Take a Step Further: Understanding **Page Spray** in Linux Kernel Exploitation

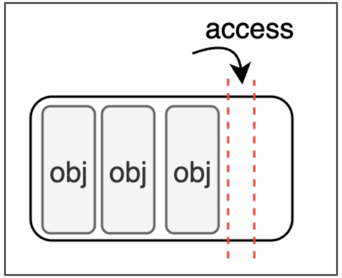
Ziyi Guo, Dang K Le, Zhenpeng Lin, Kyle Zeng,
Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, Adam Doupé, and Xinyu Xing

Northwestern University

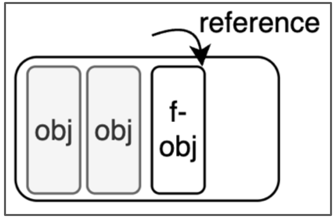
Arizona State University

Vulns in Linux Kernel

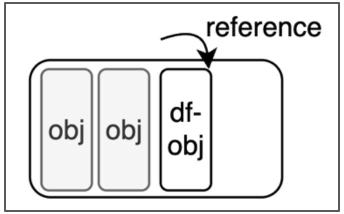
- **Out-of-bounds**
  - Access memory address based on object, but the address is actually out of the boundary of the current object.



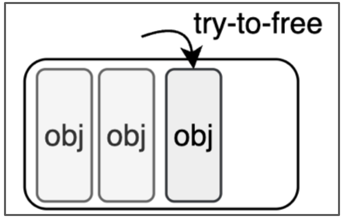
- **Use-after-free**
  - Access the object after it has been freed/discarded.



- **Double-free**
  - Free the object twice, confuse the system.



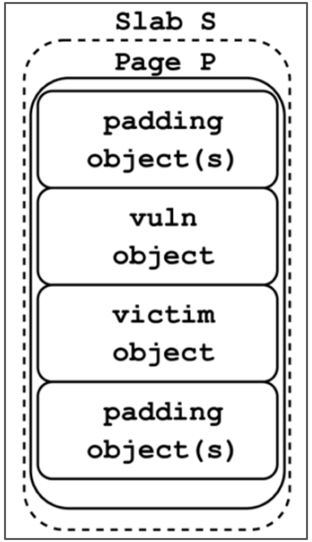
- **Invalid-free**
  - Free an address which is not the correct address of an object.



Linux Kernel Memory Management

- Heap Allocator
  - Used for **Objects-Based Management.**
  - Multiple different cache size: *kmalloc-256/kmalloc-1024* ....
  - **Built on the top of Slab Pages!**

```
 */
alloc_gfp = (flags | __GFP_NOWARN | __GFP_NORETRY) & ~__GFP_NOFAIL;
if ((alloc_gfp & __GFP_DIRECT_RECLAIM) && oo_order(oo) > oo_order(s->min))
        alloc_gfp = (alloc_gfp | __GFP_NOMEMALLOC) & ~__GFP_RECLAIM;

slab = alloc_slab_page(alloc_gfp, node, oo);
```

