black hat USA 2024

AUGUST 7-8, 2024

BRIEFINGS

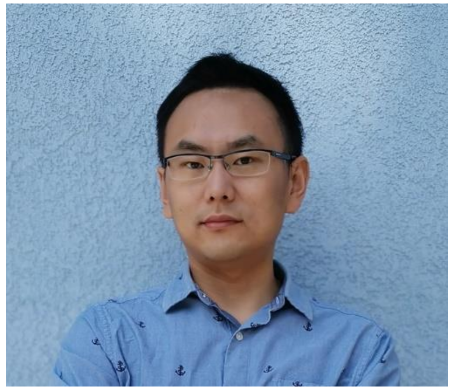
PageJack: A Powerful Exploit Technique With Page-Level UAF

Speaker: Zhiyun Qian

Contributors: Jiayi Hu, Jinmeng Zhou, Qi Tang, Wenbo Shen

8/8/2024

Who we are



Zhiyun Qian

Jiayi Hu

Jinmeng Zhou

Qi Tang

Wenbo Shen

OS kernel exploits

Control flow hijack

Ex: corrupt function pointer → return-oriented programming (ROP)

corrupted_obj->func_ptr() → Arbitrary code location

Data-only attacks

Ex: corrupt data pointer → arbitrary read/write to modify key objects (e.g., cred)

*corrupted_obj->data_ptr = val; → Arbitrary data location

Control-flow integrity

LWN .net
News from the source

Content
Weekly Edition
Archives
Search
Kernel
Security
Events calendar
Unread comments

LWN FAQ
Write for us

Edition
Return to the Front page

User: **Password:** Log in | Subscribe | Register

Control-flow integrity for the kernel

Did you know...?

LWN.net is a subscriber-supported publication; we rely on subscribers to keep the entire operation going. Please help out by buying a subscription and keeping LWN on the net.

By **Jake Edge**
January 22, 2020

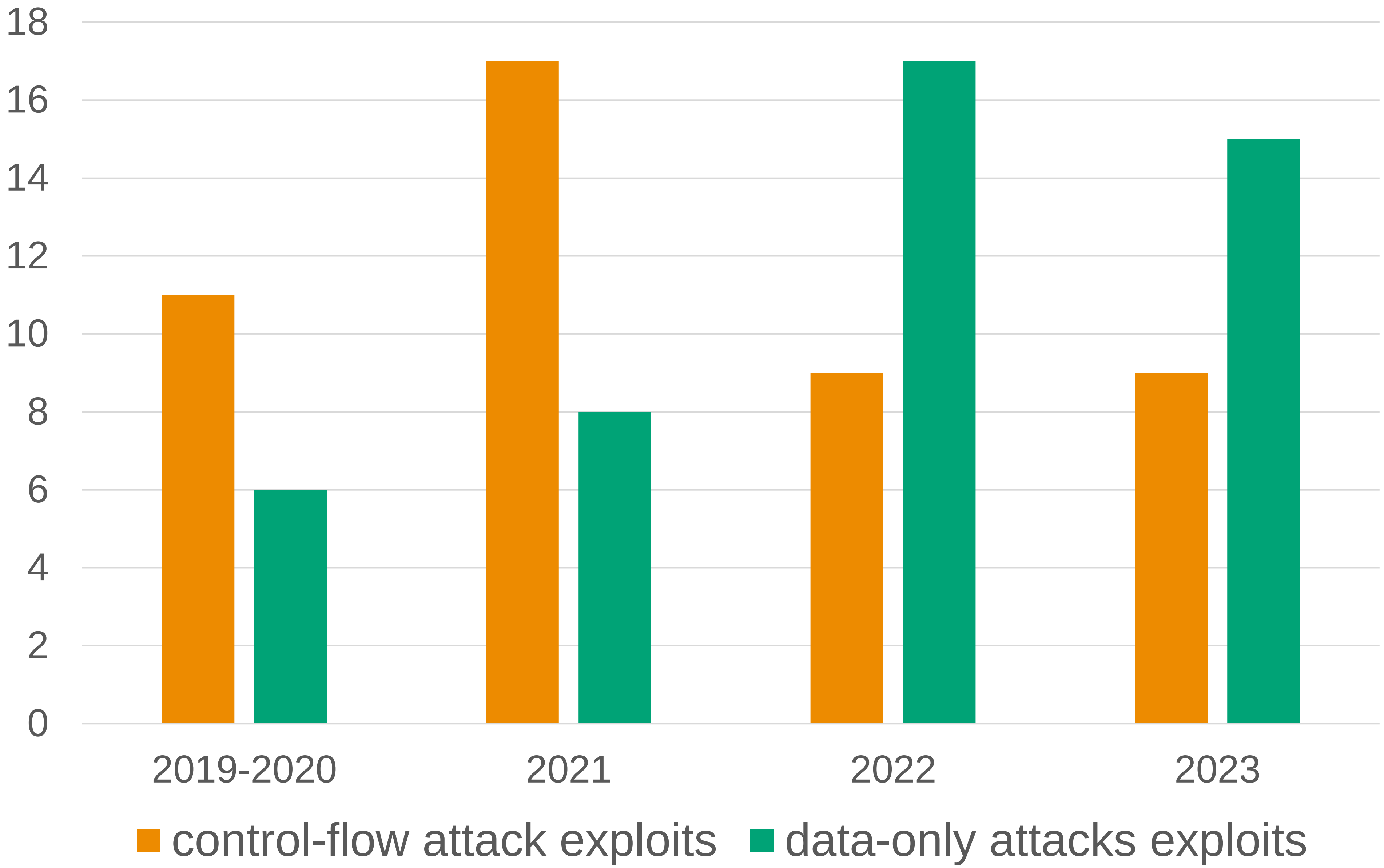
LCA

Control-flow integrity (CFI) is a technique used to reduce the ability to redirect the execution of a program's code in attacker-specified ways. The Clang compiler has some features that can assist in maintaining control-flow integrity, which have been applied to the Android kernel. Kees Cook gave a talk about CFI for the Linux kernel at the recently concluded linux.conf.au in Gold Coast, Australia.

Cook said that he thinks about CFI as a way to reduce the attack, or exploit, surface of the kernel. Most compromises of the kernel involve an attacker gaining execution control, typically using some kind of write flaw to change system memory. These write flaws come in many flavors, generally with some restrictions (e.g. can only write a single zero or only a set of fixed byte values), but in the worst case, they can be a "write anything anywhere at any time" flaw. The latter, thankfully, is relatively rare.

Data-only attack needed

Control-flow hijacking vs data-only attack



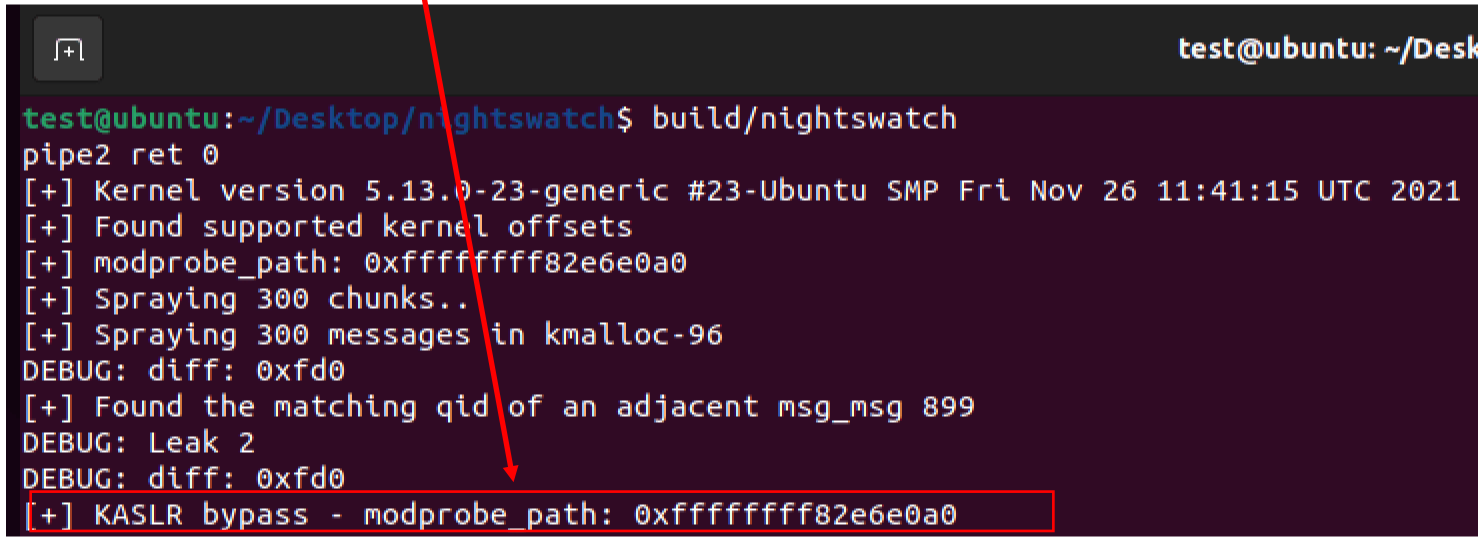
Data-only attacks

Previous data-only attacks

Corrupt
- global variable, e.g., **modprobe_path**
- heap variable, e.g., **cred**

Previous data-only attacks

Corrupt
- → global variable, e.g., **modprobe_path**
- → heap variable, e.g., **cred**

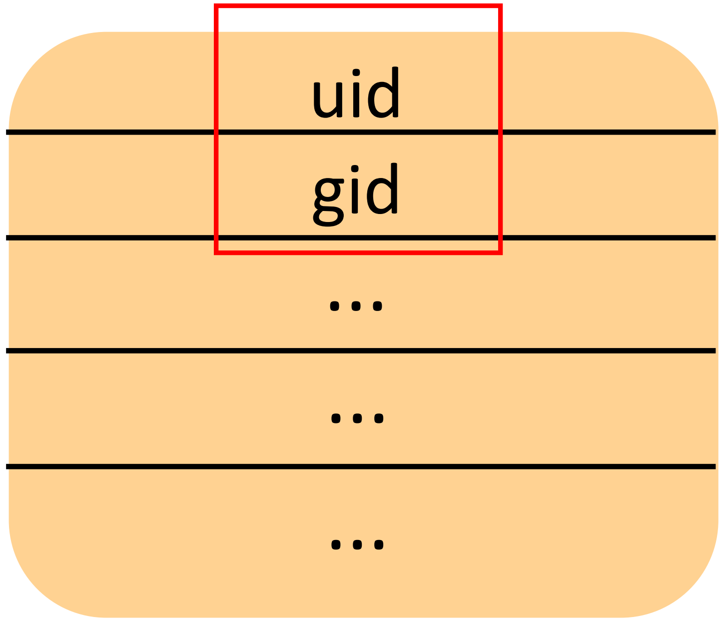


- KASLR bypass needed
- AAW capability needed
- **Protected by** CONFIG_STATIC_USERMODEHELPER

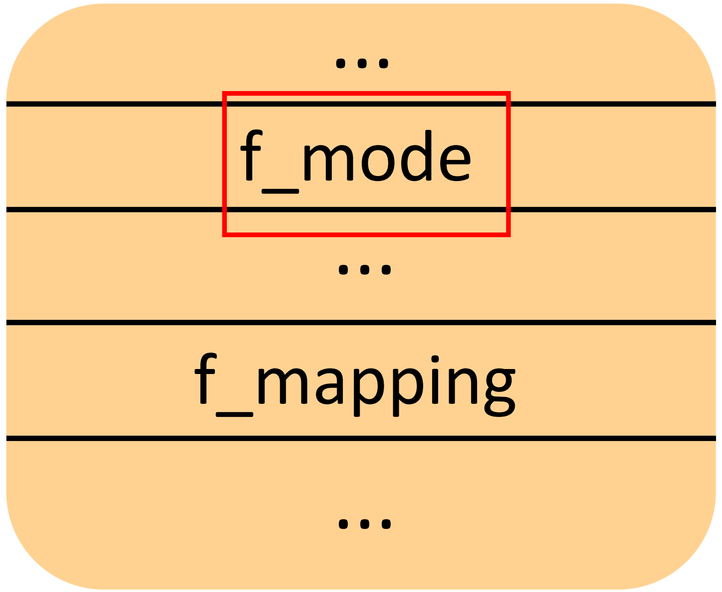
Previous data-only attack

Corrupt
- global variable, e.g., **modprobe_path**
- heap variable, e.g., **cred, file**

- Relative write (e.g., OOB) on **heap**
- AAW not needed



struct cred



struct file

Previous data-only attack: cross-cache challenge

- Most vulnerabilities happen in **generic** caches.
  (UAF, Double Free, Out-of-bound write)
- Most critical heap objects are in **dedicated** caches.
- How to reach critical heap objects with relative writes?

