting `CONFIG_STRICT_MODULE_RWX` to create TLB contention patterns via an access primitive. The insertion of a module ① causes a split of a 2 MB page into 4 kB pages to set the module's page to read-only. We then allocate the target kernel object we want to leak ②, which will be on a 4 kB page. Finally, we execute the access primitive ③, which accesses the target with other objects, loading their page-aligned addresses into the TLB and creating the contention pattern on the 4 kB entries.

of the object. Next, we must derive the object locations within this leaked page address (**C3.2**).

## 5.1 D1: Strict Kernel Memory Permissions

By massaging the kernel allocators combined with exploiting the `CONFIG_STRICT_MODULE_RWX` defense, we can solve **C1-3** and leak the location of heap objects and page tables.

**Protection Scheme.** Kernel memory containing writable code is an easy target for control-flow redirection [12, 60]. As a mitigation, the kernel supports `CONFIG_STRICT_*_RWX` to guarantee no kernel code is writable [60]. Specifically, the kernel can protect dynamically loaded module code (i.e., `CONFIG_STRICT_MODULE_RWX`) this way: When loading module code into a virtual memory range, the kernel restricts this range to be executable and read-only and the corresponding range in the DPM to be read-only. However, as the DPM is mapped mainly as 2 MB pages, the kernel has to split – prior to changing permissions – this 2 MB page into 4 kB pages. Hence, physically adjacent pages can remain writable and be used by the kernel, e.g., for the kernel heap.

**TLB Contention Pattern.** Figure 4 shows exploiting TLB contention patterns created by this defense combined with allocator massaging. In the first stage, we load a module, forcing a 2 MB to 4 kB page split ①. The kernel then sets the page within the DPM that contains the module's code (i.e., blue memory range) to read-only. Unprivileged kernel module loading can be done by opening a socket that does not have its kernel driver loaded.

In the second stage, we allocate a target object (i.e., green memory range) to claim a memory location mapped as a 4 kB page ②, using its so-called allocation primitive. To claim a 4 kB mapped slot, we drain the lower page-order free lists of the page allocator before loading the module by allocating many dummy objects. Due to the page allocator's intrinsic behavior, when the lower page-order free lists are drained, it partitions higher page-order chunks and uses them for the

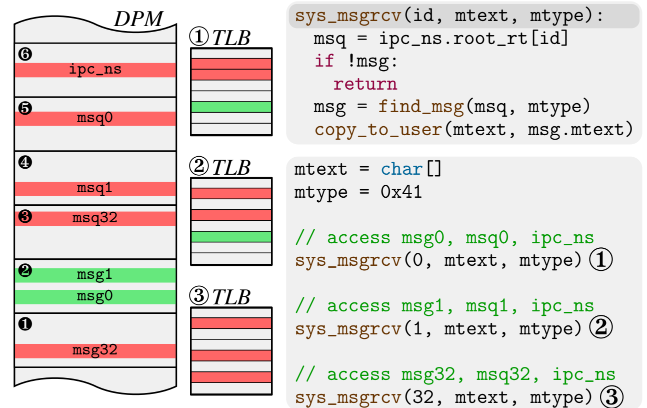


Figure 5: TLB contention patterns by calling `msgrcv` ①-③.

lower page-order allocations [17, 41, 64]. After these free lists are drained, the module allocation uses a chunk with an adjacent free chunk, located on the split 2 MB page. Now, subsequent object allocations likely claim one of these adjacent chunks, solving **C1**.

In the third stage, we use an access primitive that accesses multiple kernel addresses and induces TLB contention. The TLB contention pattern consists of the page-aligned address of our target kernel object (i.e., green memory range) and addresses of other accessed objects (i.e., red memory areas).

**Leaking the Object's Location.** While inferring the location of our target object from TLB contention is a generic approach, we explain it using the `msg_msg` as an example. For the `msg_msg` object, the `msgsnd` and `msgrcv` functions act as an allocation and access primitive. Figure 5 depicts the access primitive that first accesses the Inter-Process Communication (IPC) root tree `ipc_ns.root_rt` to obtain `msq` that matches `id`. It then accesses `msq` to obtain `msg_msg` matching the type `mtype` and copies the data stored within this object to user space. Executing this primitive with different `id`s creates

different access patterns in the TLB. This simplified example shows a contention pattern with three 4 kB TLB entries, whereas in reality, the access primitive creates patterns with 976 entries across all TLB levels.

In Figure 5, we aim to leak the address of msg0 residing on page ❷. To minimize the noise on the TLB, we exploit the caching property of the slab allocator, solving **C2**. Specifically, we make sure that the slab page of our target object (i.e., msg0) contains only msg_msgs: First, by allocating multiple msg_msgs, and second, by validating via the SLUBStick timing side channel [41] that only msg_msgs are stored on this slab page (see Section 2). We refer to the objects on page ❷ as msg0 to msg31. We repeat this process so that we have a second slab page ❶ occupied with msg_msgs.

Next, we separate other objects, such as msg_queue, on different slabs via dummy allocations between two different msq_queue (i.e., msq0 and msq1) allocations so that all other memory slots within the slab page are used for dummy objects. For example, page ❺ contains one msg_queue object, and the other slots of the slab page are dummy objects. Figure 5 shows the memory layout of the DPM when using both approaches.

With the object layout we crafted, executing msgrcv with different arguments results in different TLB entries being occupied. Executing this primitive with an id of 0 ① results in entries 2, 5, and 6 being occupied, while running it with an id of 1 ② results in 2, 4, and 6 being occupied. The execution with an id of 32 ③ results in occupancy of entries 1, 3, and 6. Hence, we can distinguish common and differential TLB access patterns: The patterns of ① and ② are common, as their accessed objects (i.e., msg0/1) are on the same slab page. Pattern ① is differential to ③ as it does not access the same target slab page. We determine the common entries of ① and ②, resulting in 2 and 6. We remove the common entries with ③, resulting in entry 2, the correct page ❷. Thus, we obtained the page-aligned address of msg0, solving **C3.1**.

To obtain the location of all objects within the leaked and controlled slab page (solving **C3.2**), we consider the alignment enforced by the page and slab allocators [42]. The page allocator enforces that the base address of its slab is always aligned to its page order. Using the kmalloc-cg-128 cache as an example, since this cache contains 0-order slab pages[1], their base addresses are page-aligned[2]. Objects allocated from this cache are then located at $128 \cdot n + slab_base$ where $n$ is between 0 and 31, and *slab_base* is the page-aligned address leaked by **C3.1**. Since we have occupied the entire slab page with msg_msgs, we obtain the location of all 32 objects within the page. While we now have all locations within the slab page, we do not know which object is at which specific leaked location but we do not need to as all objects are adversary controlled. We show in Section 7, this is sufficient for escalating privileges without crashes and nearly 100 % reliability.

[1] n-order slab pages refer to a slab of size $2^n \cdot$ PAGE_SIZE.

[2] /sys/kernel/slab/kmalloc-* contains these details, which remain consistent across the same kernel version.

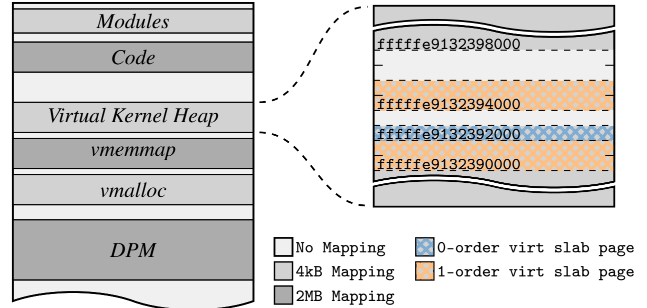


Figure 6: Memory layout when using CONFIG_SLAB_VIRTUAL, which now has a virtual kernel heap. This virtual heap contains the virtual slabs where the heap objects are located.