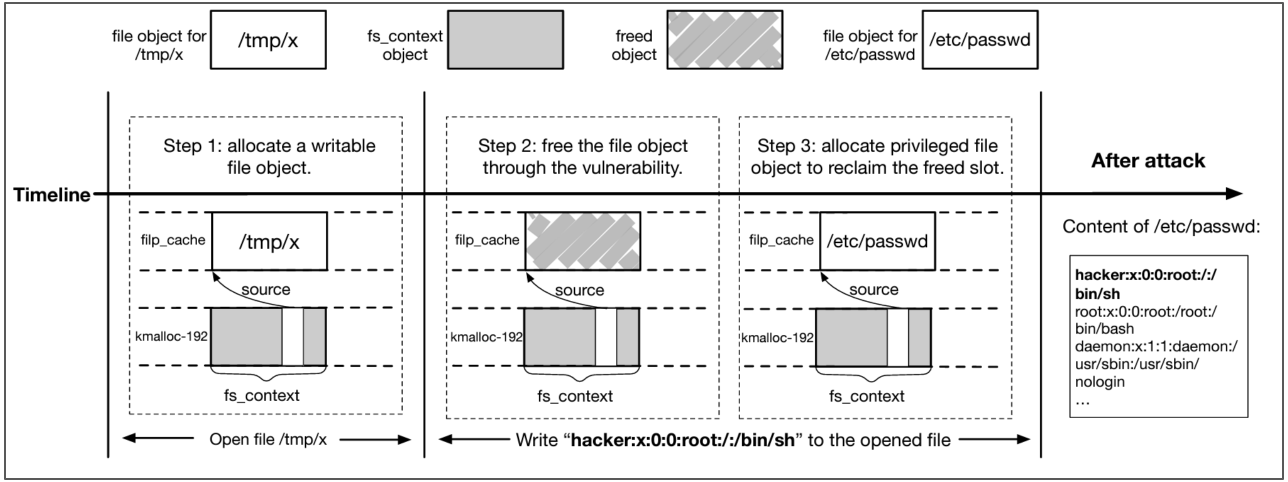


- Page Allocator
  - Used for **Pages Management**
  - Buddy System
  - **Fundamental mechanism for the system memory management!**

```
/*
 * This is the 'heart' of the zoned buddy allocator.
 */
struct page *__alloc_pages(gfp_t gfp, unsigned int order, int preferred_nid,
                                                        nodemask_t *nodemask)
{
```

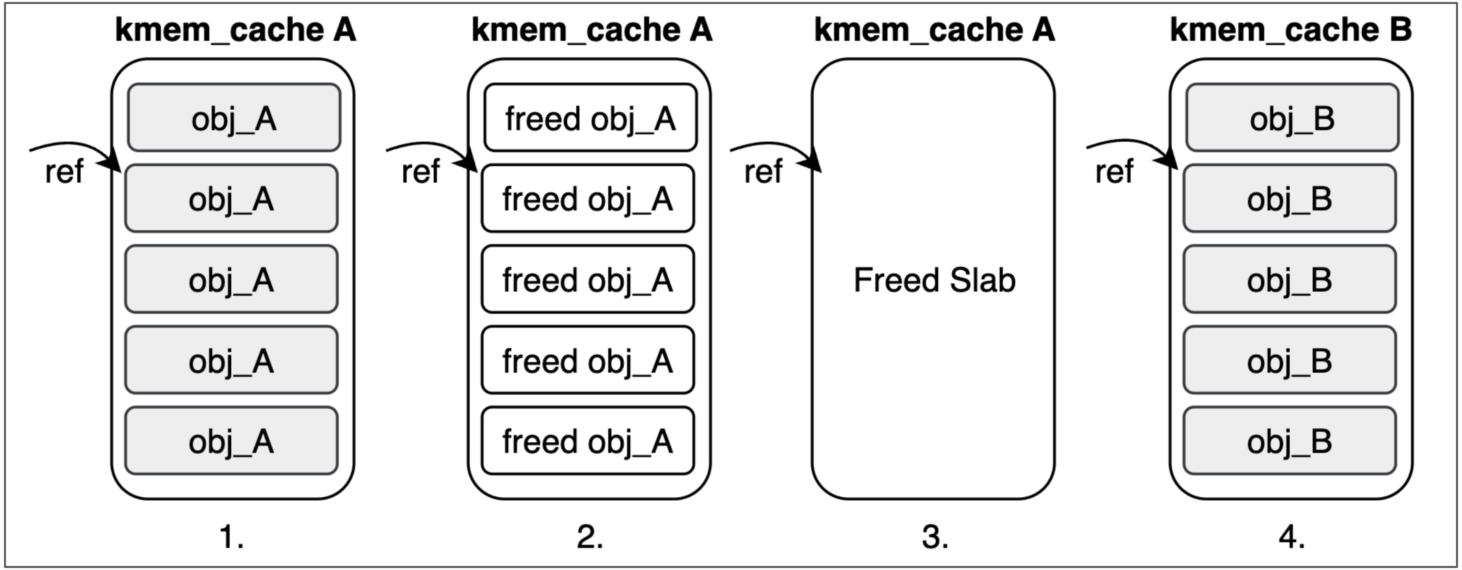
Exploits in Linux Kernel

- **DirtyCred(ACM CCS'22, Blackhat USA'22)**
  - **Use an object temporal vulnerability as starting point.**
  - **Maintain a reference to the writable object spot.(allocation first)**
  - **Free the writable objects.**
  - **Reclaim the freed slot with privileged objects**
  - **Now you can a reference to privileged object!!!**