↓

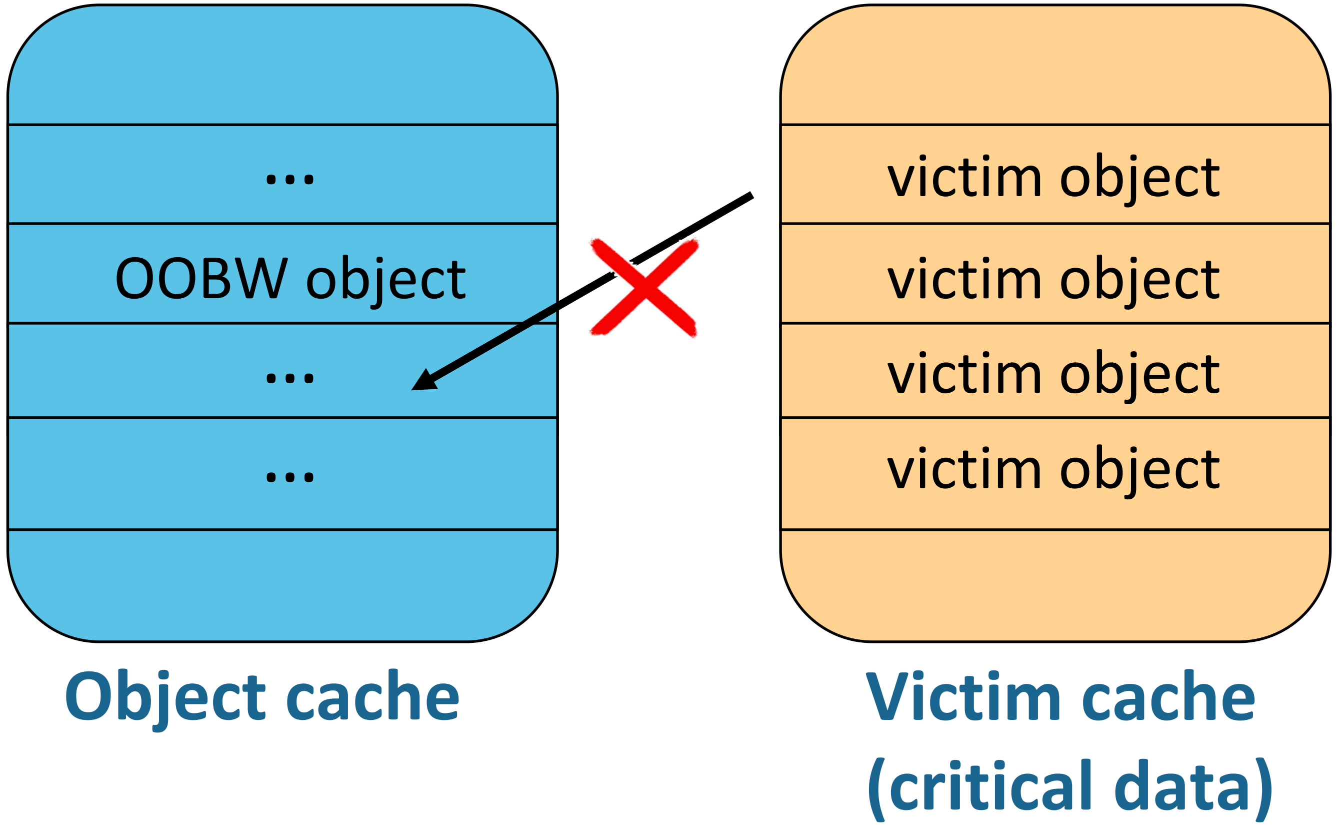
cross-cache attack needed

Previous data-only attack: cross-cache challenge

- Cross-cache attack techniques vary by vulnerability type, e.g.,
- OOB: less reliable
- UAF: more reliable but not future-proof
- Cross-cache still a significant hurdle for exploits

Previous data-only attack: cross-cache challenge

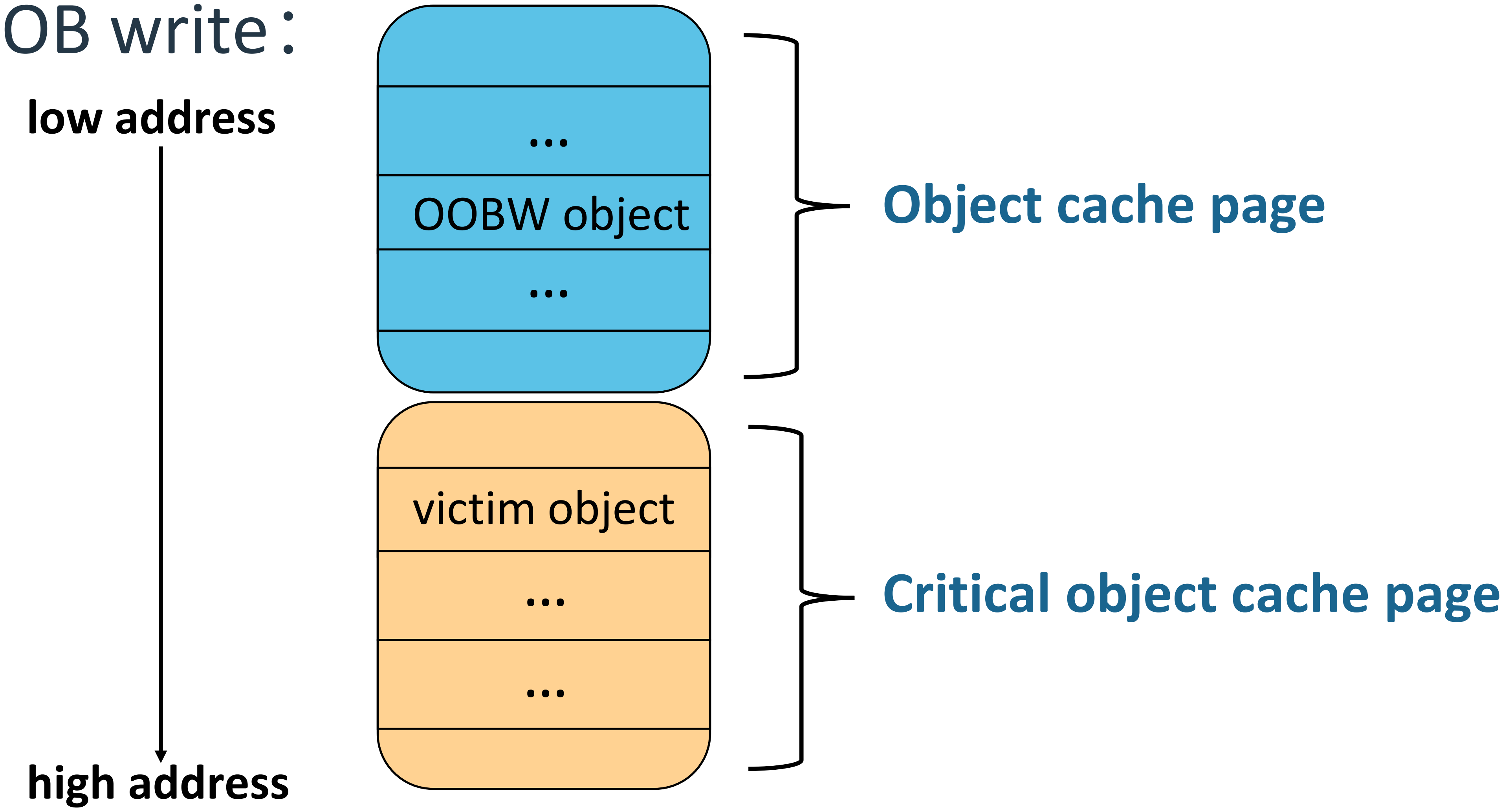
OOB write：



Any way to edit the victim critical heap objects directly with the OOB write capability?

Previous data-only attack: cross-cache challenge

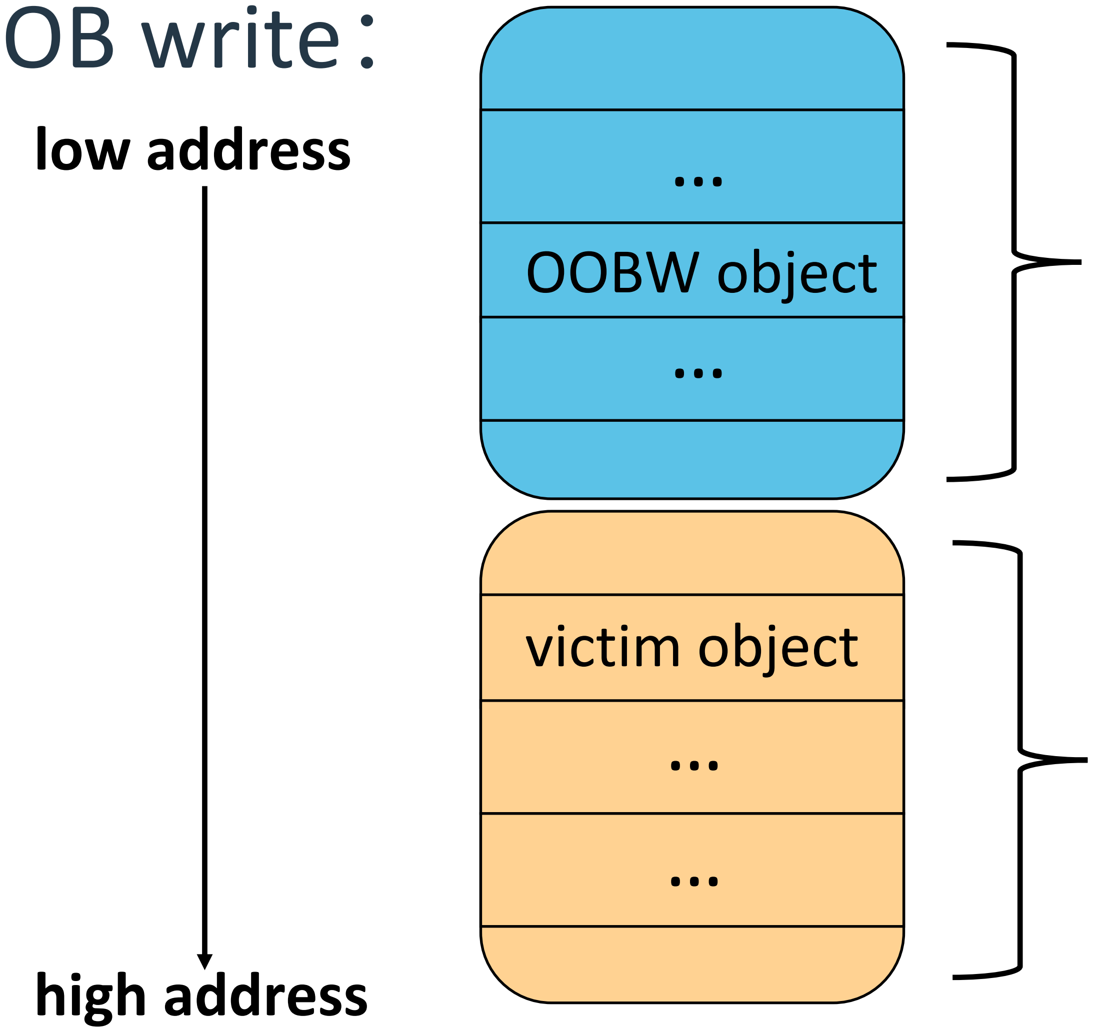
OOB write：



Page fengshui

Previous data-only attack: cross-cache challenge

OOB write：



Object cache page

CONFIG_SLAB_FREELIST_RANDOM=y?

Critical object cache page

Unreliable, low stability

Previous data-only attack: limited write capability

OOB write capability (few bytes etc..)

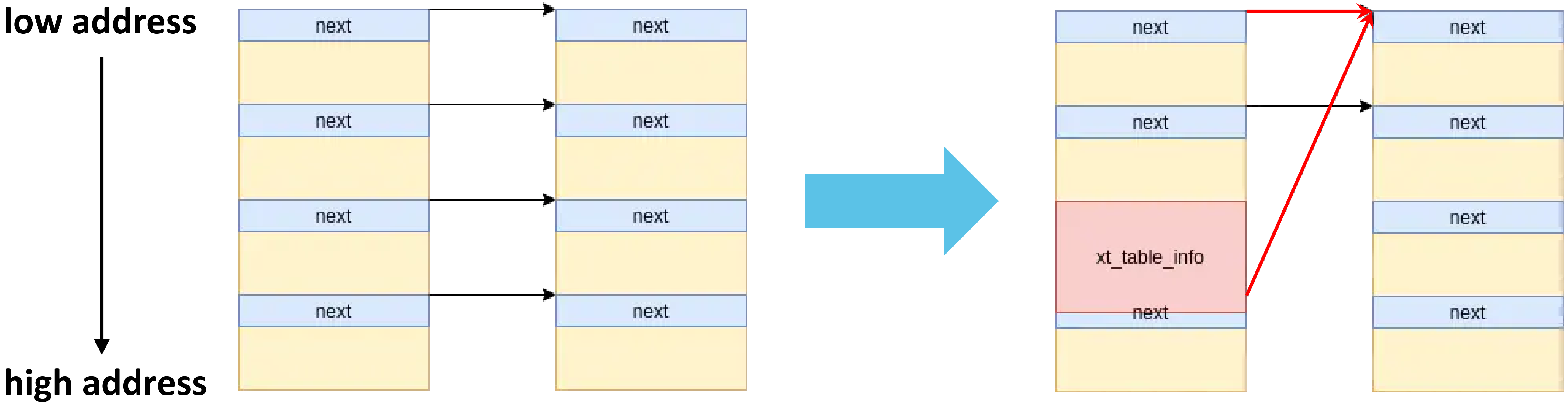
Pivoting to out-of-bound write to **double free / use after free:**

corrupt lower bits of heap data pointers

```
/* one msg_msg structure for each message */
struct msg_msg {
    struct list_head m_list;
    long m_type;
    size_t m_ts;         /* message text size */
    struct msg_msgseg *next;
    void *security;
    /* the actual message follows immediately */
};
```

Previous data-only attack: pivot OOB to UAF

Ex: CVE-2021-22555: corrupt lower byte(s) of msg_msg->mlist->next to force an object to be double referenced.