Besides msg_msg, we perform a similar approach with allocator massaging and common/differential access patterns for other objects. We leaked the locations of pipe_buffer, cred, file, seq_file, and page-tables, i.e., Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table (PT). Table 3 shows their allocation and access primitives. For heap objects, we achieve the common/differential access patterns by allocator massaging, while for the page tables, we place the page-table levels in our favor (see Appendix B).

**Takeaway 2**

While **D1** prevents tampering with module code, it creates exploitable TLB contention patterns and, thus, allows location leakage of heap objects and page tables.

## 5.2 D2: Virtual Memory for Kernel Heap

The Linux kernel has enhanced heap defenses that separate objects into different sets of allocator caches based on their security context. This separation prevents in-cache reuse, where a victim object's memory slot is directly reused for security-critical objects. While this separation makes vulnerability exploitation more difficult, cross-cache reuse [67] has been proposed to circumvent this separation. It exploits the memory reuse of the page allocator and has received considerable attention from academia [17, 33, 40, 41, 67] and industry [31, 45, 61]. To counter this development, security experts presented the defense CONFIG_SLAB_VIRTUAL [45] and deployed it in Google's hardened system for KernelCTF [13]. While this defense provides significant value in mitigating cross-cache reuse, we show that it creates TLB contention patterns that enable leaking the location of target heap objects.

**Protection Scheme.** CONFIG_SLAB_VIRTUAL [45] deterministically prevents the reclaiming of heap memory that has been returned to the page allocator. They achieve this by using a virtual mapped area as heap instead of the DPM, which is mapped with 4 kB pages. We refer to this area as the virtual kernel heap, as illustrated in Figure 6. Since the slab pages are now in the virtual heap, we refer to them as virtual slab

pages. A unique feature of this defense is that it continuously increases the heap and never returns memory used by virtual slab pages back to the page allocator. This prevents the corresponding memory chunk from being reused in different slab caches, which deterministically prevents cross-cache attacks.

**Leaking the Object's Location.** Since this defense uses 4 kB mapped virtual memory areas, applying this defense causes accesses to heap objects (except for DMA-related memory [45]) to occupy 4 kB TLB entries, satisfying **C1** by design. However, accessing nearly the entire heap via 4 kB mappings also introduces TLB noise since the previously used 2 MB mappings are now all 4 kB. To compensate for the higher TLB noise floor, we use the same approach as in Section 5.1 to create common and differential patterns in the TLB, but with more varying arguments. From these patterns, we deduce the target kernel page, solving **C2** and **C3**.

**Takeaway 3**

While **D2** provides significant value in mitigating cross-cache attacks, it enables leaking heap object locations.

### 5.3 D3: Virtual Memory for Kernel Stack

The `CONFIG_VMAP_STACK` kernel defense [35] uses a virtual stack with guard pages instead of physically-mapped kernel stacks, preventing kernel stack overflows. However, we show that it also allows the location of kernel stacks to be leaked.

**Protection Scheme.** With this defense disabled, Linux uses the DPM directly for kernel stacks. Since the DPM is (mostly) continuous, a stack overflow would corrupt pages adjacent to the kernel stack, which may cause a difficult-to-diagnose corruption. To detect these overflows, `CONFIG_VMAP_STACK` allocates the stack with `vmalloc` and includes virtual guard pages. Now, the stack is located within the virtual memory area (see Figure 1), mapped with 4 kB pages.

**Leaking the Object's Location.** With this defense enabled, the thread's stack is accessed with memory mapped via 4 kB pages, solving **C1** by design. To exploit this defense, we need a syscall that accesses the kernel stack with minimal other objects. We use an invalid syscall (i.e., syscall number -1) as an access primitive that only accesses the kernel stack and a few other kernel objects, such as `current`. To reduce the TLB noise and solve **C2**, we consider the stack's alignment, including the used guard pages. To solve **C3.1**, we repeatedly call the invalid syscall and determine the 4 kB TLB entry that occupies the page-aligned kernel stack. Lastly, since even with the kernel defense `CONFIG_RANDOMIZE_KSTACK_OFFSET_DEFAULT` enabled, the invalid syscall only accesses the top page from the kernel stack, the leaked TLB entry is the current kernel stack, satisfying **C3.2**.

**Takeaway 4**

While **D3** comfortably detects kernel stack overflows, it allows the location of the kernel stack to be leaked.

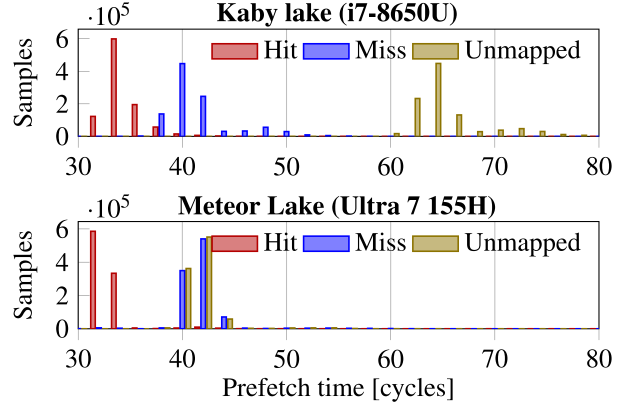


Figure 7: Prefetch timings of 4 kB pages.

## 6 Location Disclosure Attacks through Defense Side-Channel Leakages

In this section, we first describe the side-channel primitives to leak the TLB contention patterns via an Evict+Reload TLB side-channel attack (see Section 6.1). We then evaluate the location disclosure attack using Evict+Reload and leak most security-critical kernel objects (see Section 6.2).

**Evaluation Setup.** We evaluate a wide range of Intel CPUs (i.e., Kaby, Coffee, Alder, Raptor, and Meteor Lake), which are all vulnerable to our disclosure attack. The used kernels for the **D1/3** exploit are the generic v5.15, v6.5, and v6.8 ones, while for the **D2** exploit, we applied the `CONFIG_SLAB_VIRTUAL` patch to its intended v6.6 kernel (see Table 2). We extensively evaluate our attack on the Intel i7-1360 and kernel v6.8 for **D1/3** and v6.6 for **D2** to leak the location of heap objects, page tables, and thread stacks (see Table 1).

### 6.1 Distinguish Different Memory Mappings

The `prefetch` instruction [15] allows us to measure whether or not a page is currently in the TLB by measuring its execution time. As it does not architecturally access the data, this does not cause an access violation, even if performed on kernel pages from user space. We implement a fast, set-targeted TLB eviction for both TLB levels to repeatedly sample pages. We base this on the reverse-engineering of TLB addressing functions done by prior work [14, 59, 70]. While the overall TLB structure on Intel CPUs has changed significantly since the Coffee Lake architecture evaluated in prior work, we find that for 4 kB pages, the appropriate indexing functions still perform well for set eviction on Ice Lake and later generations. Combining `prefetch` and TLB set eviction, we get an equivalent attack to Evict+Reload [16] for pages.

For our attack, we need three distinguishing primitives: First, we need to distinguish TLB hits from misses on mapped pages. This allows us to locate specific kernel objects for

which we have an access primitive, e.g., `msgrcv` for `msg_msg`. Second, distinguishing mapped from unmapped pages allows us to find coarse-grained memory areas, e.g., DPM or `vmemmap` (see Figure 1). Third, our novel primitive allows us to recognize whether an address is mapped to a 2 MB or a 4 kB page, which enables us to find pages that a kernel defense has split.

Figure 7 shows the prefetch timings for Kaby and Meteor Lake. We find that `prefetchnta` and `prefetcht2` combined in one measurement leads to good results on all tested CPUs. We see that TLB hits are distinguishable from misses. Thus, for our *hit/miss primitive*, we evict the TLB set corresponding to the page we are testing and then measure the timing of prefetch on that page. As shown, the distributions are separable, so few repetitions are sufficient for stable results.