Exploits in Linux Kernel

- **Cross Cache Attack**



Free Pages Reclaim

```c
slab_empty:
        if (prior) {
                /*
                 * Slab on the partial list.
                 */
                remove_partial(n, slab);
                stat(s, FREE_REMOVE_PARTIAL);
        } else {
                /* Slab must be on the full list */
                remove_full(s, n, slab);
        }

        spin_unlock_irqrestore(&n->list_lock, flags);
        stat(s, FREE_SLAB);
        discard_slab(s, slab);
}
```

```c
static void discard_slab(struct kmem_cache *s, struct slab *slab)
{
        dec_slabs_node(s, slab_nid(slab), slab->objects);
        free_slab(s, slab);
}
```

```c
static void __free_slab(struct kmem_cache *s, struct page *page)
{
        int order = compound_order(page);
        int pages = 1 << order;

        if (kmem_cache_debug_flags(s, SLAB_CONSISTENCY_CHECKS)) {
                void *p;

                slab_pad_check(s, page);
                for_each_object(p, s, page_address(page),
                                                page->objects)
                        check_object(s, page, p, SLUB_RED_INACTIVE);
        }

        __ClearPageSlabPfmemalloc(page);
        __ClearPageSlab(page);
        /* In union with page->mapping where page allocator expects NULL */
        page->slab_cache = NULL;
        if (current->reclaim_state)
                current->reclaim_state->reclaimed_slab += pages;
        unaccount_slab_page(page, order, s);
        __free_pages(page, order);
}
```

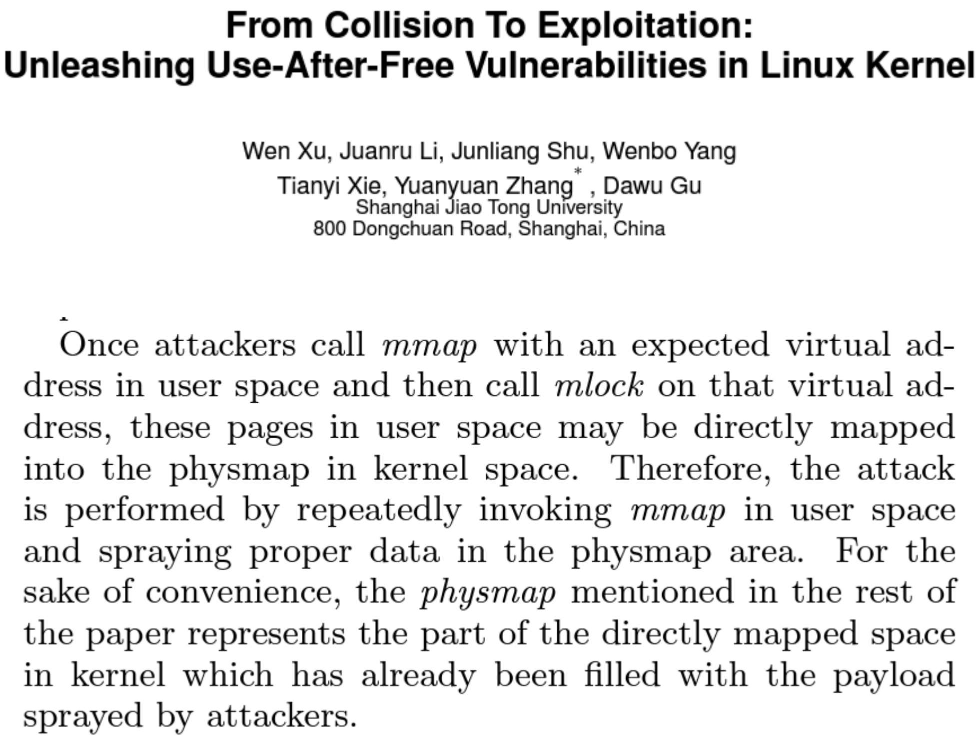
```c
static struct slab *new_slab(struct kmem_cache *s, gfp_t flags, int node)
{
        if (unlikely(flags & GFP_SLAB_BUG_MASK))
                flags = kmalloc_fix_flags(flags);

        WARN_ON_ONCE(s->ctor && (flags & __GFP_ZERO));

        return allocate_slab(s,
                flags & (GFP_RECLAIM_MASK | GFP_CONSTRAINT_MASK), node);
}
```

```c
alloc_gfp = (flags | __GFP_NOWARN | __GFP_NORETRY) & ~__GFP_NOFAIL;
if ((alloc_gfp & __GFP_DIRECT_RECLAIM) && oo_order(oo) > oo_order(s->min))
        alloc_gfp = (alloc_gfp | __GFP_NOMEMALLOC) & ~__GFP_RECLAIM;

slab = alloc_slab_page(alloc_gfp, node, oo);
```

```c
static inline struct slab *alloc_slab_page(gfp_t flags, int node,
                struct kmem_cache_order_objects oo)
{
        struct folio *folio;
        struct slab *slab;
        unsigned int order = oo_order(oo);

        if (node == NUMA_NO_NODE)
                folio = (struct folio *)alloc_pages(flags, order);
        else
                folio = (struct folio *)__alloc_pages_node(node, flags, order);

        if (!folio)
                return NULL;

        slab = folio_slab(folio);
        __folio_set_slab(folio);
        if (page_is_pfmemalloc(folio_page(folio, 0)))
                slab_set_pfmemalloc(slab);

        return slab;
}
```