Previous msg_msg linked list

Corrupted msg_msg linked list

Figure credit: Andy Nguyen

Previous data-only attack: pivot OOB to UAF

Free the object once and create a dangling pointer → UAF

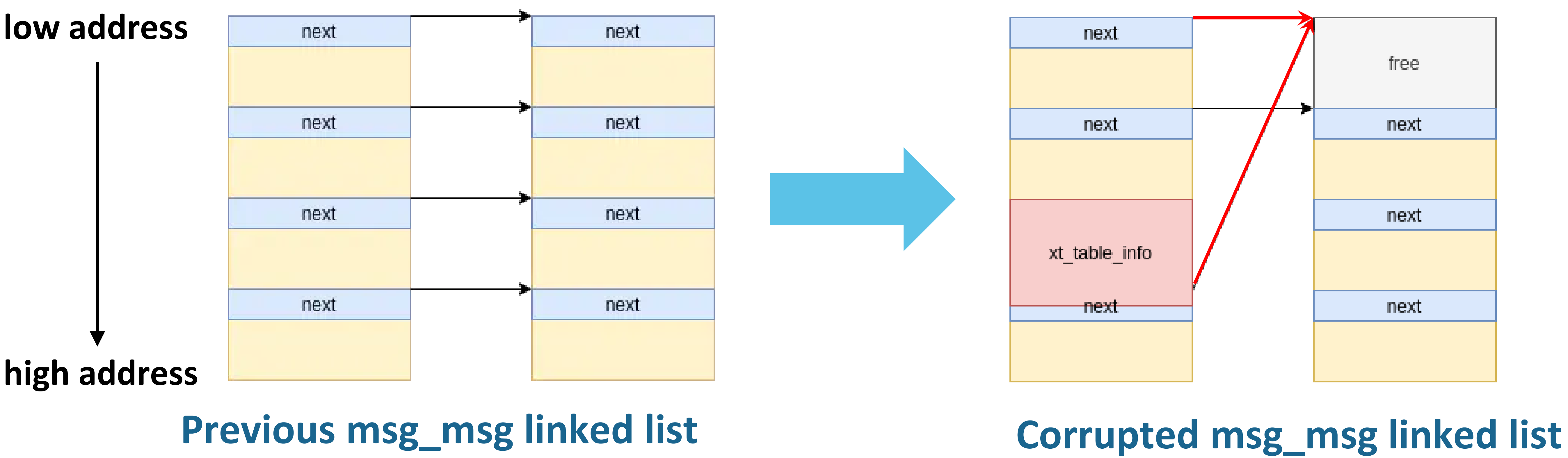


Figure credit: Andy Nguyen

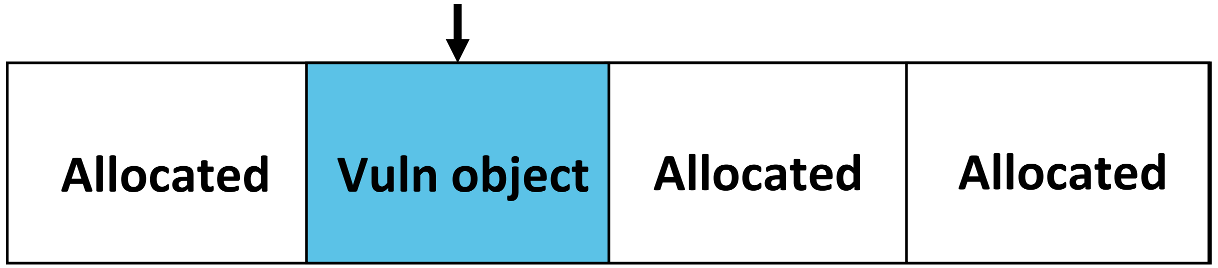
UAF to privilege escalation

Two challenges

- How to **overlap** the UAF object with the victim critical object?
- How to corrupt victim object without causing side effects?

UAF to privilege escalation

Challenge Ⅰ: Bypass cache isolation



Step1: Allocate many padding objects and vuln object in the same cache P.

occupying specific positions in the heap memory

which has UAF vulnerability

UAF to privilege escalation

Challenge Ⅰ: Bypass cache isolation

Allocated	Vuln object	Allocated	Allocated

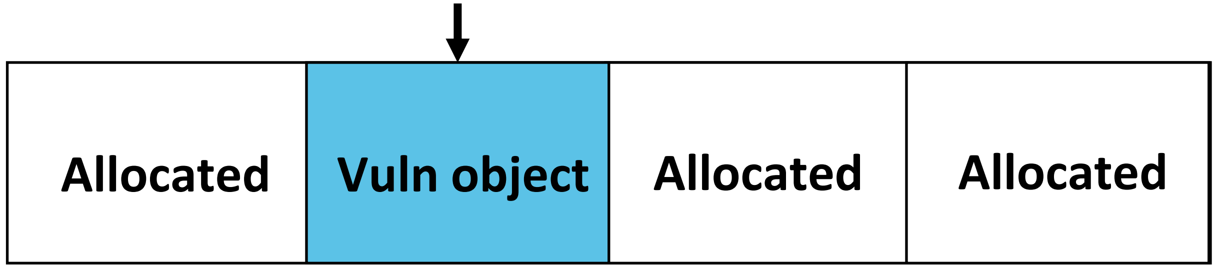
Step1: Allocate many padding objects and vuln object in the same cache P.

Allocated	Vuln object	Allocated	Allocated

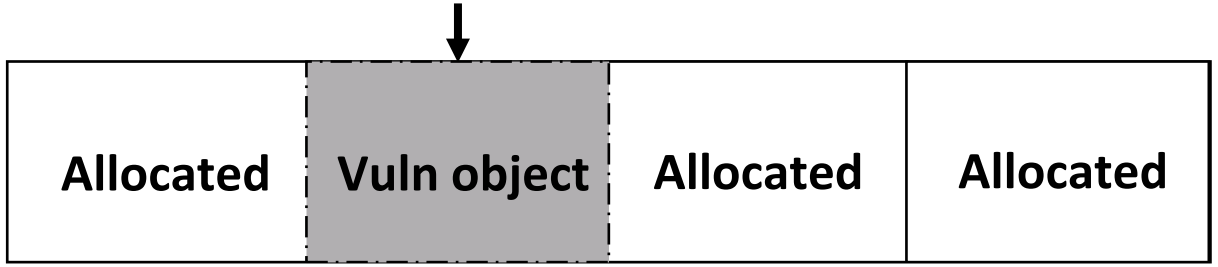
Step2: Free the vuln object in the cache P.

UAF to privilege escalation

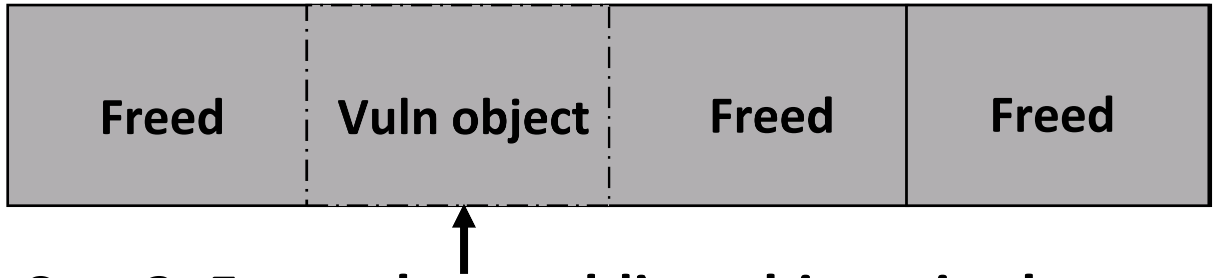
Challenge Ⅰ: Bypass cache isolation



Step1: Allocate many padding objects and vuln object in the same cache P.



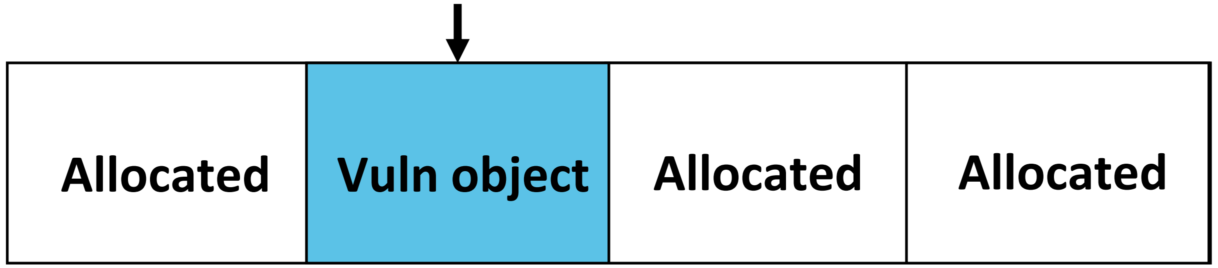
Step2: Free the vuln object in the cache P.



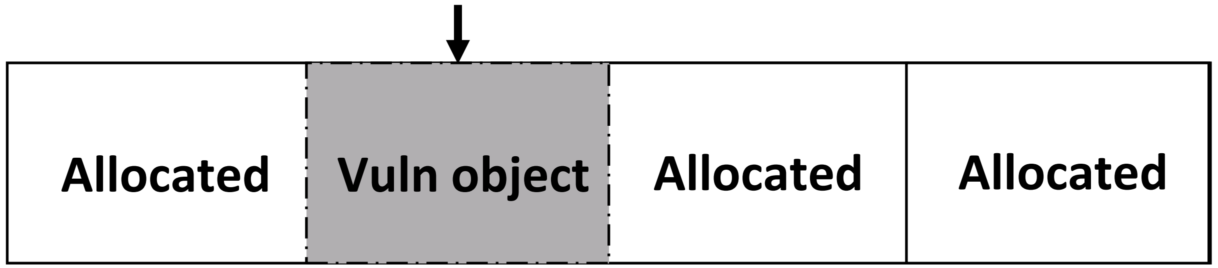
Step3: Free other padding objects in the cache P to recycle the page of cache P.

UAF to privilege escalation

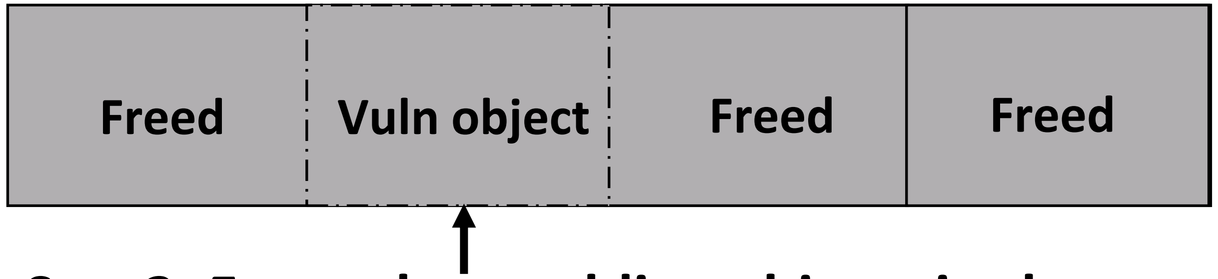
Challenge Ⅰ: Bypass cache isolation



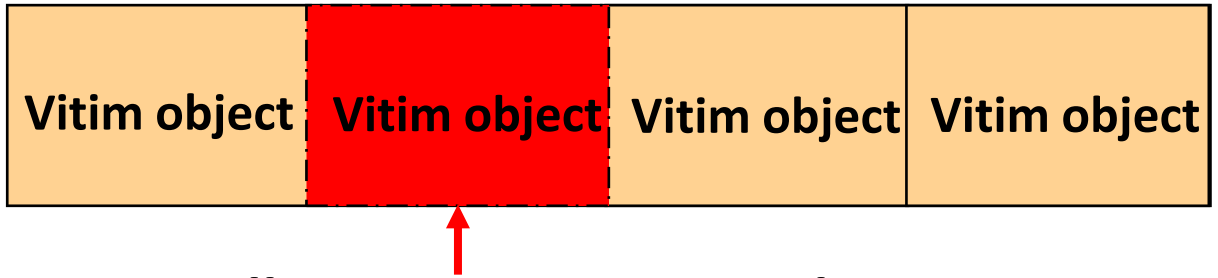
Step1: Allocate many padding objects and vuln object in the same cache P.



Step2: Free the vuln object in the cache P.



Step3: Free other padding objects in the cache P to recycle the page of cache P.

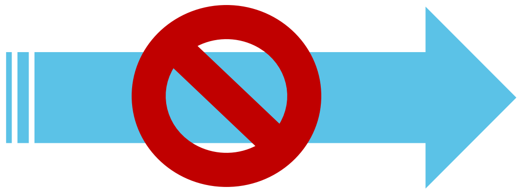


Step4: Allocate many victim objects to new a cache to reuse the freed page of cache P.

UAF to privilege escalation

Challenge Ⅰ: Bypass cache isolation

```
+config SLAB_VIRTUAL
+       bool "Allocate slab objects from virtual memory"
+       depends on SLUB && !SLUB_TINY
+       # If KFENCE support is desired, it could be implemented on top of our
+       # virtual memory allocation facilities
+       depends on !KFENCE
+       # ASAN support will require that shadow memory is allocated
+       # appropriately.
+       depends on !KASAN
+       help
+         Allocate slab objects from kernel-virtual memory, and ensure that
+         virtual memory used as a slab cache is never reused to store
+         objects from other slab caches or non-slab data.
+
```



Cross cache attack

Has killed

Google new mitigation: CONFIG_SLAB_VIRTUAL

UAF to privilege escalation

Two challenges

- How to **overlap** the UAF object with the victim critical object?
- How to corrupt victim object without causing side effects?