On Kaby Lake, we see that mapped and unmapped pages are also clearly distinguishable. However, changes in the newer Meteor Lake microarchitecture cause TLB misses on mapped pages to no longer show a different timing from unmapped pages with our measurement setup. We, therefore, do not rely on this difference but simply repeatedly measure the prefetch timing of a page without evicting it from the TLB to get the *mapped/unmapped primitive*. If the tested page is mapped, accesses after the first one will produce a hit timing because Intel TLBs do not cache unmapped page translations.

For the *2 MB/4 kB primitive*, we combine the two prior approaches. We repeatedly evict the target TLB entry, then prefetch another address in the same 2 MB-aligned memory at least 4 kB away, and lastly, measure the prefetch timing of the target address. If our target is mapped on a 2 MB page, accessing another address on the same page will load it into the TLB, causing a hit on the measured target access.

## 6.2 Evaluation of Disclosure Attacks

In this section, we evaluate our location disclosure attacks by exploiting **D1-3** via the prefetch side channel and allocator massaging. We first leak the coarse-grain location where the object resides, and then leak the fine-grain location of that object, e.g., DPM with subsequent kernel heap object leak.

**Leaking Coarse-Grained Kernel Sections.** We leak the base of the DPM, `vmalloc`, `vmemmap`, and virtual heap for **D2** as follows: We iterate through kernel memory in 1 GB steps since the mappings are 1 GB-aligned. At each step, we use the *mapped/unmapped primitive* to determine whether the first page is mapped. If it is, we have found the region's base. For the DPM base (i.e., `page_offset_base` of Figure 1), we start at `ffff888000000000`; for `vmalloc` at the end of the identified DPM region; and for the virtual heap, we start at `fffffffe8000000000`, its lowest possible address. For `vmemmap`, we search backward from `fffffffe0000000000`. Leaking these locations requires less than 1 s.

**Leaking Fine-Grained Locations.** For defenses that access all kernel objects with 4 kB page mappings, we iterate in 4 kB steps over the entire possible memory area, i.e., virtual heap for **D2** and `vmalloc` region for **D3**. At each step, we call the access primitive for the corresponding target object and use the *hit/miss primitive* to leak the contention of the tested 4 kB TLB entry. We repeat this with different arguments for the access primitive and reconstruct the common and differential patterns to deduce the target address. For **D1**, we only perform this iteration for areas within the DPM mapped with 4 kB pages using the *2 MB/4 kB primitive*. Since most of the DPM is mapped with 2 MB pages, we can skip most addresses. On our test system more than 98 % of the DPM is typically mapped with 2 MB pages.

Table 1: Evaluation results of exploiting defenses **D1-3**, showing their Success Rate (SR), the Time (T) required, the Corrected Rate (CR), where we consider that we can repeat detect false negatives, † indicating that no stable exploit was possible, and ✻ indicating that we can reallocate it in-cache from a leaked `msg_msg` with a CR of 100 % instead.

| **Objects** | **D1** | | | **D2** | | | **D3** | | |
|---|---|---|---|---|---|---|---|---|---|
| | SR % | T s | CR % | SR % | T s | CR % | SR % | T s | CR % |
| `msg_msg` | 78 | 12.3 | **100** | 66 | 0.6 | **100** | - | - | - |
| `cred` | † | † | † | 77 | 6.6 | **98** | - | - | - |
| `file` | 80 | 8.1 | **100** | 82 | 0.4 | **100** | - | - | - |
| `seq_file` | 77 | 6.6 | **100** | 93 | 0.4 | **98** | - | - | - |
| `pipe_buffer` | 54 | 15.6 | **96 ✻** | 51 | 1.0 | **100** | - | - | - |
| PT | 83 | 17.8 | **100** | - | - | - | - | - | - |
| PMD | 93 | 14.5 | **100** | - | - | - | - | - | - |
| PUD | 85 | 14.0 | **100** | - | - | - | - | - | - |
| Kernel Stack | - | - | - | - | - | - | 98 | 0.3 | **100** |

For the evaluation, we perform the disclosure attacks 200 times with 5 reboots in between[3]. Table 1 shows the evaluation results, where we leak heap objects by exploiting **D1/2**, page tables by exploiting **D1**, and kernel stacks by exploiting **D3**. The Success Rate (SR) represents the true positives divided by the total number of runs within the averaged required Time (T). The Corrected Rate (CR) considers that we can repeat detected false negatives[4]. Examples of detected false negatives are no address found or the contention pattern does not satisfy alignment constraints. The † denotes that exploiting **D1** to leak `cred` was not stable because we perform `cred` spraying with `fork`, which allocates numerous other objects, resulting in the credentials rarely being allocated to 4 kB mapped memory. Table 1 shows that most attacks have no false positives – referred to as misidentified addresses – and the attacks that do have only about 2 %. The main cause of false positives for the **D1** exploit is that its allocation creates numerous other objects, resulting in, e.g., `pipe_buffer` may not be located on 4 kB mapped memory. However, instead of leaking the `pipe_buffer` ✻, we can leak `msg_msg` and reallocate its slot in-cache with a CR of 100 %, working

[3] For the **D1** exploit, we only insert modules in the first run after boot.

[4] $\frac{\frac{TP}{TP+FP}\cdot FN+TP}{total}$, with true/false positives $TP/FP$, and false negatives $FN$.

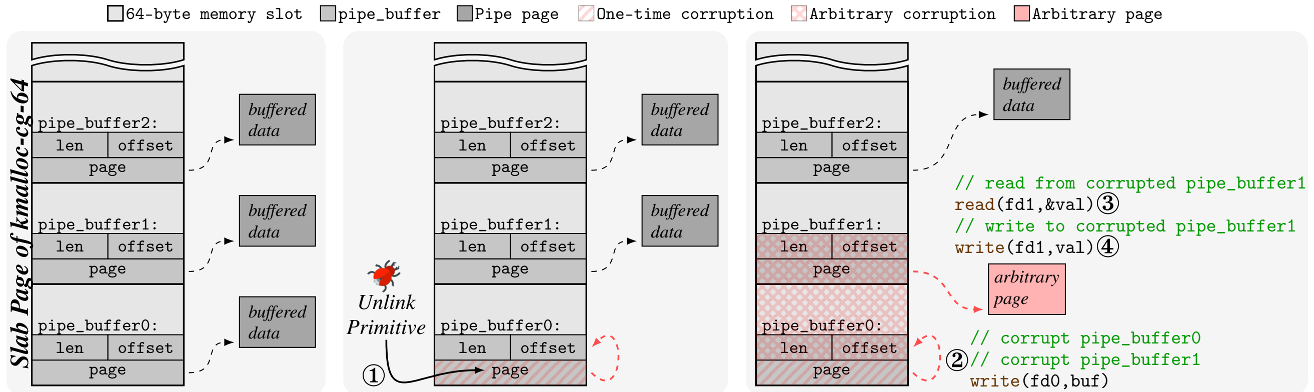


Figure 8: The exploit technique uses the unlink primitive and the known location of this slab page to obtain an arbitrary r/w. Initially, the unlink primitive ① corrupts the pipe buffer, i.e., `pipe_buffer0.page`, to point to its physical page. Writing with `fd0` ② corrupts the adjacent `pipe_buffer1`, whereby reading from ③ or writing to ④ (via `fd1`) enables the arbitrary r/w.

with all defenses in v6.9, including **D2**. The main cause of false positives for the **D2** exploit is that we could not apply the slab side channel [41] to `cred` and `seq_file`.

We also evaluated other Intel CPUs from the 8th to the most recent 14th generation with generic kernels ranging from v5.15 to v6.8. In particular, we exploited **D1** and **D3** as these defenses are integrated in these generic kernels. Table 2 shows the evaluation results of the stack disclosure attacks.