Is that possible we do not reclaim the pages by heap allocator or slab allocator? 🤔

Flash back....


From Collision To Exploitation:
Unleashing Use-After-Free Vulnerabilities in Linux Kernel

Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang
Tianyi Xie, Yuanyuan Zhang*, Dawu Gu
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai, China

Once attackers call *mmap* with an expected virtual address in user space and then call *mlock* on that virtual address, these pages in user space may be directly mapped into the physmap in kernel space. Therefore, the attack is performed by repeatedly invoking *mmap* in user space and spraying proper data in the physmap area. For the sake of convenience, the *physmap* mentioned in the rest of the paper represents the part of the directly mapped space in kernel which has already been filled with the payload sprayed by attackers.

Understanding the Root Cause

- **Allocation + Copy Write**

```c
struct pipe_buffer {
 struct page *page;
 unsigned int offset, len;
 const struct pipe_buf_operations *ops;
 ...
};
static ssize_t
pipe_write(..., struct iov_iter *from) {
for (;;) {
 if (!page) {
 page = alloc_page(GFP_HIGHUSER | __GFP_ACCOUNT);
 ...
 }
 buf->page = page;
 copied = copy_page_from_iter(page, 0, PAGE_SIZE, from);
 }
}
```

raw page-level buffer

```c
typedef struct bio_vec skb_frag_t;
static int packet_snd(struct socket *sock, struct msghdr *msg, size_t len) {
 ...
 skb = packet_alloc_skb(sk, hlen + tlen, hlen, len, linear, msg->msg_flags & MSG_DONTWAIT, &err);
 ...
 err = skb_copy_datagram_from_iter(skb, offset, &msg->msg_iter, len);
}
```

```c
skb = alloc_skb_with_frags(header_len, data_len, max_page_order,
                           errcode, sk->sk_allocation);
```

```c
for (i = 0; npages > 0; i++) {
        int order = max_page_order;

        while (order) {
                if (npages >= 1 << order) {
                        page = alloc_pages((gfp_mask & ~__GFP_DIRECT_RECLAIM) |
                                           __GFP_COMP |
                                           __GFP_NOWARN,
                                           order);
                        if (page)
                                goto fill_page;
                        /* Do not retry other high order allocations */
                        order = 1;
                        max_page_order = 0;
                }
                order--;
        }
```