UAF to privilege escalation

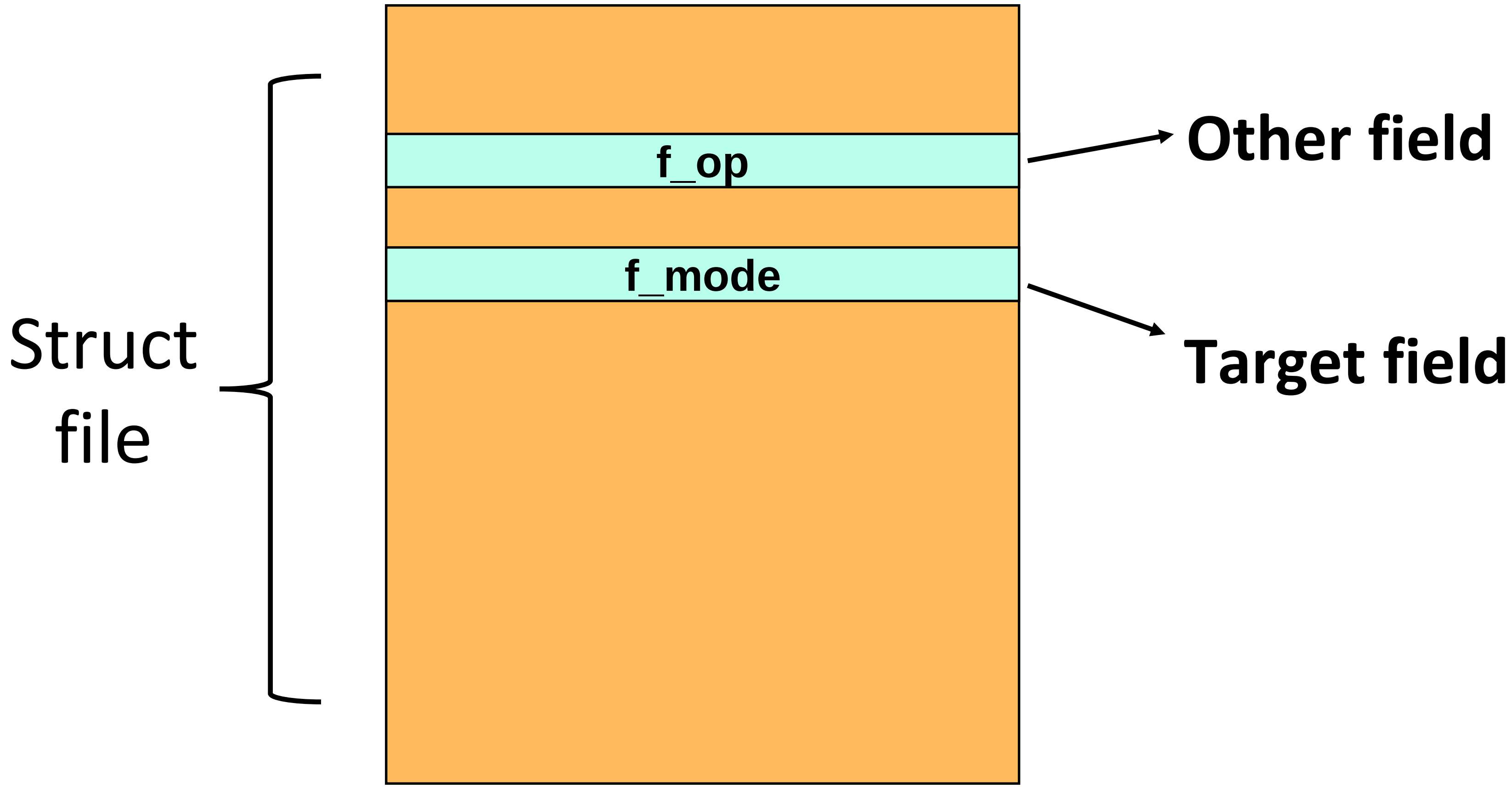
Challenge Ⅱ: avoid damaging other fields

```
struct file {
        union {
                struct llist_node  f_llist;              /*     0     8 */
                struct callback_head f_rcuhead __attribute__((__aligned__(8))); /*     0    16 */
                unsigned int       f_iocb_flags;         /*     0     4 */
        } __attribute__((__aligned__(8)));               /*     0    16 */
        struct path                f_path;               /*    16    16 */
        struct inode *             f_inode;              /*    32     8 */
        const struct file_operations  * f_op;            /*    40     8 */
        spinlock_t                 f_lock;               /*    48     4 */

        /* XXX 4 bytes hole, try to pack */

        atomic_long_t              f_count;              /*    56     8 */
        /* --- cacheline 1 boundary (64 bytes) --- */
        unsigned int               f_flags;              /*    64     4 */
        fmode_t                    f_mode;               /*    68     4 */
        struct mutex               f_pos_lock;           /*    72    32 */
        loff_t                     f_pos;                /*   104     8 */
        struct fown_struct         f_owner;              /*   112    32 */
        /* --- cacheline 2 boundary (128 bytes) was 16 bytes ago --- */
        const struct cred  *       f_cred;               /*   144     8 */
        struct file_ra_state       f_ra;                 /*   152    32 */
        u64                        f_version;            /*   184     8 */
        /* --- cacheline 3 boundary (192 bytes) --- */
        void *                     f_security;           /*   192     8 */
        void *                     private_data;         /*   200     8 */
        struct hlist_head *        f_ep;                 /*   208     8 */
        struct address_space *     f_mapping;            /*   216     8 */
        errseq_t                   f_wb_err;             /*   224     4 */
        errseq_t                   f_sb_err;             /*   228     4 */

        /* size: 232, cachelines: 4, members: 20 */
        /* sum members: 228, holes: 1, sum holes: 4 */
        /* forced alignments: 1 */
        /* last cacheline: 40 bytes */
} __attribute__((__aligned__(8)));
```

Target field = Offset + Field size

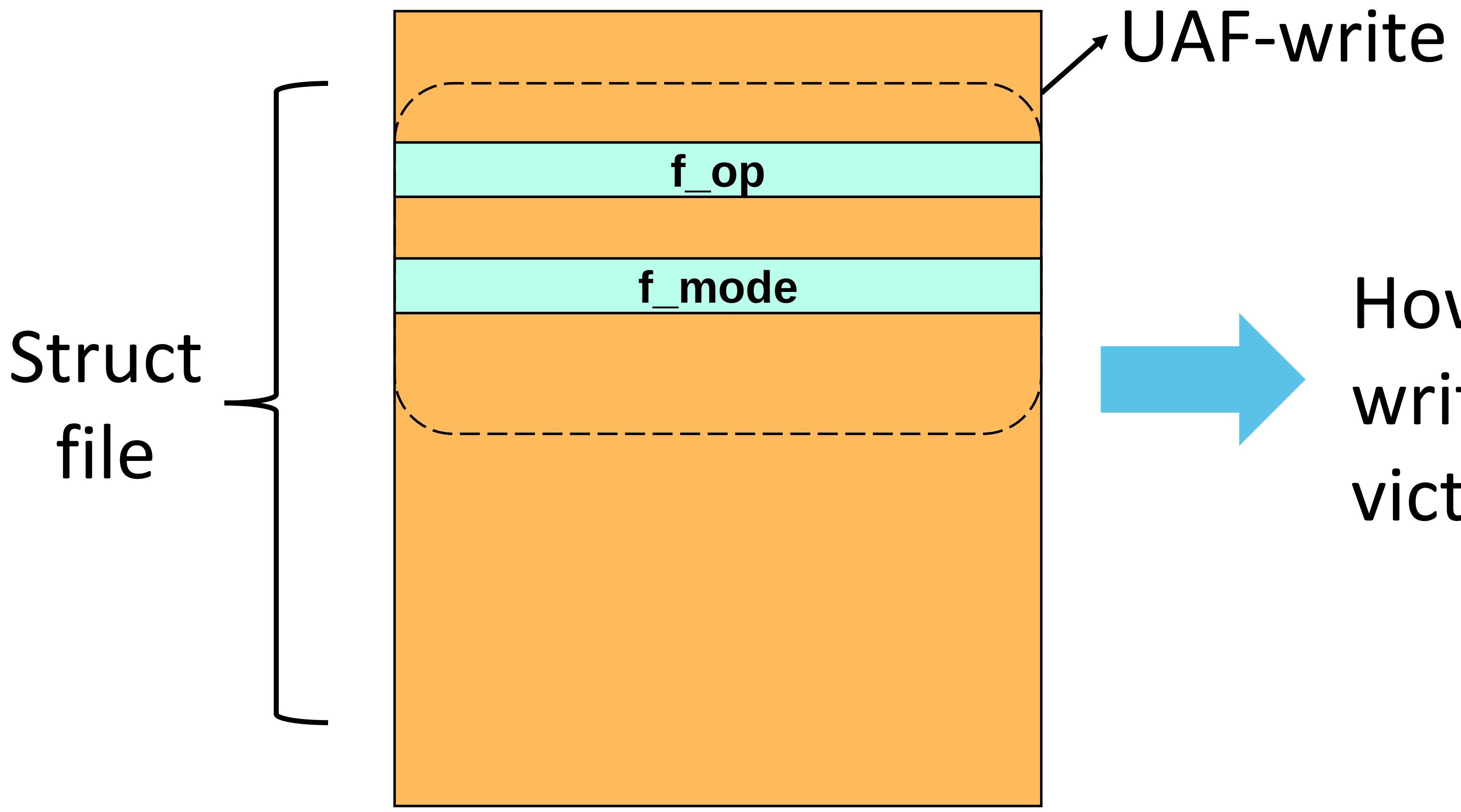
UAF to privilege escalation

Challenge Ⅱ: avoid damaging other fields



UAF to privilege escalation

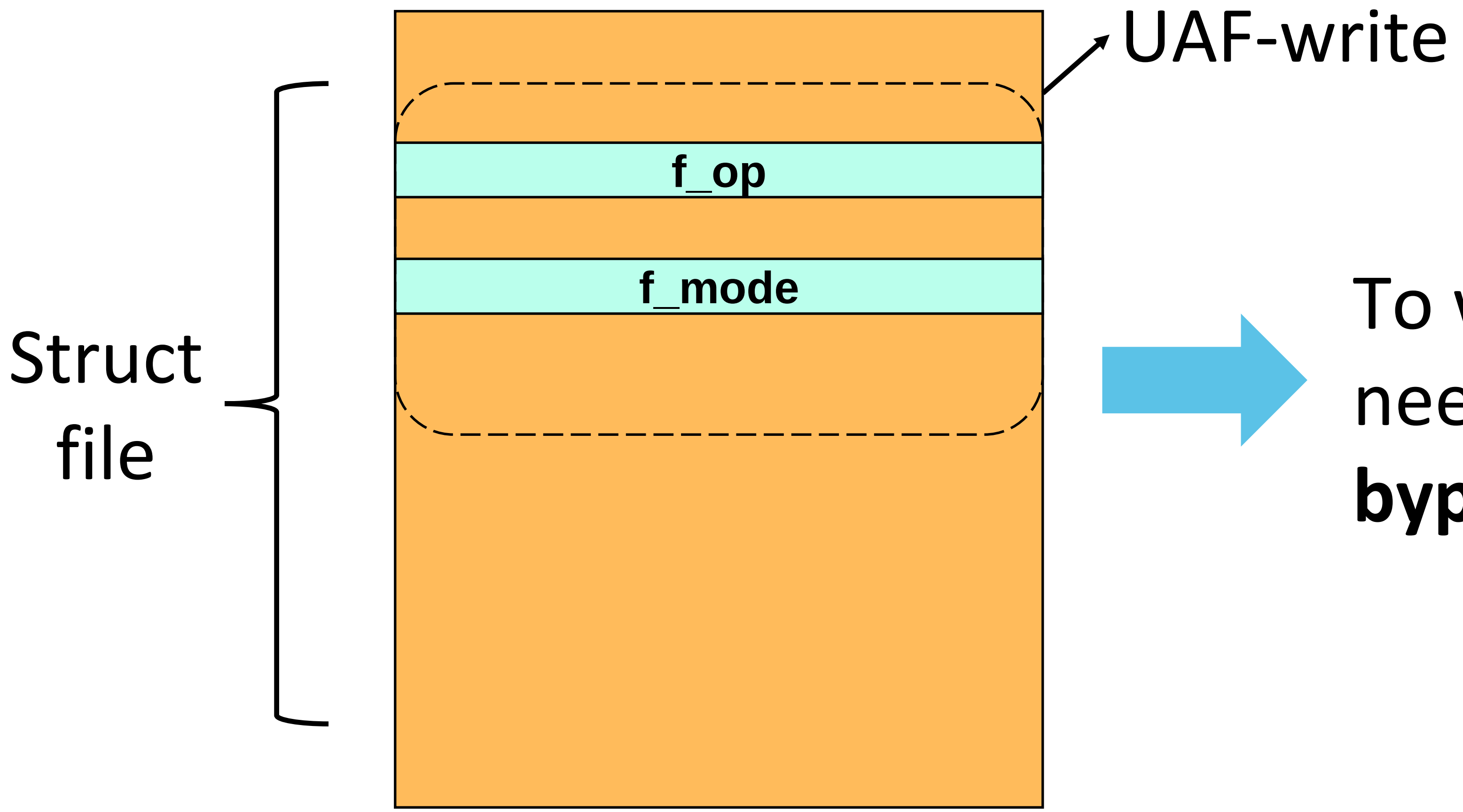
Challenge Ⅱ: avoid damaging other fields



However, the capability of UAF-write often doesn't match the victim field.