**Stress.** To test the resilience of our side channel, we test the stack disclosure attack against TLB pressure. We start `stress -m <nr_cpus-1> -vm-keep` to create memory stress with TLB pressure on all other cores, including the exploit's sibling hyperthread. Performance counters show that it causes around $60 \times 10^6$ dTLB misses per second, while the exploit causes around $4 \times 10^6$ misses, representing considerable stress. On our Raptor Lake, stressing the system causes the CR to drop to around 93 %. By monitoring the CPU frequency, we find this is caused almost entirely by the fluctuating frequency resulting from adaptive power management on the laptop CPU in line with prior findings [42]. When fixing the frequency, this TLB pressure has a negligible effect on the CR.

## 7 Reliable and Stable Kernel Exploitation

In this section, we discuss that by enabling defenses such as `CONFIG_SLAB_VIRTUAL` or `CONFIG_KMALLOC_CG`, kernel exploitation by pure vulnerabilities has been made much more difficult. In particular, we discuss that the inclusions prevent or severely limit the use of existing exploit techniques. We then show that with our location disclosure attack, which is counterintuitively possible due to some defenses, we re-enable the prevented exploit techniques or enable a new one.

In the following, we present three exploit techniques as case studies. These techniques exploit write primitives to perform privilege escalation with an exceptional reliability of more than 99.99 % on real hardware. First, we exploit the unlink primitive (see Section 7.1), which has been used by several real-world exploits [40, 48, 50, 51, 55, 58] but has also been largely mitigated by modern defenses. Second, we exploit the more generic UAF and OOB write (see Section 7.2), which is considered a weak exploitation primitive [41]. While it typically requires complex primitive conversions for privilege escalation, we present a stable and reliable exploit technique. Third, we exploit a constrained write primitive (see Section 7.3) where no read primitive is available. Prior work requires either complex primitive conversions [10, 18, 24, 52] or re-triggering the same [46] or another vulnerability [48,53], all of which come with the risk of crashes. For this primitive, we present a novel exploit technique for privilege escalation.

**Setup.** We implement Proof-Of-Concepts (POCs) for the following exploit techniques to escalate privileges. We also implement a helper kernel module that provides the initial primitive. We evaluate them on two configurations, Ubuntu 24.04 with the generic kernel v6.8 (i.e., **D1** and **D3** enabled) and the kernel v6.6 with the **D2** enabled (i.e., all three enabled). We run Ubuntu on real hardware, i.e., Intel 13th Gen i7-1360P and 32 GB of RAM. To show the reliability of our techniques, we repeat their execution 1000 times. A run is considered successful if we achieve privilege escalation. If a run results in a system crash, we explicitly mark it as a system crash. We repeat the 1000 executions for 10 reboots, also demonstrating reliability between different reboots.

### 7.1 Unlink Primitive

In this case study, we discuss the problem statement of exploiting the unlink primitive. We discuss that modern defenses either prevent or significantly increase the difficulty of its exploitation. We then show that with our exploit technique (exploiting **D1/2**), we obtain an arbitrary physical read/write, allowing us to escalate privileges.

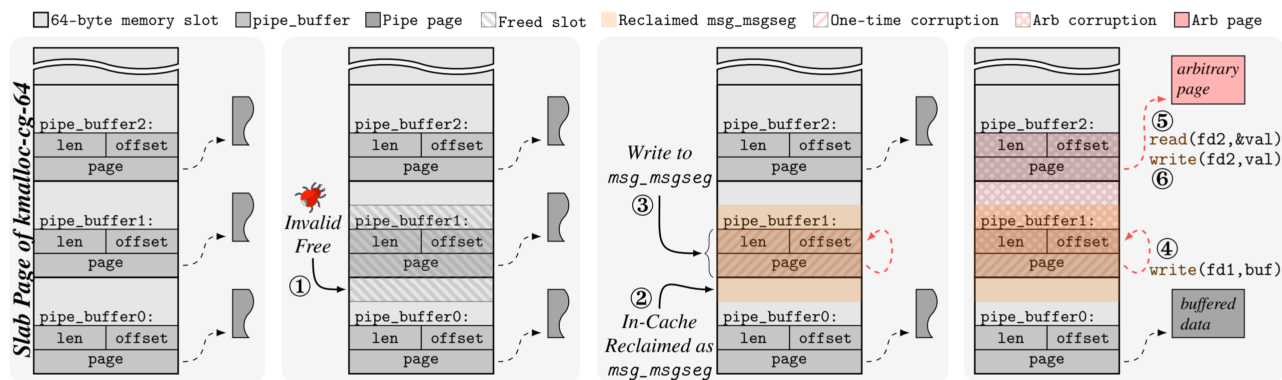


Figure 9: The exploit technique uses an IF and the known location of this slab page to obtain an arbitrary r/w. Initially, the IF ① frees a memory slot containing `pipe_buffer1`. In-cache reclaiming of this slot ② as a `msg_msgseg` enables corrupting `pipe_buffer1` to point to its physical page ③. Writing to with `fd1` ④ corrupts `pipe_buffer2`, while ⑤/⑥ allows arbitrary r/w.

To exploit the unlink primitive, a bad actor typically exploits a UAF to gain overwrite capability on an object with a linked-list member. They then trigger the overwrite to corrupt the linked-list member. When the corrupted object is unlinked, the following 2 writes are performed instead of unlinking the element: `*(next + 8) = prev; *(prev) = next;`. Various prior exploits convert this primitive to a more stable read or write primitive, e.g., exploits [40, 48, 50, 51, 58] do this by corrupting objects, such as `msg_msg` [48], or `seq_file` [50]. They typically trigger the unlink primitive multiple times to get a prior read primitive for leaking heap pointers. However, with the `CONFIG_KMALLOC_CG` defense, these pointer leaks have become much more difficult because the used security-critical objects are separated from vulnerable objects. Therefore, bad actors would need more powerful read primitives, which are typically not present at this stage and are difficult to transform by the unlink primitive itself.

**Reliable and Stable Exploit Technique.** We demonstrate the exploit technique to convert the unlink primitive into an arbitrary read/write by corrupting security-critical objects. While this is a generic technique usable with objects such as pipe_buffer [34], `file` [53, 56], or `seq_file` [50, 58], we detail it using the `pipe_buffer` as an example.

Figure 8 illustrates the high-level technique of exploiting `pipe_buffers`, which we detailed in Appendix A. Here, we require that the slab page's address is leaked and that all memory slots on this page are populated with `pipe_buffers`. We showed in Section 5 how to achieve both. The `pipe_buffer` is the kernel object created when a user calls `pipe2`, and acts as a physically-backed ring buffer. It provides operations to read data from and write data to this buffer, which contains `page`, `len`, and `offset` as members. While `pipe_buffer.page` refers to its physically-backed page used as the ring buffer, `pipe_buffer.len/offset` store the read and write end within the physical page. We initially trigger the unlink primitive ① to refer the target `pipe_buffer0.page` to the physical page it resides on. Consequently, writing to this physically-backed page via `fd0` now corrupts `pipe_buffer0/1` ②. We corrupt `pipe_buffer0` to enable arbitrary corruption of `pipe_buffer0/1`, while `pipe_buffer1` corruption allows us to read ③ from and write ④ to the controlled kernel address, i.e., `pipe_buffer1.page`. This results in the arbitrary physical read/write primitive.

We also provide an alternative technique, where we are required to leak a slab page populated with `pipe_buffers` and a page table, i.e., PUD. Here, we first use the unlink primitive to overwrite `pipe_buffer0.page` with the leaked page table. We then convert the single page-table overwrite to a physical read/write primitive similar to SLUBStick [41].

**Evaluation.** We implement a helper, allowing us to trigger an unlink primitive. We then implement two POCs that leverage this primitive to manipulate `pipe_buffer` for v6.8 and a `pipe_buffer`/page table for v6.6, obtaining the arbitrary read/write. Evaluating them results in 100 % reliability.

**Takeaway 5**

While modern kernel defenses largely mitigate the unlinking primitive, our exploit technique re-enables it.