Non-linear Page-Frags Buffer

Understanding the Root Cause

- **mmap() & zero copy**

```c
static struct pgv *alloc_pg_vec(struct tpacket_req *req, int order){
 unsigned int block_nr = req->tp_block_nr;
 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);
 ...
 }
}
```

```c
static __cold int io_uring_mmap(struct file *file, struct vm_area_struct *vma)
{
        size_t sz = vma->vm_end - vma->vm_start;
        unsigned long pfn;
        void *ptr;

        ptr = io_uring_validate_mmap_request(file, vma->vm_pgoff, sz);
        if (IS_ERR(ptr))
                return PTR_ERR(ptr);

        pfn = virt_to_phys(ptr) >> PAGE_SHIFT;
        return remap_pfn_range(vma, vma->vm_start, pfn, sz, vma->vm_page_prot);
}
```

```c
static int packet_mmap(..., struct vm_area_struct *vma){
...
for (i = 0; i < rb->pg_vec_len; i++) {
 struct page *page;
 void *kaddr = rb->pg_vec[i].buffer;
 int pg_num;
 for (pg_num = 0; pg_num < rb->pg_vec_pages; pg_num++) {
 page = pgv_to_page(kaddr);
 err = vm_insert_page(vma, start, page);
 ...
 }
 }
}
```

Model of Page Spray

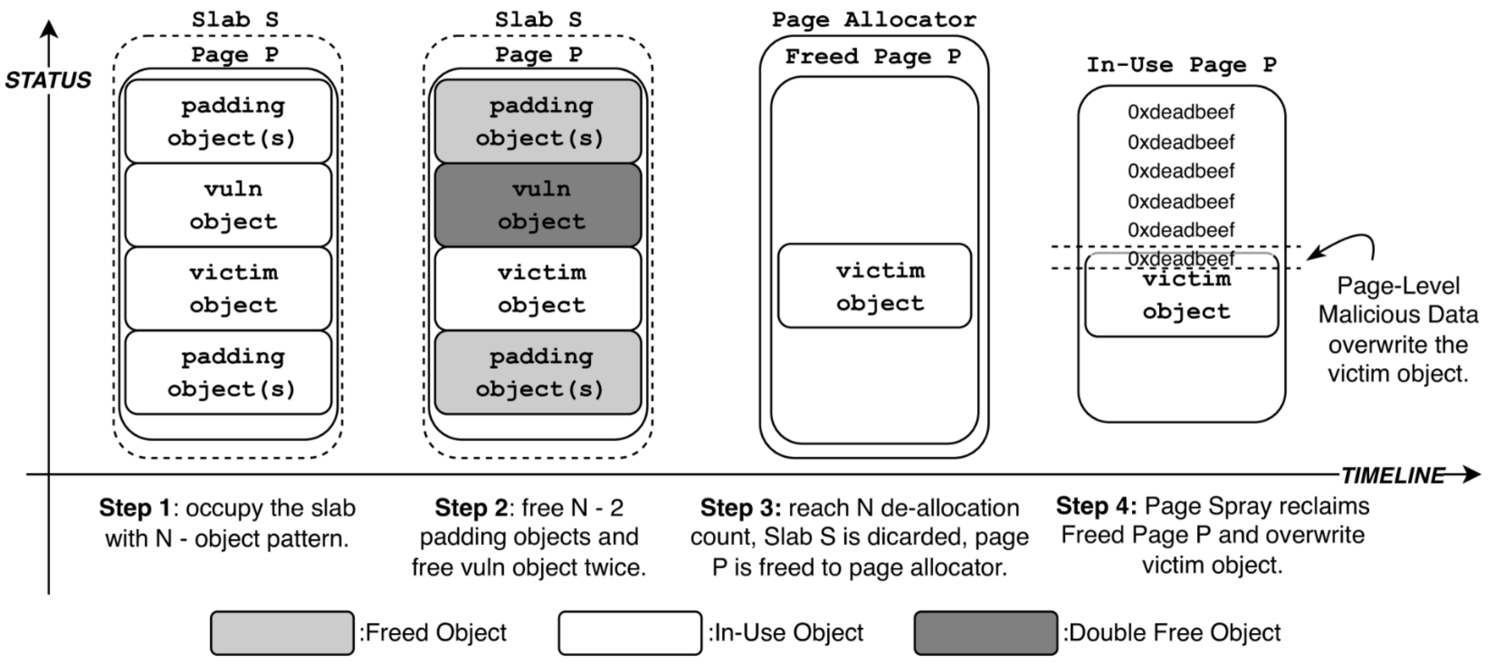


Figure 1: Page Spray Exploit Model for Double Free.