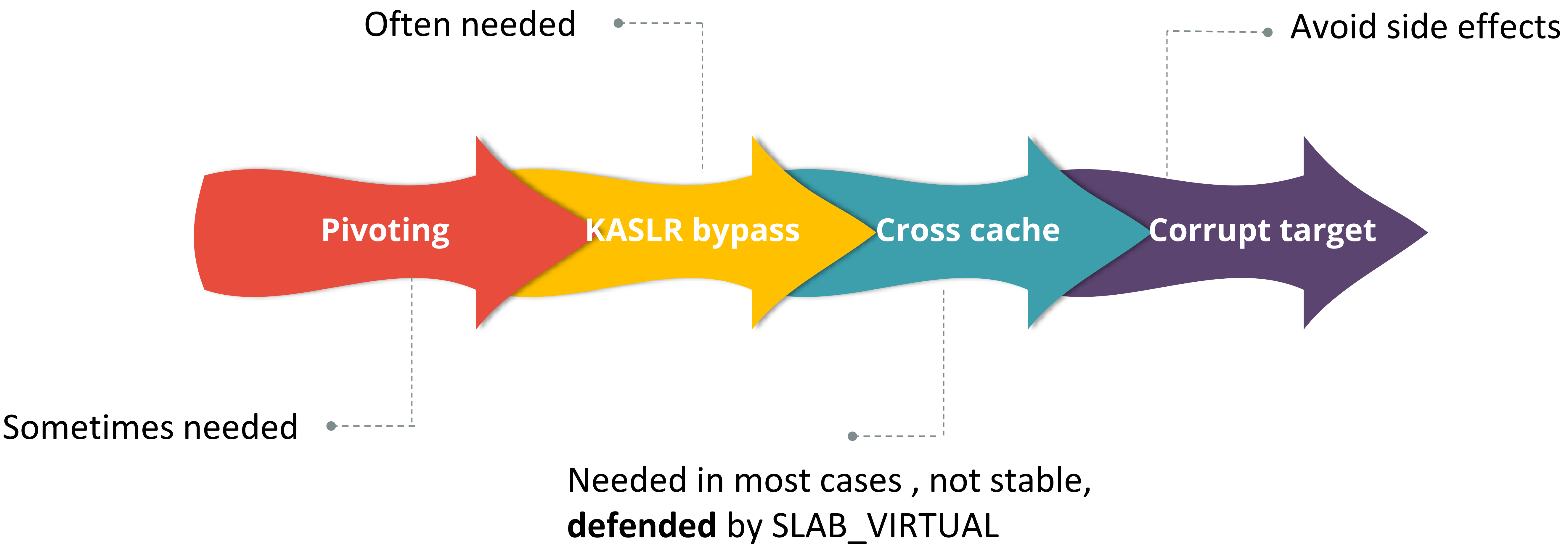
UAF to privilege escalation

Challenge Ⅱ: avoid damaging other fields



To write appropriate values, might need **additional steps** for **KASLR bypass** or other info-leak.

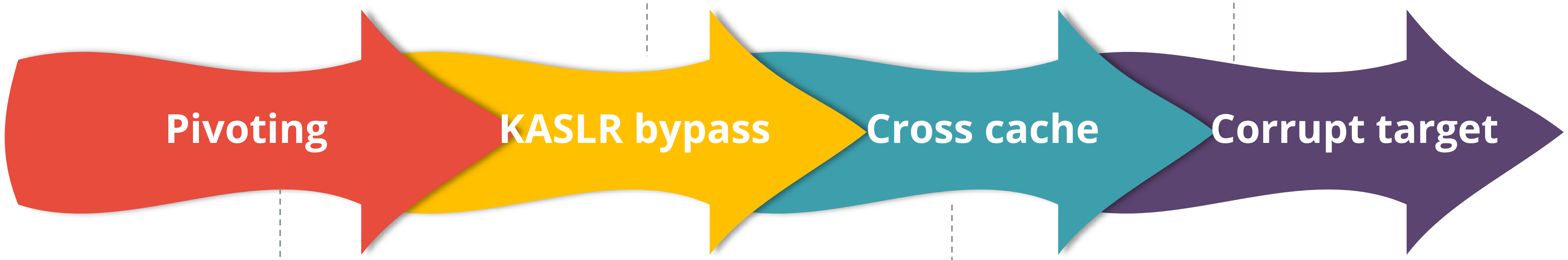
Review typical kernel exploit steps



Review typical kernel exploit steps

Often needed

Avoid side effects



Sometimes needed

Needed in most cases , not stable,
defended by SLAB_VIRTUAL

New ideas?

Page UAF to the rescue

ue 2504: Linux >=6.4: io_uring: page UAF via buffer ring mmap

Reported by jannh@google.com on Tue, Nov 28, 2023, 3:12 AM GMT+8 **Project Member**

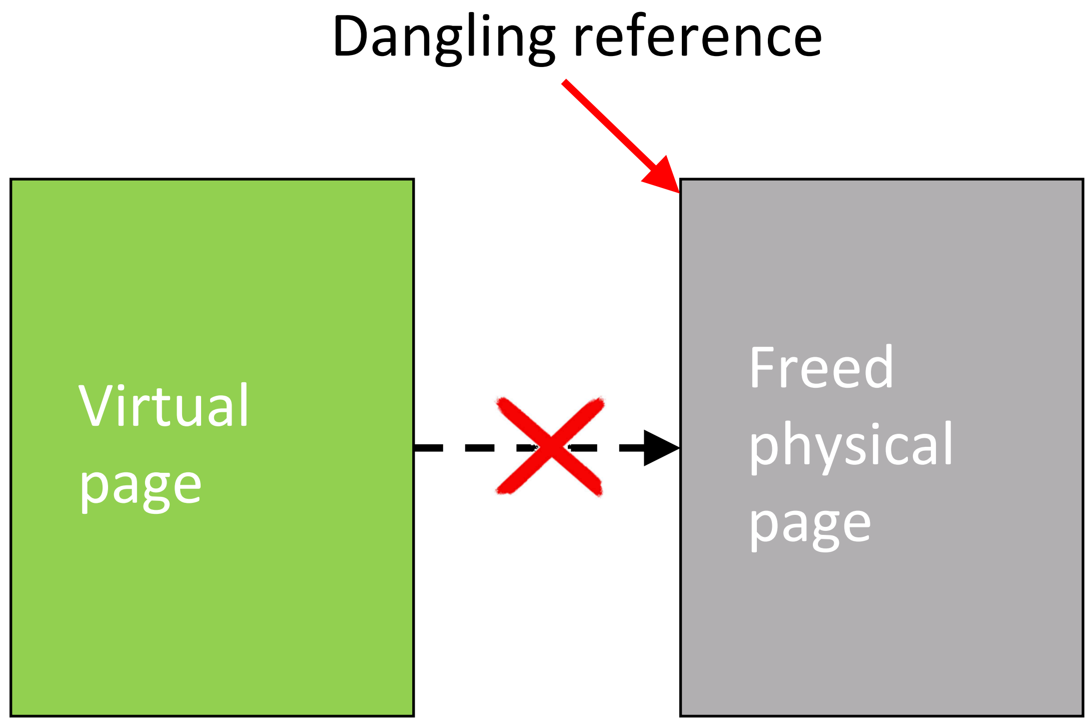
Since commit c56e022c0a27 ("io_uring: add support for user mapped provided buffer ring"), landed in Linux 6.4, io_uring makes it possible to allocate, mmap, and deallocate "buffer rings".

A "buffer ring" can be allocated with io_uring_register(..., IORING_REGISTER_PBUF_RING, ...) and later deallocated with io_uring_register(..., IORING_UNREGISTER_PBUF_RING, ...). It can be mapped into userspace using mmap() with offset IORING_OFF_PBUF_RING|..., which creates a VM_PFNMAP mapping, meaning the MM subsystem will treat the mapping as a set of opaque page frame numbers not associated with any corresponding pages; this implies that the calling code is responsible for ensuring that the mapped memory can not be freed before the userspace mapping is removed.

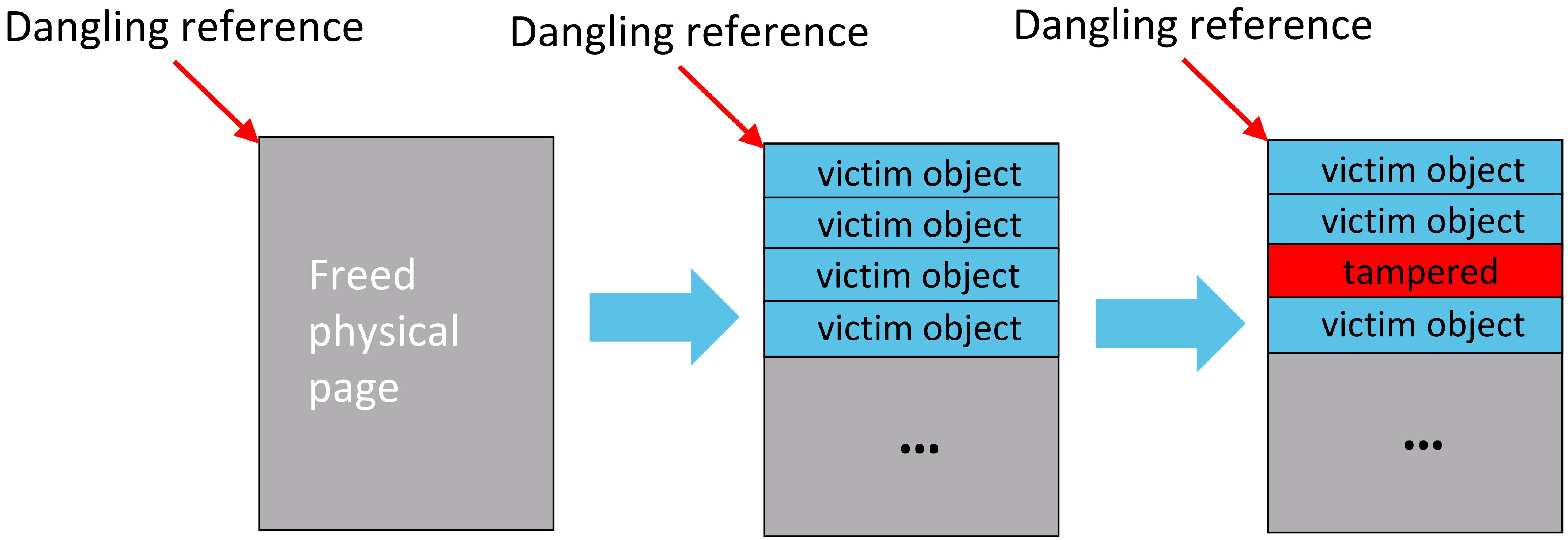
However, there is no mechanism to ensure this in io_uring: It is possible to just register a buffer ring with IORING_REGISTER_PBUF_RING, mmap() it, and then free the buffer ring's pages with IORING_UNREGISTER_PBUF_RING, leaving free pages mapped into userspace, which is a fairly easily exploitable situation.