## 7.2 Use-After-Free & Out-Of-Bounds Write

A common technique is to convert an in-cache write primitive due to a UAF or OOB bug into a Double-Free (DF) or an Invalid-Free (IF), as they are typically more powerful. Prior research [33, 40, 41] and real-world exploits [32, 46–48, 56, 64, 65] have followed this approach. In particular, prior work [41] has shown how to convert it to a DF or IF, which, in the following, we transform to an arbitrary read/write primitive.

**Reliable and Stable Exploit Technique.** Figure 9 illustrates the exploit technique to convert an IF to privilege escalation. Again, we require that the slab page address be leaked and that all memory slots of that slab page be populated with `pipe_buffers`. Since we know the memory slot layout of the slab page and its base address, we leverage the IF to free

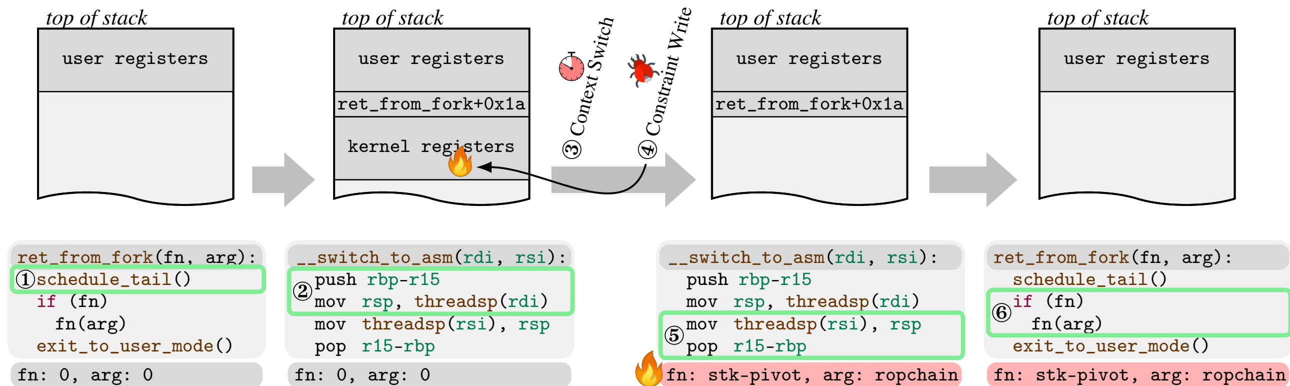


Figure 10: The exploit technique of register corruption for control-flow hijacking. A freshly cloned thread first executes `ret_from_fork`, which internally calls `schedule_tail` ① and saves the callee-saved registers to the kernel stack ②. It then performs a context switch ③, which takes a considerable time to continue. During this time, a bad actor uses the constrained write to corrupt the kernel registers ④. On register restoration ⑤, `fn/arg` are corrupted, resulting in kernel code execution ⑥.

a slot, marking `pipe_buffer1` as free ①. Subsequently, we in-cache reclaim the invalid slot as a `msg_msgseg` object ② via the `msgsnd` syscall. This is possible because both objects are allocated for the control-group allocator cache `kmalloc-cg-*`. After the reclaiming, the `msgsnd` syscall overwrites `pipe_buffer1` with attacker-controlled data, corrupting its members to refer to the physical page it resides on ③. Writing the corrupted physically-backed page via `fd1` now allows arbitrary corruption and control of `pipe_buffer1/2` ④. Controlling `pipe_buffer2` enables to read ③ from and write ④ to the controlled kernel address, i.e., `pipe_buffer2.page`, resulting in the arbitrary physical read/write.

**Evaluation.** We first implement a helper that allows us to free a controlled address, mimicking the capabilities of a UAF or OOB write to a free-able heap pointer. We then implement two POCs that leverage this free to manipulate `pipe_buffer` for v6.8 and a `pipe_buffer`/page table for v6.6, obtaining the arbitrary read/write. Evaluating them results in a 100 % and 99.99 % reliability, respectively, with no crashes.

**Takeaway 6**

The UAF or OOB write is a generic exploitation primitive. We showed a compelling and reliable exploit technique to convert this primitive to an arbitrary physical r/w.

### 7.3 Constrained Write Primitive

Exploits that only have a constrained write with no read primitive either perform complex primitive conversions [10, 18, 24, 52] or re-trigger the same [46] or another vulnerability [48, 53] to escalate privileges, all of which have the risk of crashes. Instead, we demonstrate 3 reliable and stable exploit techniques: First, we perform an approach similar to the technique presented in Section 7.1. Second, we overwrite a leaked `cred/file` directly. Third, we perform the following novel exploit technique for privilege escalation, which builds on a control-flow hijacking primitive from prior work [43].

**Reliable and Stable Exploit Technique.** Figure 10 illustrates the technique that hijacks the control flow, whereas Appendix A details it. Its prerequisites are to leak the location of the kernel stack and an object that contains the ROP chain. The first is solved by exploiting the side-channel leak of **D3** (see Section 5.3). The second is solved by exploiting the leakage of **D1/2** (see Section 5.1 or 5.2) and storing the ROP chain in all controlled objects (i.e., `msg_msg.mtext`) from the leaked slab page. Other options are to leak the DPM base address, store the ROP chain on user pages, and access one via the DPM, i.e., ret2dir [26].

A thread (i.e., *T0*) initially calls `clone`, which creates a new thread (i.e., *T1*) with a leaked stack location. On first *T1* scheduling, `ret_from_fork` inherently calls `schedule_tail` ①. This schedule function initiates a context switch, saving all *T1* callee-saved registers to its stack ②. It keeps the *T1* to sleep and switches the execution context to the next thread ③. At this stage, the entire state of the *T1* is stored to memory, while it will be restored at its next scheduling. During this time window, *T0* leverages the constrained write ④ to tamper with the stack, specifically where `fn` and `arg` are located. Since the kernel does not include any randomization at this stage, these locations can be determined with our known kernel stack location. After restoring the corrupted state of *T1* ⑤, it returns to `ret_from_fork+0x1a`, where it calls `fn(arg)` ⑥ and redirects the control flow.

**Evaluation.** We implement a helper for a constrained write primitive and a v6.8 POC for privilege escalation. The evaluation results in 100 % reliability.

**Takeaway 7**

While the security benefit of **D3** is low, it reliably allows a novel control-flow hijacking technique.

## 8 Discussion and Related Work

This section discusses the security implications of our attacks, the mitigation challenges, related work, and future work.

**Security Implications.** We have shown that while certain kernel defenses improve security in one dimension, they can reduce it in another. For our 3 identified exploitable defenses, we discussed that **D1/2** substantially limits the exploitation of kernel memory-corruption bugs. In contrast, **D3** provides little security benefit compared to other stack-based defenses, e.g., `CONFIG_STACKPROTECTOR` or `CONFIG_RANDOMIZE_KSTACK_OFFSET_DEFAULT`, in conjunction with the low incidence of stack-based corruption flaws. Since **D3** allows the kernel stack to be leaked and the exploit technique in Section 7.3, we argue that its security benefit doesn't outweigh its security drawback.

While we have shown the threat of our disclosure attacks, the actual impact is more severe. Consider kernel attacks such as the file UAF [56, 63], which requires a kernel heap pointer leak. With our location disclosure attacks, this heap pointer leak can be done reliably without the risk of crashing the system. The same is true for other exploit techniques.