Callsite Examples

- packet_snd
- packet_mmap
- tcp_send_rcvq
- pipe_write
- io_uring_mmap
- aead_sendmsg
- skcipher_sendmsg
- mptcp_sendmsg
- rds_message_copy_from_user
- ……

Callsite	**Usability**	**Syscall**
packet_set_ring	●	setsockopt
packet_snd	●	sendmsg
packet_mmap	●	mmap
rds_message_copy_from_user	●	sendmsg
unix_dgram_sendmsg	◐†	sendmsg
unix_stream_sendmsg	◐†	sendmsg
netlink_sendmsg	◐✜	sendmsg
tcp_send_rcvq(inet6)	●	sendto
tcp_send_rcvq	●	sendto
tun_build_skb	◐✝	write
tun_alloc_skb	◐✝	write
tap_alloc_skb	◐✝	write
pipe_write	●	write
fuse_do_ioctl	◐†	ioctl
io_uring_mmap	●	mmap
array_map_mmap	◐†	mmap
ringbuf_map_mmap	◐†	mmap
aead_sendmsg	●	sendmsg
skcipher_sendmsg	●	sendmsg
mptcp_sendmsg	●	sendmsg
xsk_mmap	◐☨	mmap

Exploitability

CVE-ID	Type	Object Spray	Page Spray
CVE-2016-4557	UAF	✔	✔
CVE-2016-8655	UAF	✔	✔
CVE-2017-10661	UAF	✔	✔
CVE-2017-11176	UAF	✔	✔
CVE-2017-15649	UAF	✔	✔
CVE-2018-6555	UAF	✔	✔
CVE-2016-0728	OOB	✔	✔
CVE-2021-22555	OOB	✔	✔
CVE-2022-2588	DF	✔	✔
CVE-2017-6074	DF	✔	✔
CVE-2017-8890	DF	✔	✔
CVE-2022-29581 †	UAF	✔	✔
CVE-2016-10150	UAF	✔	✗
CVE-2022-20409 ★	UAF	✔	✔
CVE-2022-2585 †	UAF	✗	✔

Mobile Device CVE:

CVE-2022-20409

Cross Cache Included CVE:

CVE-2022-20409

Refurbish Intractable Exploit:

CVE-2022-2585 in Case Study Section

8 Case Study: Refurbish Intractable Exploit

In this section, we demonstrate how Page Spray make improvements to certain hard-to-exploit vulnerability case, and enhance the exploitability. To achieve this, Page Spray employs two novel approaches, kernel information leakage by new channel, and halting the CPU execution to improve exploitability. We successfully apply Page Spray into a real world zero-day bug (CVE-2022-2585 [8]) and achieve privilege escalation.

Stability

Type	CVE	Slab-Cache	Single-Thread Spray	Multi-Process Spray	Page Spray	Subtypes
			In IDLE State			
UAF	CVE-2016-4557 †	Kmalloc-256	100%	100%	100%	eBPF
UAF	CVE-2016-8655 †	Kmalloc-2048	99.4%	99.3%	100%	Race
UAF	CVE-2017-10661 †	Kmalloc-256	41.4%	64.1%	99.8%	Race
UAF	CVE-2017-11176 †	Kmalloc-2048	99.4%	99.8%	99.7%	Normal
UAF	CVE-2017-15649 †	Kmalloc-4096	61.4%	99.4%	97.9%	Race
UAF	CVE-2018-6555	Kmalloc-96	98.9%	100%	87.7%	Normal
OOB	CVE-2016-0728 †	Kmalloc-256	91.3%	99.8%	99.3%	Race
OOB	CVE-2021-22555	Kmalloc-1024	77.3%	46.0%	61.2%	Normal
DF	CVE-2017-8890 †	Kmalloc-64	74.3%	94.6%	94.4%	Normal
DF	CVE-2022-2588 †	Kmalloc-256	87.3%	10.6%	91.4%	Normal
			In BUSY State (stress-ng)			
UAF	CVE-2016-4557	Kmalloc-256	75.6%	97.4%	84.4%	eBPF
UAF	CVE-2016-8655 †	Kmalloc-2048	64.3%	58.1%	61.5%	Race
UAF	CVE-2017-10661 †	Kmalloc-256	28.6%	78.3%	98.1%	Race
UAF	CVE-2017-11176	Kmalloc-2048	79.8%	94.4%	63.7%	Normal
UAF	CVE-2017-15649 †	Kmalloc-4096	38.1%	98.8%	99.2%	Race
UAF	CVE-2018-6555 †	Kmalloc-96	92.0%	98.1%	90.7%	Normal
OOB	CVE-2016-0728	Kmalloc-256	40.4%	99.9%	87.3%	Race
OOB	CVE-2021-22555	Kmalloc-1024	71.8%	39.4%	43.4%	Normal
DF	CVE-2017-8890 †	Kmalloc-64	18.7%	27.8%	49.0%	Normal
DF	CVE-2022-2588 †	Kmalloc-256	50.9%	19.0%	54.0%	Normal