Physical page freed, but still accessible

- Direct physical page read/write

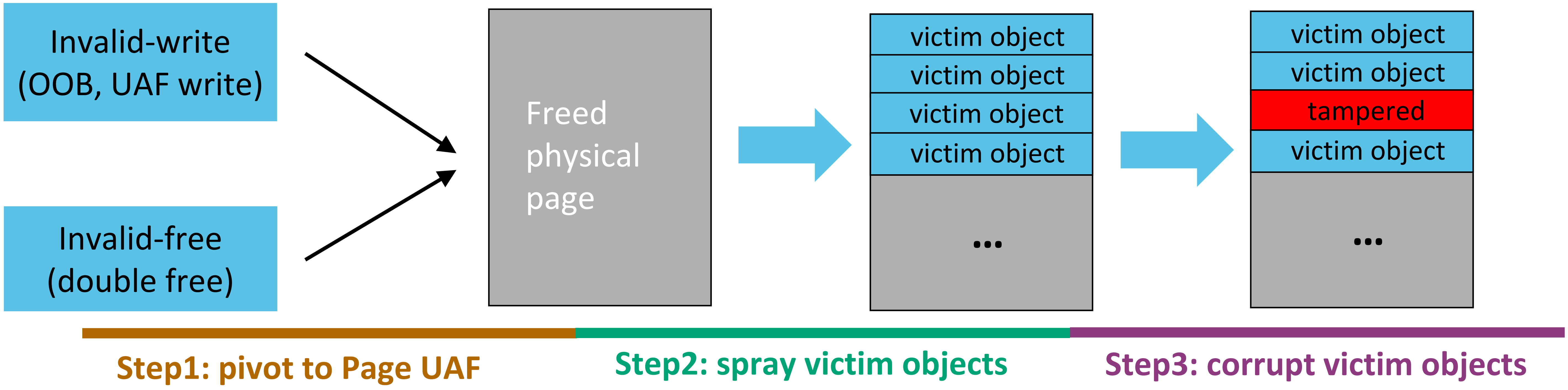


Page UAF to the rescue



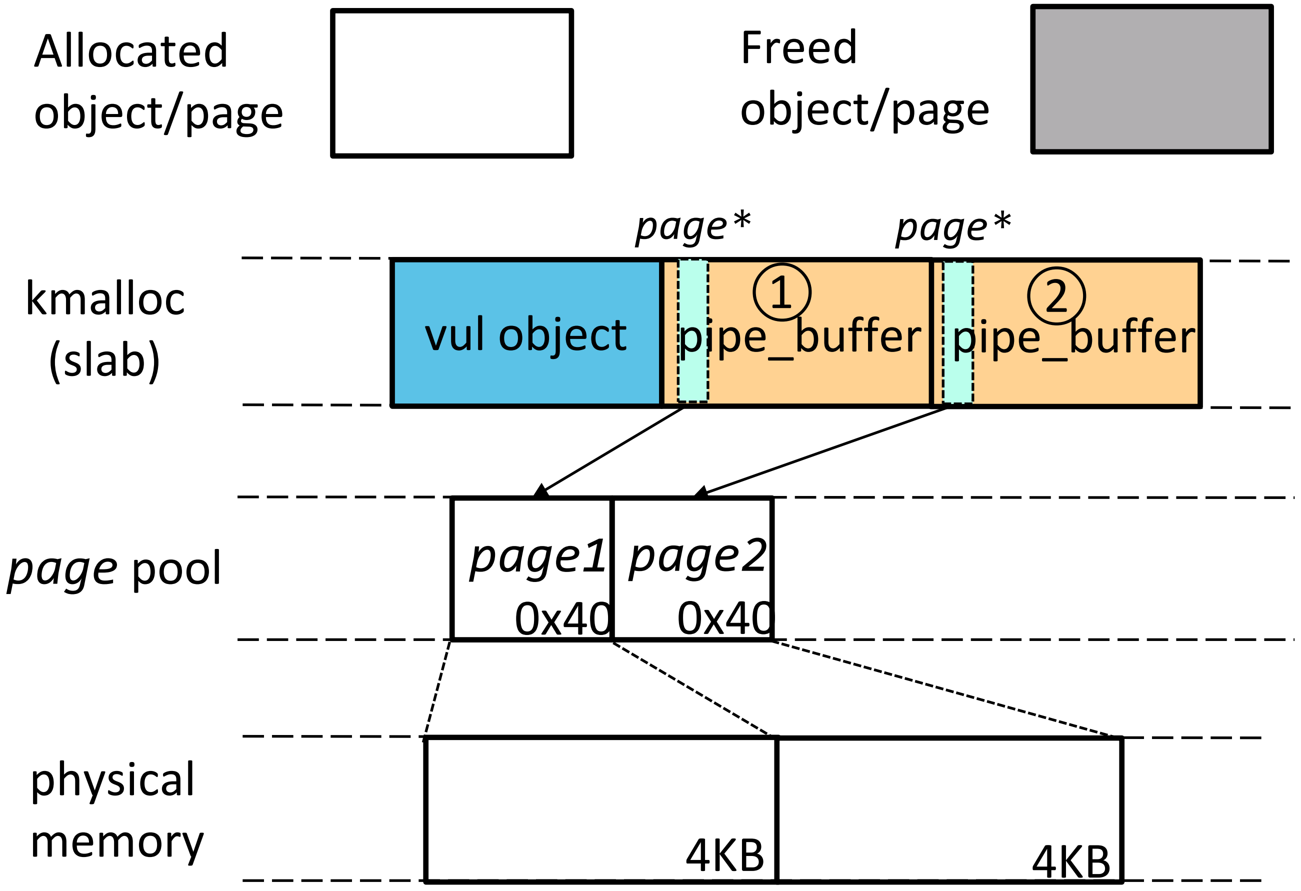
PageJack: a new exploit strategy

To derive Page UAF from different initial primitives



PageJack: pivoting relative writes to page UAF

Step 1 Memory layout manipulation (OOB example): arrange the vulnerable object to be adjacent to the objects containing the `struct page*`.