Going one step further, in addition to the objects we leak due to kernel defense leakages, our disclosure attacks can be performed on other objects residing in 4 kB mappings. For instance, objects allocated via `vmalloc` (see Section 2) can be equally susceptible. If these objects also have appropriate allocation and access primitives, bad actors can leak their locations, making their exploitation more reliable. A notable example of such objects is the bytecode for the extended Berkeley Packet Filter (eBPF), where eBPF is widely used for network packet filtering, profiling, and monitoring. A recent exploit technique [3] demonstrated how a limited OOB write could manipulate the eBPF bytecode and achieve privilege escalation on the latest Ubuntu systems. A critical component of this attack is heap shaping, where the bytecode must be positioned at a specific offset to align with the OOB write constraints. With our location disclosure attacks, this heap-shaping step can be stabilized and verified, increasing the reliability of this exploit technique. As a result, beyond the security-critical kernel objects discussed in this paper, our disclosure attacks can be used generically to leak the locations of kernel objects allocated in memory regions mapped with 4 kB pages, further extending the scope of potential kernel exploitation.

**Mitigations.** The key factors of our exploit techniques are the leverage of *location disclosure attacks*, *exploit primitives*, or *TLB side channels*. Eliminating any of these can partially or fully prevent the techniques. First, the underlying problem with *disclosure attacks* is prevented by never placing kernel objects in memory slots mapped with 4 kB pages, thus eliminating **C1**. For **D1**, one solution is to have a dedicated page allocator cache for the physical pages of kernel modules. This results in the module code never sharing the same 2 MB memory area as kernel objects. Therefore, the mapping of kernel objects cannot be changed to 4 kB using **D1**, which prevents leakage of their location. For **D2/3**, one solution is to map the memory of these allocators with 2 MB. However, while these two solutions seem appealing, they must be developed with memory performance and memory reuse in mind, and require significant engineering effort to redesign various allocators.

Second, if bad actors cannot obtain *exploit primitives*, they cannot perform the exploit techniques. Security experts continue incorporating defenses that complicate converting UAF or OOB vulnerabilities into write primitives. For instance, combining the more-fined separation of heap objects – e.g., per call-site [9], which is under discussion, or user controllable [5], which will be included – with the cross-cache mitigation [45] complicates obtaining write primitives from UAF vulnerabilities. However, this only partially prevents the exploit techniques, as UAF [53, 58], which provides a more powerful write primitive, or OOB writes [3, 10, 37, 46], still cannot be mitigated.

Third, *TLB side channels* can be prevented either by software or hardware. Software mitigations include designing existing kernel memory management such that information disclosure is limited or defenses like FLARE [2], preventing the distinction between mapped and unmapped pages. Another mitigation is KPTI, which comes with significant performance overhead. In hardware, starting with Sierra Forest and Lunar Lake architectures, Intel CPUs will feature Linear Address-Space Separation (LASS) [21], separating kernel and user space addresses by the MSB of the virtual address. This terminates illegal accesses before any paging-based timing differences become visible and should prevent access-based attacks like double-page fault or prefetch [15, 20], though a different distinguishing mechanism might exist and be used.

**Prior Kernel Exploit Techniques.** Recently, there has been a burst of novel exploit techniques. Some also presented exploits combined with side-channel leaks [1,30,38,39,41,52]. Prior work has used the prefetch side channel [15] to leak the per-CPU entry area, either to hijack the control flow [52] or to store attacker-controlled data [13]. Liu et al. [38, 39] demonstrated a KASLR break even with KPTI enabled and a vulnerability exploit combined with a kernel base leak via the prefetch side channel. However, these works only leak coarse-grained locations, e.g., the kernel base and per-CPU entry area, while we presented location leakage of kernel heap objects, page tables, and stacks. Lee et al. [30] and Maar et al. [41] presented a side channel on the slab allocator to make heap spraying and cross-cache attacks more reliable. In addition, many other privilege escalation techniques have been proposed. These include DirtyCred [33], which exchanges low-privileged with high-privileged `creds` for privilege escalation and Dirty PageTable [65], Dirty PageDirectory [47], and SLUBStick [41], which exploits a write primitive on a page table for an arbitrary read/write.

**TLB-based Side-Channel Attacks.** Prior work has used

TLB side channels to break aspects of KASLR. In 2013, Hund et al. [20] used the timing difference of page faults that depend on the TLB to detect the mapping layout of the kernel and locate specific code executed by a syscall or driver. Gruss et al. [15] demonstrated that the `prefetch` instruction can be used to break code KASLR and find driver locations on Intel, and Lipp et al. [36] later extended this work to AMD. TLB reverse-engineering efforts have revealed the workings of TLBs and demonstrated that they can also be attacked in similar ways as caches [14, 29, 59, 70]. They reverse-engineer the dimensions and properties of the TLB structures on Intel CPUs and their addressing functions and tagging functionality, which can also be used to break KASLR.

**Software-Only Location Leaks.** While most prior work has focused on leaking code or physical KASLR [15, 20, 29, 38], Maar et al. [42] have shown how to leak kernel heap pointers via a hardware-agnostic software side channel.

**Future Work.** We may extend to AMD and ARM, as the TLB side channel is present [15, 36]. However, AMD has the additional complication that the TLB caches unmapped translations for `prefetch`, which would require a slightly different treatment of the mapped/unmapped distinction.

## 9 Conclusion

Based on a systematic analysis of 127 defenses, we showed that 3 of them create exploitable, fine-grained TLB contention patterns. By combining strategic kernel allocator massaging with these patterns, we presented location disclosure attacks that leak the locations of security-critical kernel objects. We demonstrated that our attacks enable reliable and stable exploitation of kernel vulnerabilities even on the latest Linux kernel and across a wide range of Intel CPUs and kernel versions. With an attack runtime of 0.3 s to 17.8 s and almost no false positives, we showed that our attack is highly practical. We concluded that while defenses close the door to one attack variant, e.g., vulnerability exploitation, they may open the door to another, e.g., side-channel leakage.

## Acknowledgements

This research was funded in whole or in part by the Austrian Science Fund (FWF) [SFB project SPyCoDe 10.55776/F85], the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087), and the European Research Council (ERC project FSSec 101076409). Additional funding was provided by a generous gift from Intel. 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.

## 10 Ethics Considerations

We have followed standard responsible disclosure practices by disclosing our findings to the Linux kernel security team's mailing list before the submission, leaving more than 90 days until the earliest date of publication.

Our experiments were conducted locally on our own machines without involvement of third-party participants or data.

## 11 Open Science

All distinct experiments/exploits discussed in the paper will be published and made available as POCs for the artifact evaluation[5].

## References

[1] *EntryBleed: A Universal KASLR Bypass against KPTI on Linux*, 2023.

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

[3] Pumpkin Chang. How I use a novel approach to exploit a limited OOB on Ubuntu at Pwn2Own Vancouver 2024, 2024. URL: https://u1f383.github.io/slides/talks/2024_POC-How_I_use_a_novel_approach_to_exploit_a_limited_OOB_on_Ubuntu_at_Pwn2Own_Vancouver_2024.pdf.

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

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

[6] Jonathan Corbet. Solutions for direct-map fragmentation, 5 2022. URL: https://lwn.net/Articles/894557/.

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

[8] Jonathan Corbet. The proper time to split struct page, 2023. URL: https://lwn.net/Articles/937839.

[9] Jonathan Corbet. Per-call-site slab caches for heap-spraying protection, 2024. URL: https://lwn.net/Articles/986174/.

[5] https://zenodo.org/records/14736361

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

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

[12] Jake Edge. Control-flow integrity for the kernel, 2020. URL: https://lwn.net/Articles/810077/.

[13] Google. kernelCTF rules, 2023. URL: https://google.github.io/security-research/kernelctf/rules.html.

[14] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*, 2018.

[15] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*, 2016.

[16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[17] Ziyi Guo, Dang K Le, Zhenpeng Lin, Kyle Zeng, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, Adam Doupé, and Xinyu Xing. Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation. In *USENIX Security*, 2024.

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

[19] Hongli Han, Rong Jian, Xiaodong Wang, and Peng Zhou. Typhoon Mangkhut: One-click Remote Universal Root Formed with Two Vulnerabilities, 2021. URL: https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Typhoon-Mangkhut-One-Click-Remote-Universal-Root-Formed-With-Two-Vulnerabilities.pdf.