Two system workloads:
IDLE and BUSY

Different Vulnerability Types:
OOB/UAF/DF…

https://github.com/haruki3hhh/PageSpray/tree/main/stability_exploitability

Mitigation Discussion

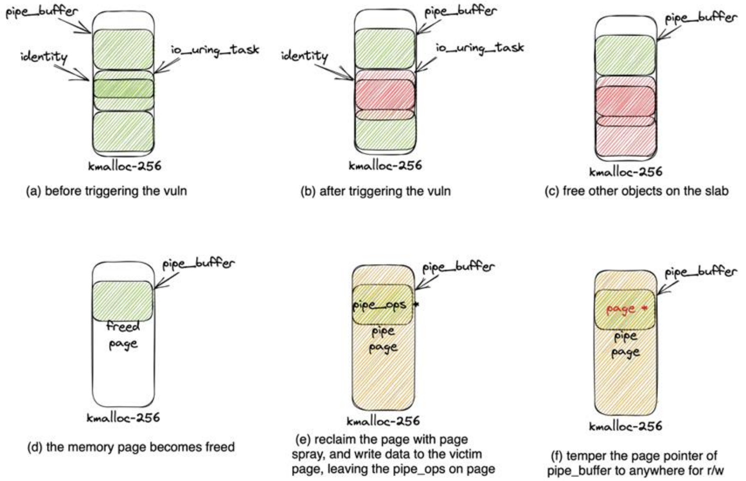
Page-level Memory Reuse is dangerous!

- A straightforward idea to mitigate:
  - isolate the pages to another memory area, by GFP_<FLAG>
  - Even page-spray can be triggered, overlap between critical objects and data won't happen.
- An external mitigation:
  - **SLAB_VIRTUAL**
  - https://patchwork.kernel.org/project/linux-mm/patch/20230915105933.495735-15-matteorizzo@google.com/#25513020
  - Prevent slab virtual address reuse!

Realworld

- Some **Realworld Exploits**, our team use Page Spray
- **CVE-2022-20409 in Google Pixel 6 and Samsung S22**
  - **Blackhat USA 2023, “Bad io_uring”**

- **CVE-2022-2585 in Google kCTF, TyphoonPWN**



(a) before triggering the vuln
(b) after triggering the vuln
(c) free other objects on the slab
(d) the memory page becomes freed
(e) reclaim the page with page spray, and write data to the victim page, leaving the pipe_ops on page
(f) temper the page pointer of pipe_buffer to anywhere for r/w

8 Case Study: Refurbish Intractable Exploit

In this section, we demonstrate how Page Spray make improvements to certain hard-to-exploit vulnerability case, and enhance the exploitability. To achieve this, Page Spray employs two novel approaches, kernel information leakage by new channel, and halting the CPU execution to improve exploitability. We successfully apply Page Spray into a real world zero-day bug (CVE-2022-2585 [8]) and achieve privilege escalation.

Conclusion

- Page Spray provides comparable even superior exploitability and stability in real-world scenarios.
- Root cause of Page Spray is associated with some mechanisms in the Linux Kernel’s design.
- Rethink the reuse of pages! Design and introduce more powerful mitigation into kernel to mitigate page spray attack.