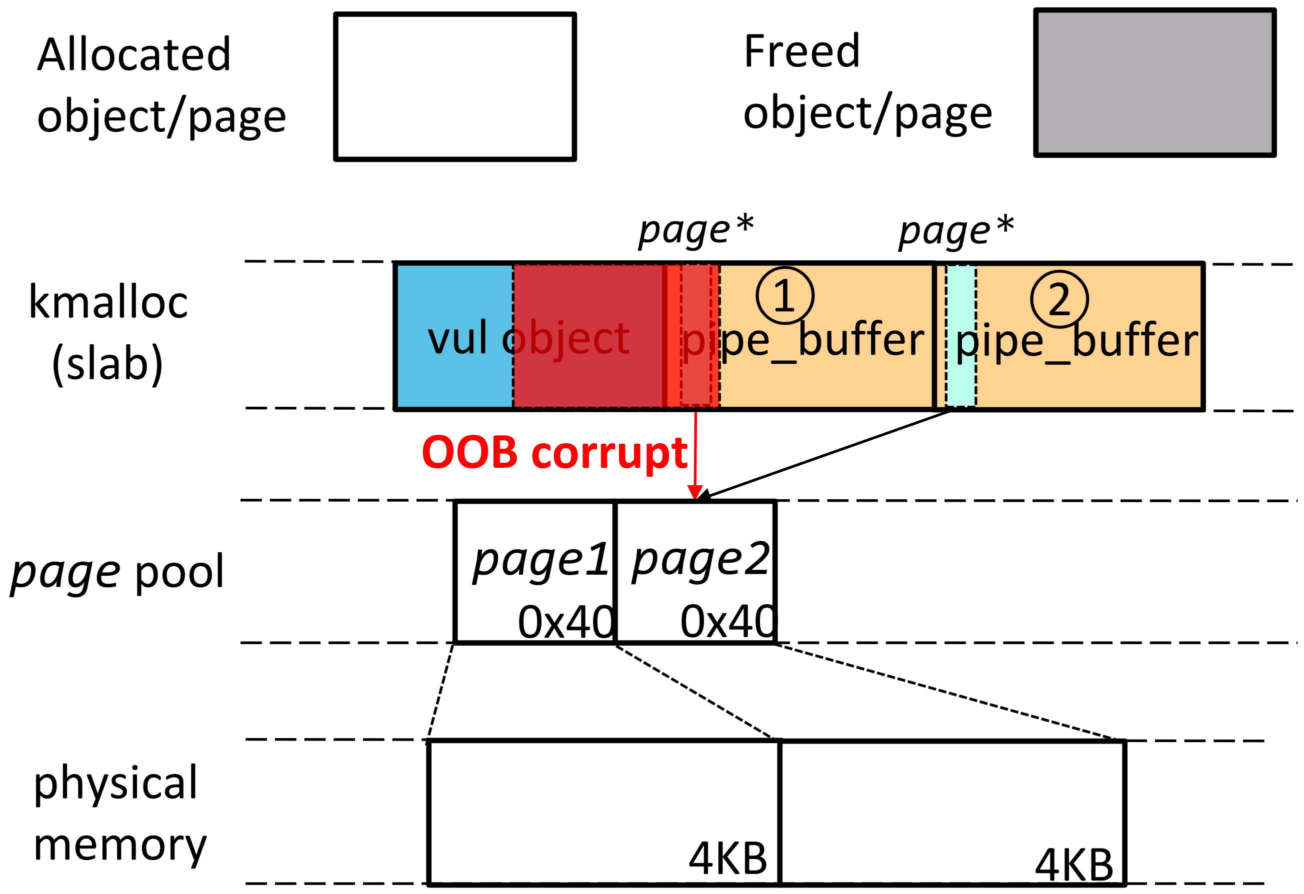



```c
struct page {
        unsigned long flags;

        /*
         * Five words (20/40 bytes) are
         * WARNING: bit 0 of the first w
         * means the other users of this
         * avoid collision and false-pos
         */
        union {
                struct {        /* Page cache and anonymous pages */
                        /**
                         * @lru: Pageout list, eg. active_list protected by
                         * lruvec->lru_lock.  Sometimes used as a generic list
                         * by the page owner.
                         */
                        struct list_head lru;
                        /* See page-flags.h for PAGE_MAPPING_FLAGS */
                        struct address_space *mapping;
                        pgoff_t index;          /* Our offset within mapping. */
                        /**
                         * @private: Mapping-private opaque data.
                         * Usually used for buffer_heads if PagePrivate.
                         * Used for swp_entry_t if PageSwapCache.
                         * Indicates order in the buddy system if PageBuddy.
                         */
                        unsigned long private;
                };
                struct {        /* page_pool used by netstack */
```

```c
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

PageJack: pivoting relative writes to page UAF

Step 2 Page pointer corruption: Trigger the OOB write to corrupt a *page** pointer to make it point to the nearby struct page object.

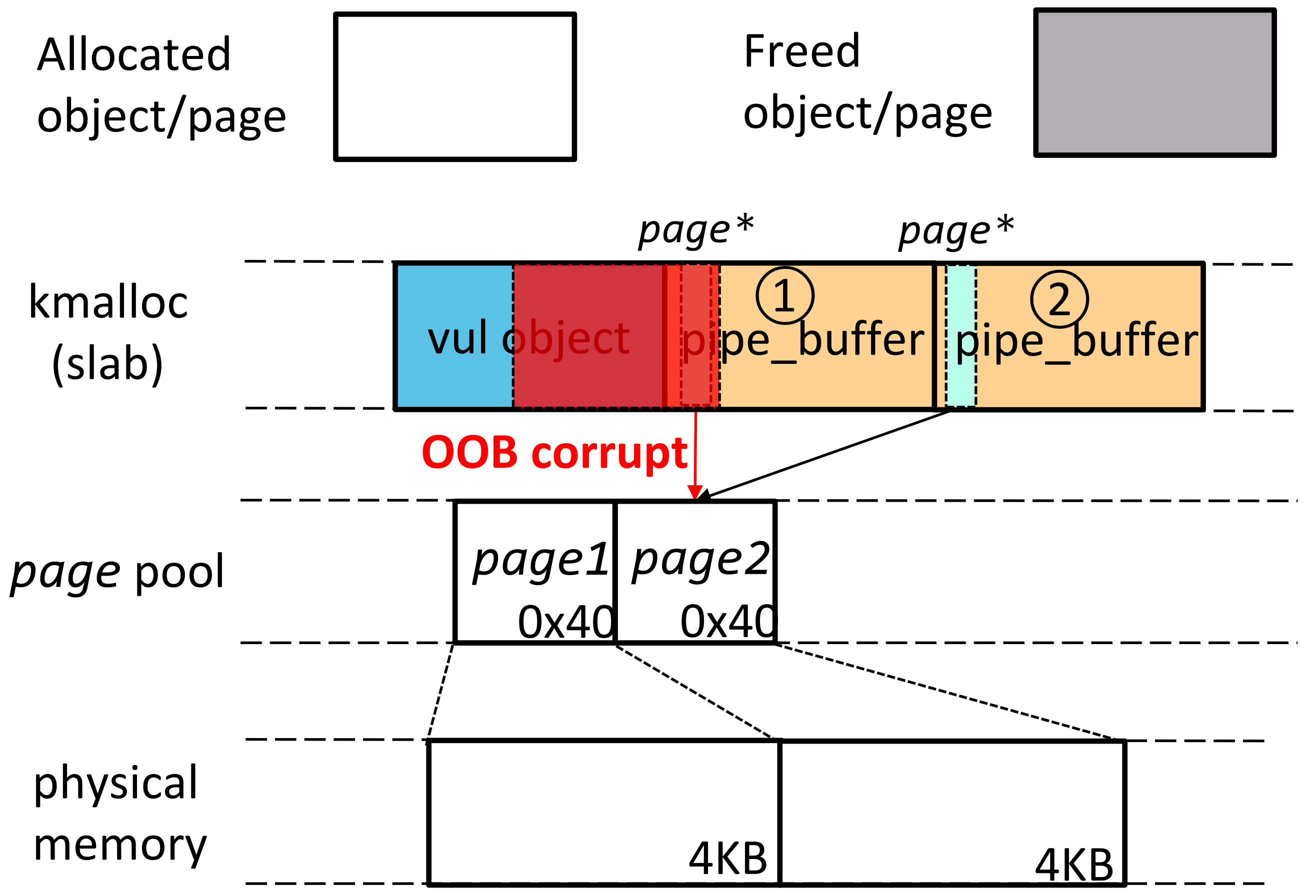


General to all kinds of bugs:

- Pivoting Invalid-Write (e.g., OOB & UAF write)
  We use OOB as an example.
- Pivoting Invalid-Free (e.g., Double-Free)
  we can use heap spray&&FUSE technique.

PageJack: pivoting relative writes to page UAF

Step 2 Page pointer corruption: Trigger the OOB write to corrupt a *page** pointer to make it point to the nearby struct page object.

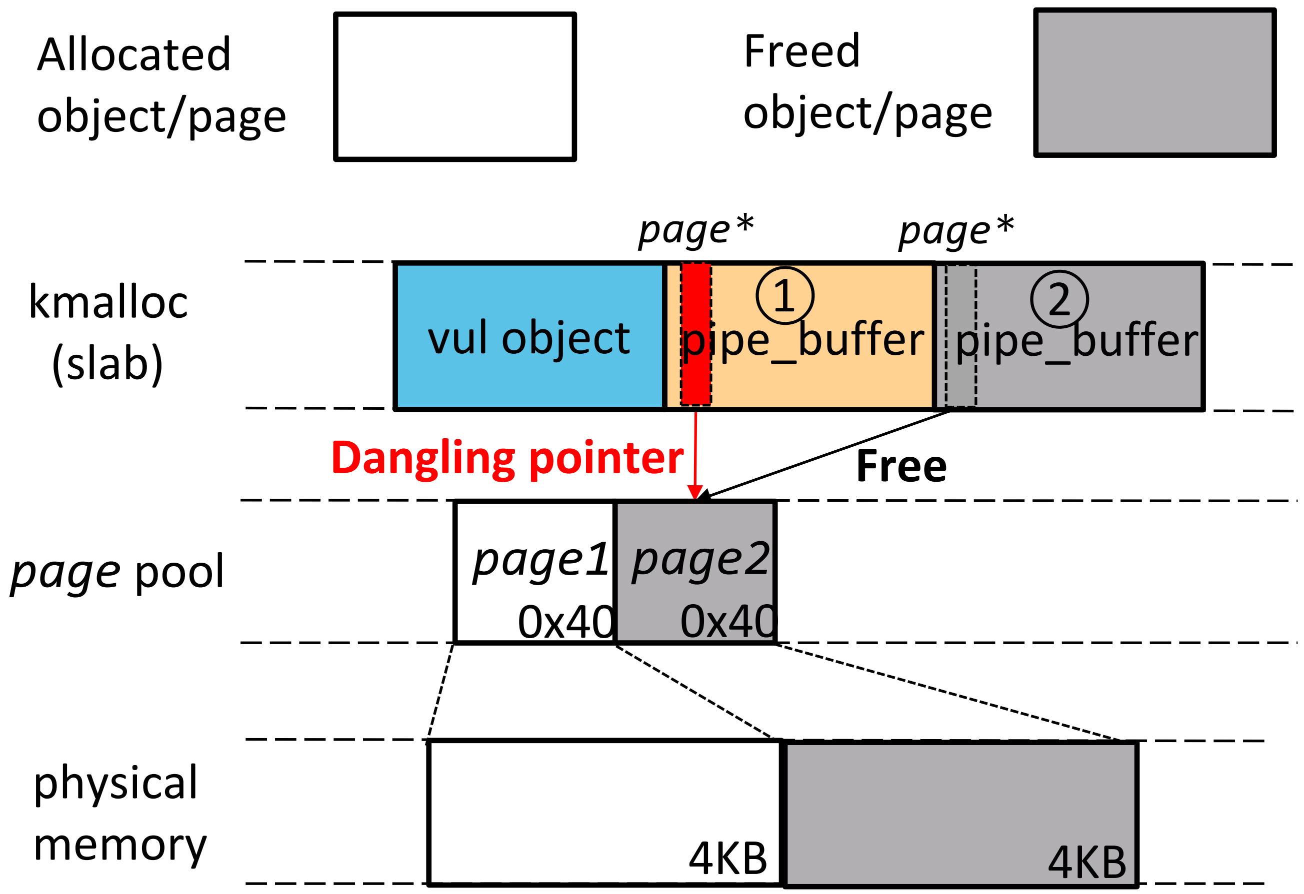


No need to bypass KASLR:

- sizeof (struct page) = 0x40
- change the last byte to to 0x00
  - succuss if the last byte is originally: 0x40, 0x80, 0xC0
  - fail but no harm if it is: 0x00

PageJack: pivoting relative writes to page UAF

Step 3 Page UAF construction: free the 4KB physical page, leaving a dangling pointer still points (reads and writes) to it.

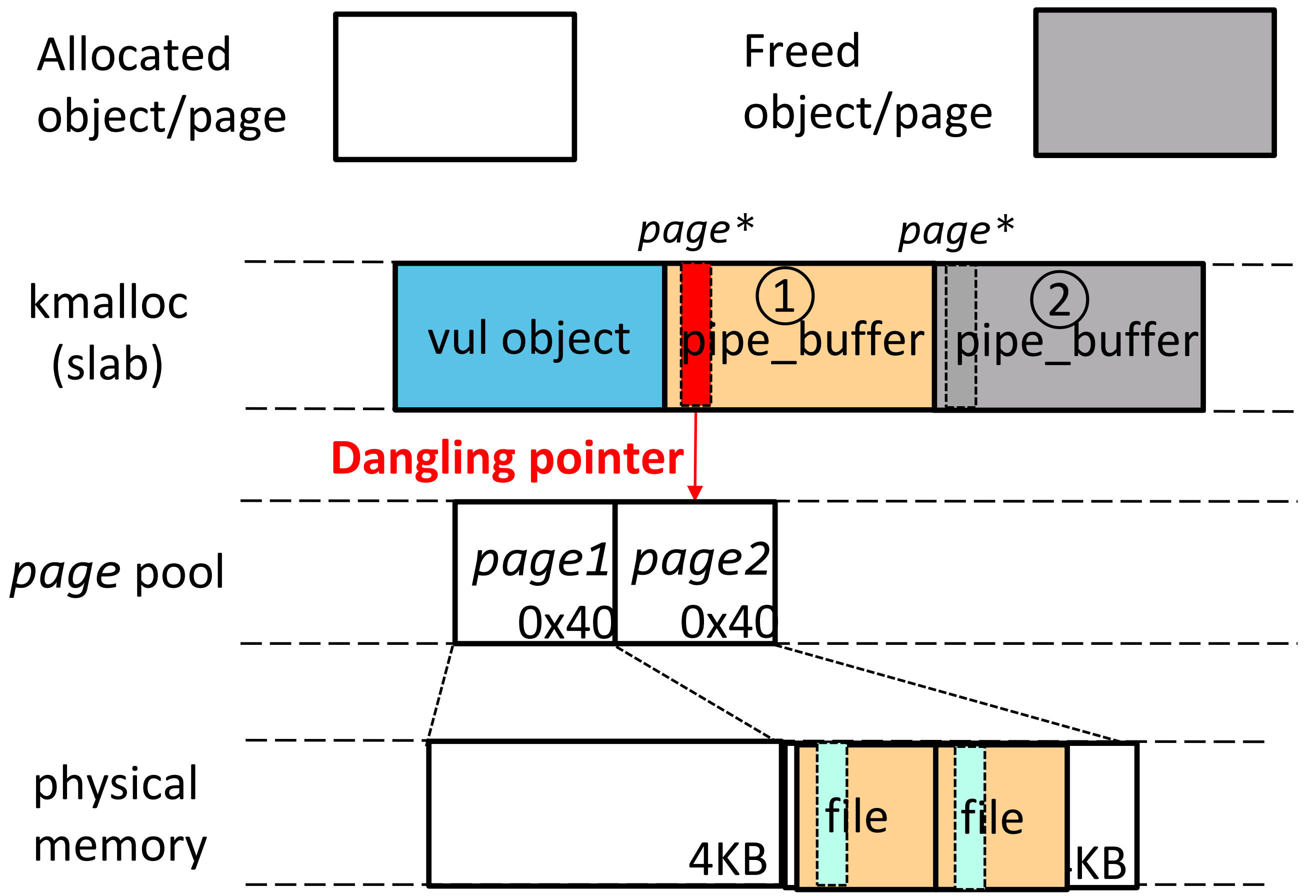


The freed page is reclaimed in buddy system:

- A 4KB physical page is managed by *a struct page* object.
- We trigger *a free_page()* to tell the buddy system the page can be reclaimed.

PageJack: tamper with critical objects

Step 4 Spray critical objects: allocate many critical objects (e.g., *file*) to reuse the freed page.

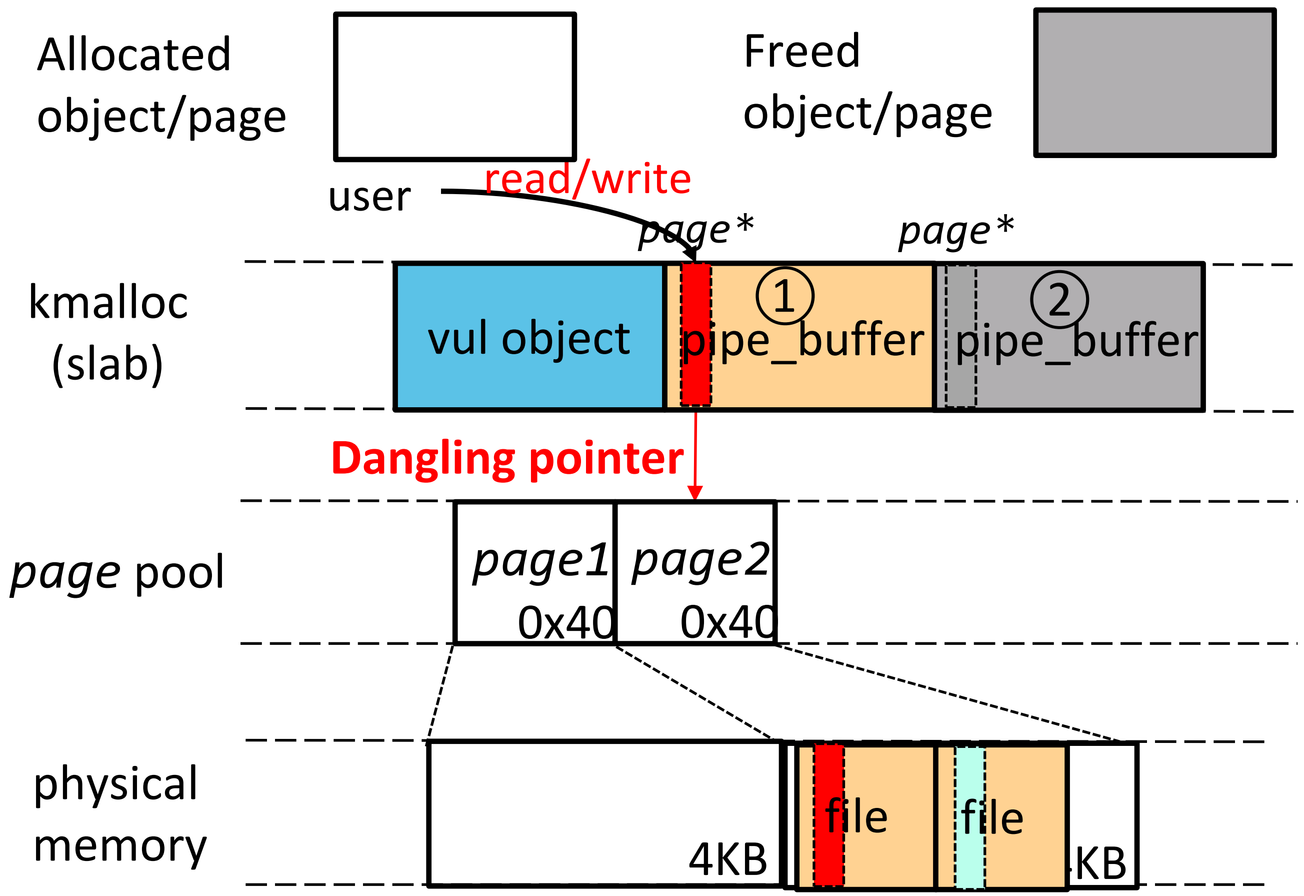


Slub page reuse:

- Spray many critical objects to claim the freed page in the buddy system.
- The page is full of critical objects, which is used as a page of its slub cache.
- It access the critical objects easily without cross cache attack, can bypass SLAB_VIRTUAL.

PageJack: tamper with critical objects

Step 5 Read/Write critical objects: we can read/write the whole 4KB physical page through the dangling pointer.



Read/Write the whole page (arbitrary read/write):

- Linux kernel provides the read/write interfaces based on *a* `struct page *`, such as `copy_page_from_iter`, `copy_page_to_iter`.
- Corrupt `file->f_mode` to gain root privilege.

CVE-2022-0995

- An out-of-bounds (OOB) memory write in the watch_queue event notification subsystem.

```
long watch_queue_set_filter(…)
{
        …
        if (copy_from_user(&filter, _filter, sizeof(filter)) != 0)
                return -EFAULT;
        …
        tf = memdup_user(_filter->filters, filter.nr_filters * sizeof(*tf));
        for (i = 0; i < filter.nr_filters; i++) {
                …
                if (tf[i].type >= sizeof(wfilter->type_filter) * 8)
                        continue
                nr_filter++;
        }
        …
        wfilter = kzalloc(struct_size(wfilter, filters, nr_filter), GFP_KERNEL);
         …
        wfilter->nr_filters = nr_filter;
        q = wfilter->filters;
        for (i = 0; i < filter.nr_filters; i++) {
                if (tf[i].type >= sizeof(wfilter->type_filter) * BITS_PER_LONG)
                        continue;

                q->type                 = tf[i].type;
                …
                __set_bit(q->type, wfilter->type_filter);
                q++;
        }
}
```

type compare with 0x80 when compute the number of filters

but type compare with 0x400 when populate and set_bit!!!

Primitive #1:

Out of bound populate!

```
for (i = 0; i < filter.nr_filters; i++) {
        if (tf[i].type >= sizeof(wfilter->type_filter) * BITS_PER_LONG)
                continue;
        q->type = tf[i].type;
        /*use q to copy the filter*/
        …
        q++;
  }
```

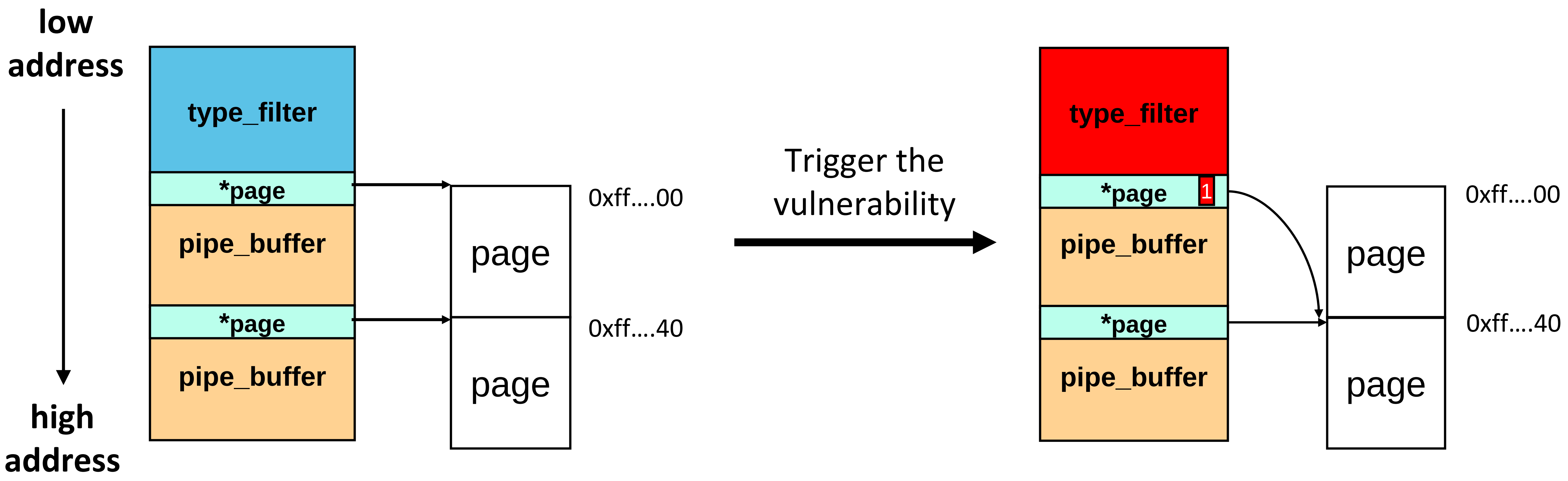
Primitive #2 :

Out of bound set_bit!