[20] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[21] Intel. Intel Architecture Instruction Set Extensions Programming Reference, 2024.

[22] Jann Horn. Mitigations are attack surface, too, 2020. URL: https://googleprojectzero.blogspot.com/2020/02/mitigations-are-attack-surface-too.html.

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

[24] 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/.

[25] Xingyu Jin and Richard Neal. The Art of Exploiting UAF by Ret2bpf in Android Kernel, 2021. URL: https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf.

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

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

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

[29] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *EuroS&P*, 2020.

[30] 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.

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

[32] Zhenpeng Lin, Yueqi Chen, Xinyu Xing, , and Kang Li. Your Trash Kernel Bug, My Precious 0-day, 2021. URL: https://www.blackhat.com/eu-21/briefings/schedule/#your-trash-kernel-bug-my-precious--day-24849.

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

[34] 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.

[35] Linux Kernel Driver DataBase. CONFIG_VMAP_-STACK: Use a virtually-mapped stack, 2024. URL: https://cateee.net/lkddb/web-lkddb/VMAP_STACK.html.

[36] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD Prefetch Attacks through Power and Time. In *USENIX Security*, 2022.

[37] William Liu. CVE-2022-0185 - Winning a $31337 Bounty after Pwning Ubuntu and Escaping Google's KCTF Containers, 2022. URL: https://www.willsroot.io/2022/01/cve-2022-0185.html.

[38] William Liu. EntryBleed: Breaking KASLR under KPTI with Prefetch (CVE-2022-4543), 2022. URL: https://www.willsroot.io/2022/12/entrybleed.html.

[39] William Liu. corCTF 2023 sysruption - Exploiting Sysret on Linux in 2023, 2023. URL: https://www.willsroot.io/2023/08/sysruption.html.

[40] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In *USENIX Security*, 2024.

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

[42] Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In *NDSS*, 2025.

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

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

[45] Ingo Molnar. Re: [RFC PATCH 00/14] Prevent cross-cache attacks in the SLUB allocator, 2023. URL: https://lore.kernel.org/all/CAHKB1wLetbLZjhg1UVhA1QwZHo226BRL=Khm962JEfh0F+CVbQ@mail.gmail.com/T/.

[46] Andy Nguyen. CVE-2021-22555: Turning x00x00 into 10000$, 2021. URL: https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html.

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

[48] 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.

[49] James Randall. pipe_buffer arbitrary read write, 2022. URL: https://www.interruptlabs.co.uk/articles/pipe-buffer.

[50] Eloi Sanfelix. A bug collision tale, 2020. URL: https://labs.bluefrostsecurity.de/files/OffensiveCon2020_bug_collision_tale.pdf.

[51] Blue Frost Security. Exploiting CVE-2020-0041 - Part 2: Escalating to root, 2020. URL: https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/.

[52] Seth Jenkins. Exploiting CVE-2022-42703 - Bringing back the stack attack, 2022. URL: https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html.

[53] Seth Jenkins. Analyzing a Modern In-the-wild Android Exploit, 2023. URL: https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html.

[54] Seth Jenkins. Driving forward in Android drivers, 2024. URL: https://googleprojectzero.blogspot.com/2024/06/driving-forward-in-android-drivers.html.

[55] Maddie Stone. Bad Binder: Android In-The-Wild Exploit, 2019. URL: https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[56] Maddie Stone. A Very Powerful Clipboard: Analysis of a Samsung in-the-wild exploit chain, 2022. URL: https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html.

[57] Maddie Stone. The Ups and Downs of 0-days: A Year in Review of 0-days Exploited In-the-Wild in 2022, 2023. URL: https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html.

[58] Zi Fan Tan, Gulshan Singh, and Eugene Rodionov. Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938, 2024. URL: https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938/#unlink-primitive.

[59] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering. In *USENIX Security*, 2022.

[60] The Linux Kernel. Kernel Self-Protection, 2024. URL: https://docs.kernel.org/security/self-protection.html.

[61] Eduardo Vela. Making Linux Kernel Exploit Cooking Harder, 2022. URL: https://security.googleblog.com/2022/08/making-linux-kernel-exploit-cooking.html.

[62] 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.

[63] Yong Wang. Ret2page: The Art of Exploiting Use-After-Free Vulnerabilities in the Dedicated Cache, 2022. URL: https://i.blackhat.com/USA-22/Thursday/US-22-WANG-Ret2page-The-Art-of-Exploiting-Use-After-Free-Vulnerabilities-in-the-Dedicated-Cache.pdf.

[64] Le Wu and Qi Zhang. Game of Cross Cache: Let's win it in a more effective way!, 2024. URL: https://i.blackhat.com/Asia-24/Presentations/Asia-24-Wu-Game-of-Cross-Cache.pdf.

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

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

[67] 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.

[68] Ptr Yudai. Understanding Dirty Pagetable - m0leCon Finals 2023 CTF Writeup, 2023. URL: https://ptr-yudai.hatenablog.com/entry/2023/12/08/093606.

Table 2: Evaluation results on various CPUs and kernel versions of the **D3** exploit to leak the location of kernel stacks.

| CPU Model | Architecture | Kernel | SR | T | CR |
|---|---|---|---|---|---|
| i7-8650U | Kaby Lake (8th Gen) | v6.8 | 100 | 0.2 | 100 |
| i9-9900K | Coffee Lake (9th Gen) | v5.15 | 97 | 1.4 | 100 |
| i7-1260P | Alder Lake (12th Gen) | v6.5 | 92 | 0.3 | 99 |
| | | v6.8 | 97 | 0.3 | 100 |
| i7-1270P | Alder Lake (12th Gen) | v5.15 | 99 | 0.4 | 100 |
| i7-1360P | Raptor Lake (13th Gen) | v6.8 | 98 | 0.3 | 100 |
| Ultra 7 155H | Meteor Lake (14th Gen) | v6.8 | 96 | 0.2 | 97 |

[69] 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.

[70] Weixi Zhu. Exploring superpage promotion policies for efficient address translation. Master's thesis, 2019.

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

## A Detailed Exploitation

This section details the reliable and stable exploitation.

**Unlink Primitive.** As shown in Figure 11, the `pipe_buffer` stores the physical-backed page via a `page` reference of the `vmemmap` region. Before corruption, the `pipe_buffer` refers to the `pipe_page` ❶ that stores the user data. On triggering the unlink primitive ❷, `*(pipe_buffer.page)` is overwritten with `slot_page`, while `*(slot_page+8)` is overwritten with `pipe_buffer.page`. The first overwrite is willing by the bad actor to refer the `pipe_buffer` with the physical page it resides on ❸, while the second overwrite is an unwilling artifact of the unlink primitive. However, unwilling writing does not affect any functionality as it corrupts currently unused data [8]. As a result of this unlink primitive triggering, writing to the pipe via its file descriptor now corrupts this `pipe_buffer` and all adjacent ones.

In addition to the leaked slab page, we need to leak `anon_pipe_buf_ops` (stored in `ops`) and `vmemmap[slot_pfn/pipe_pfn]`. Obtaining `anon_pipe_buf_ops` is straightforward, as we can use the TLB side channel to leak the kernel base address and increment the `ANON_PIPE_BUF_OPS_OFFSET`, obtained by the kernel binary. Obtaining `vmemmap[pfn]` works as follows: First, we leak `vmemmap_base` via the TLB side channel. Second, since the `vmemmap` is indexed by the physical frame number, we can reconstruct the virtual address with the leaked `vmemmap_base` and the physical address of `slot_page`.

**Constrained Write Primitive.** Since a gadget that performs stack pivoting within one instruction sequence is hard

Table 3: Allocation and access primitives for allocating and accessing kernel objects.

<table>
<tr><th>Kernel Objects</th><th>Allocation Primitive</th><th>Access Primitive</th></tr>
<tr><td>msg_msg</td><td>int qid = msgget(IPC_PRIVATE, 0666 | IPC_CREAT);<br>struct msg message = {.mtype = MSG_TYPE};<br>msgsnd(qid, &message, MSG_SIZE, 0);</td><td>struct msg message = {.mtype = MSG_TYPE};<br>msgrcv(qid, &message, MSG_SIZE, 0, MSG_COPY|IPC_NOWAIT);</td></tr>
<tr><td>cred</td><td>unshare(CLONE_NEWUSER);<br>open("/proc/$PID/ns/user");</td><td>getuid();</td></tr>
<tr><td>file</td><td>int fd = open("/path/to/file");</td><td>struct stat buf;<br>fstat(fd, &buf);</td></tr>
<tr><td>seq_file</td><td>int fd = open("/proc/self/stat");</td><td>lseek(fd, 0, SEEK_SET);</td></tr>
<tr><td>pipe_buffer</td><td>int pipe[2];<br>pipe2(pipe, O_NONBLOCK);<br>fcntl(pipe[0], F_SETPIPE_SZ, 8192);<br>char buffer[0x1000] = {0};<br>write(pipe[1], buffer, 8);</td><td>read(pipe[0], (void *)0xdeadbeef000, 8);</td></tr>
<tr><td>PUD, PMD, and PT</td><td>void *addr = mmap(ADDR, PAGE_SIZE, 0,<br>MAP_PRIVATE | MAP_ANON | MAP_FIXED, -1, 0);</td><td>alternating between:<br>mprotect(addr, PROT_WRITE);<br>mprotect(addr, PROT_READ);</td></tr>
<tr><td>Kernel stack</td><td>clone();</td><td>syscall(-1);</td></tr>
</table>

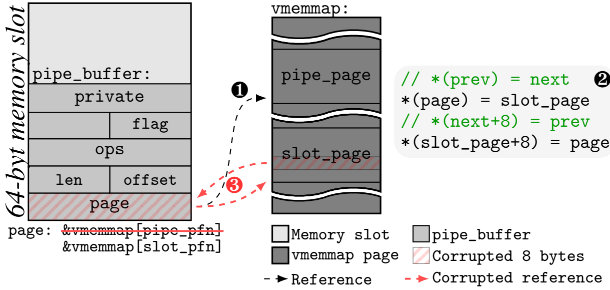


Figure 11: The detailed exploit technique of using the unlink primitive to allow arbitrary overwriting of a page containing pipe_buffer.

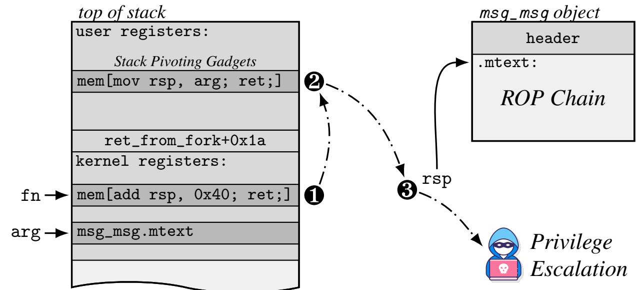


Figure 12: Technique to turn a write primitive into privilege escalation. First, a bad actor tampers ❶ with of fn and arg located on the stack. The call to fn adjusts the rsp to point to stack pivoting gadgets ❷ passed to the kernel via user registers. Pivoting then overwrites the rsp with msg_msg.mtext where the ROP chain for privilege escalation is located ❸.

to find in the Linux kernel [66, 69], we leverage user space data already present on the kernel stack as a first stage of the ROP attack. As depicted in Figure 12, we prepare user registers, which are then spilled to the kernel stack of *T0* by the syscall, i.e., clone. These user registers contain the location of ROP gadgets to perform stack pivoting within multiple instruction sequences and will be copied to the kernel stack of *T1* during cloning. When *T1* is put to sleep by the context switch, *T0* overwrites fn with the location of a gadget that increases rsp to reference the stack pivoting gadgets, i.e., mem[add rsp, 0x40; ret;] ❶. *T0* also overwrites arg with the ROP chain location, i.e., msg_msg.mtext. When the control flow is redirected to the stack pivoting gadgets ❷, they overwrite the stack pointer with msg_msg.mtext and initiate the ROP chain to perform privilege escalation ❸. While we performed an ROP attack in this example, JOP would be another possibility. Zeng et al. [69] have demonstrated a systematic analysis of the use of user registers to initiate control-flow hijacking attacks.

## B Page-Table Contention Patterns

Figure 13 illustrates the workflow of leaking the location of the page table ❸. We first allocate three pages with fixed virtual addresses, whose address translation is as follows: The addr0 uses page tables ❶→❷→❸→❹ to refer to its page, the addr1 uses page tables ❶→❷→❸→❺ to refer to its page, and the addr2 uses page tables ❶→❷→❻→❼ to refer to its page. Executing the mprotect syscall performs a software page-table walk and loads the physical addresses (accessible via the DPM) of the page tables into the TLB, while also accessing other kernel objects, e.g., vma_struct. Similar to the approach in Section 5.1, we call mprotect with different addresses to create common and differential patterns. From these patterns, we deduce the physical address of the page table ❸. In particular, calling mprotect with addr0 ① creates a common pattern with addr1 ②, as both use page table ❸, resulting in a pattern of ❶, ❷, and ❸. Conversely, calling mprotect with addr2 ③ creates a differential pat-

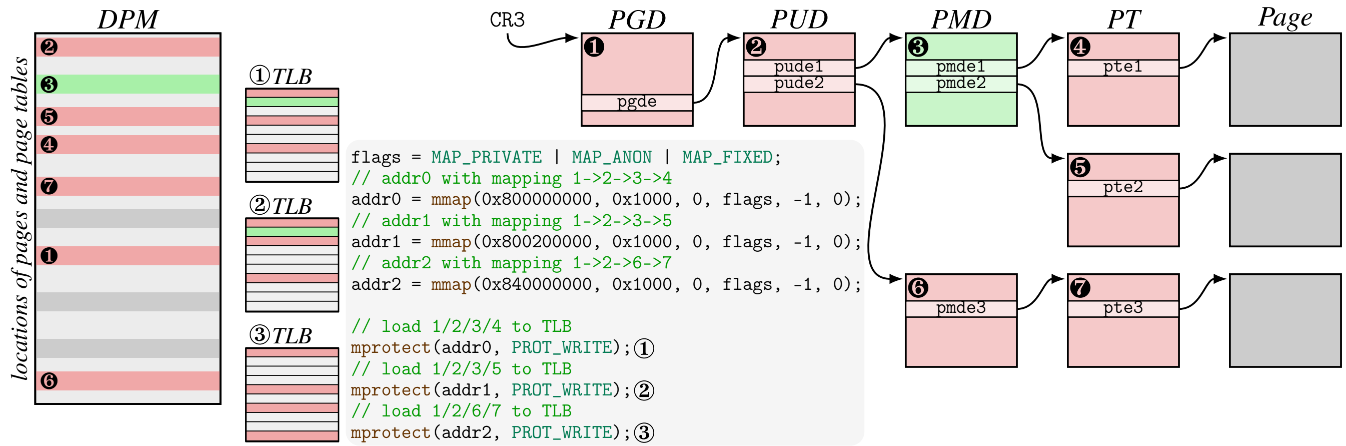


Figure 13: By strategically allocating user memory, we get an address translation for `addr0` with ❶/❷/❸/❹, `addr1` with ❶/❷/❸/❺, and `addr2` with ❶/❷/❻/❼. Executing `mprotect` performs a software page-table walk (i.e., ①, ②, and ③) that creates contention patterns of the page table's physical addresses, allowing to leak the address of *PT*.

tern to `addr0` of ❶ and ❷. Eliminating this pattern from the common one results in the correct derived page table ❸.