```
#define BIT_MASK(nr) (UL(1) << ((nr) % BITS_PER_LONG))
#define BIT_WORD(nr) ((nr) / BITS_PER_LONG)
static inline void __set_bit(int nr, volatile unsigned long *addr)
{
        unsigned long mask = BIT_MASK(nr);
        unsigned long *p = ((unsigned long *)addr) + BIT_WORD(nr);
        *p  |= mask;
}
```

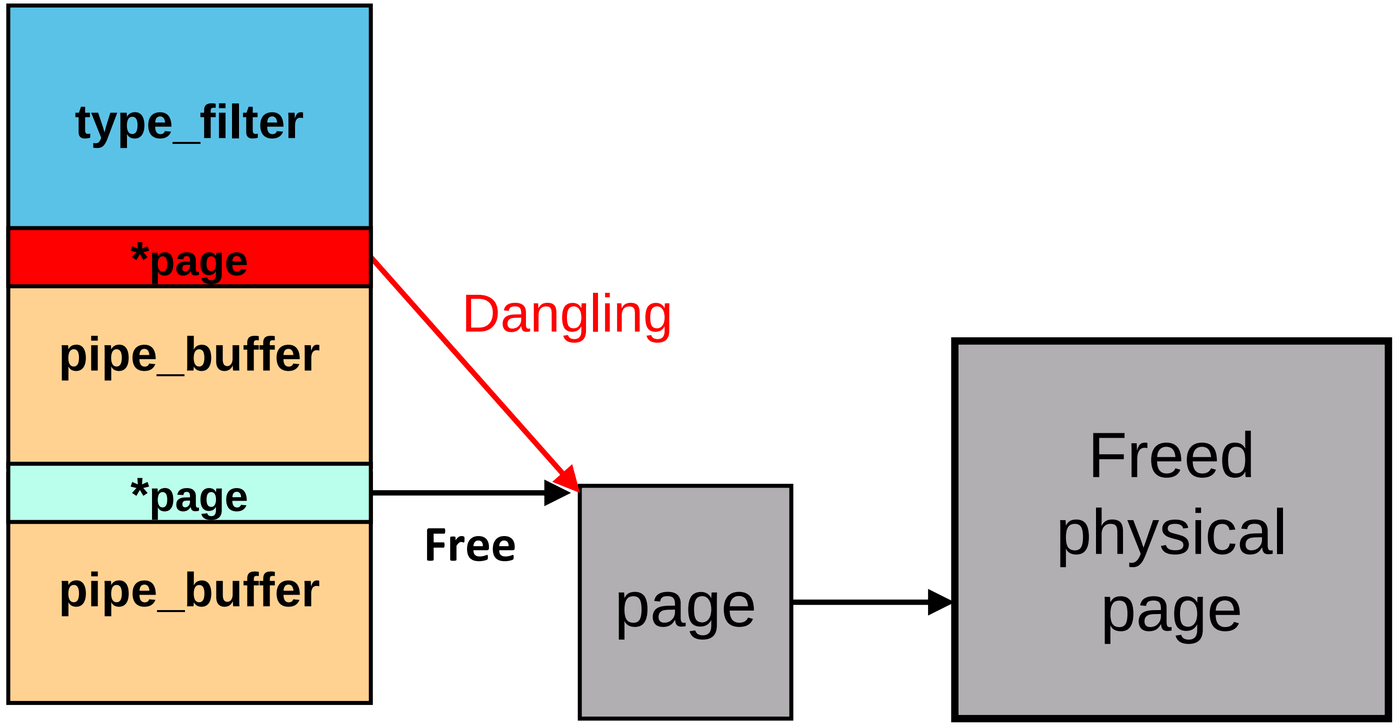
Exploit CVE-2022-0995

- We use **primitive #2** for exploit , modify the 6th bit of the page* in the pipe_buffer, making two pipe_buffer->page points to the **same page**.



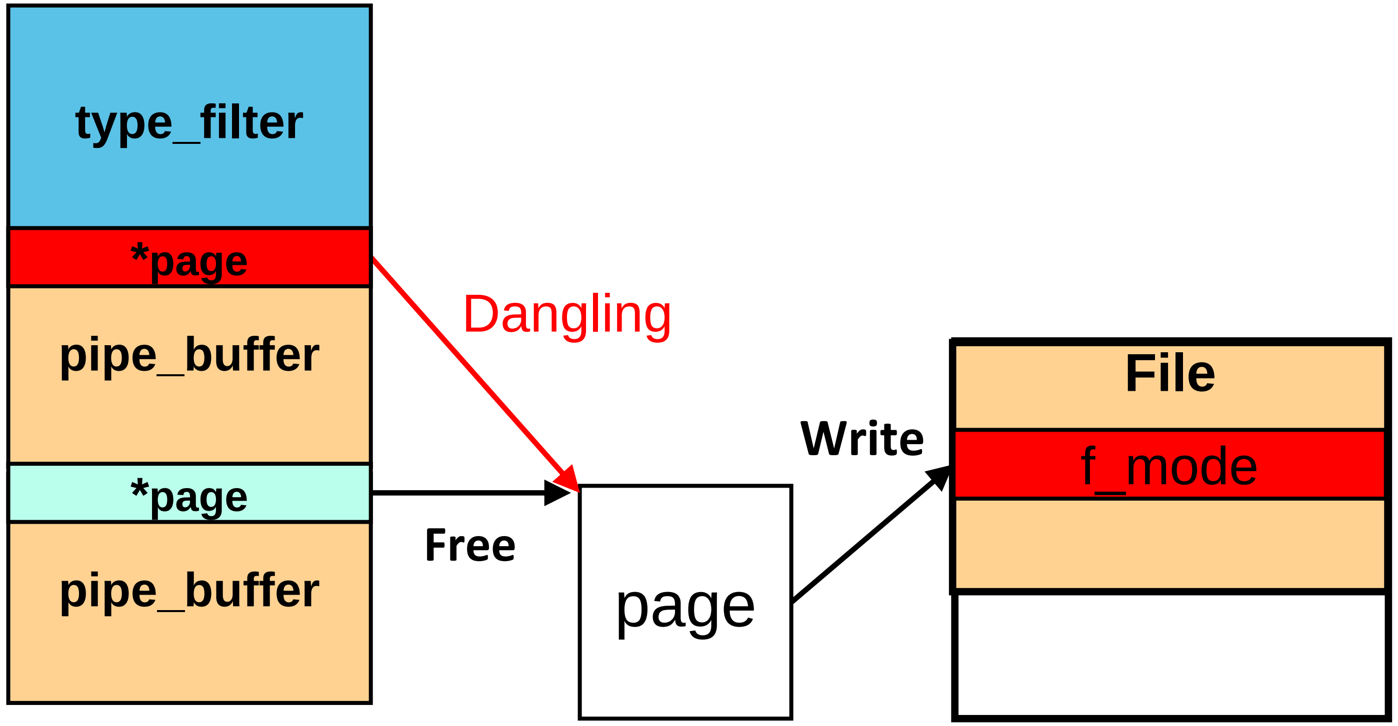
Exploit CVE-2022-0995

- Close one of the pipe_buffer to free the page, creating page UAF



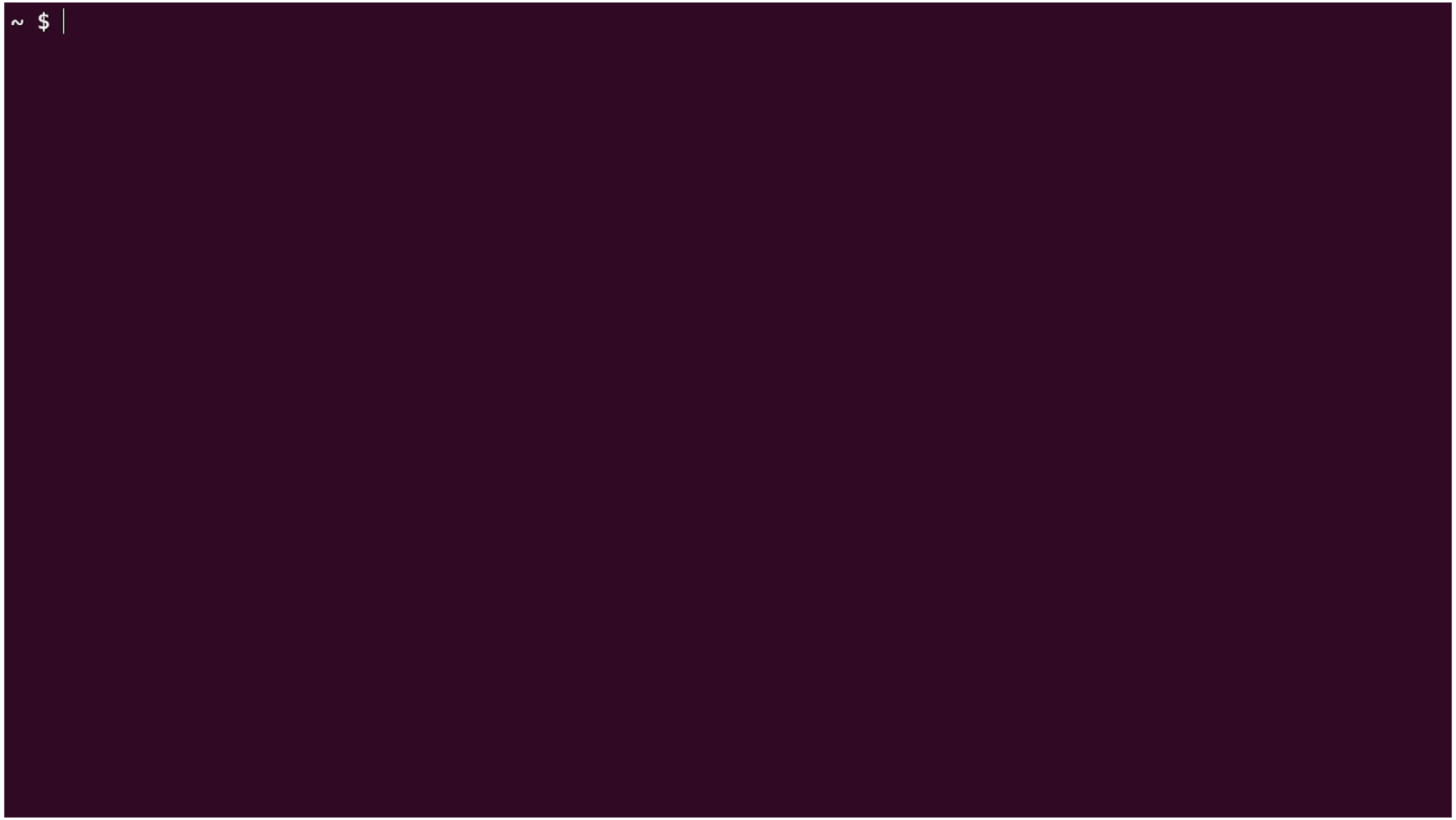
Exploit CVE-2022-0995

- Spray "/etc/passwd" or suid struct file objects to realloc the uaf page.
- Write to uaf pipe_buffer to **modify** the file->f_mode to O_RW.
- Edit the passwd or suid file to get root.



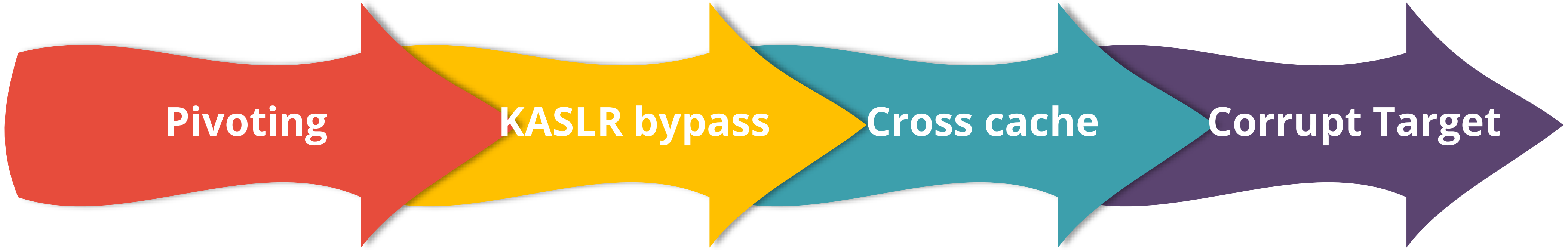
Demo: SLAB_VIRTUAL and CFI enabled

Demo: SLAB_VIRTUAL and CFI enabled

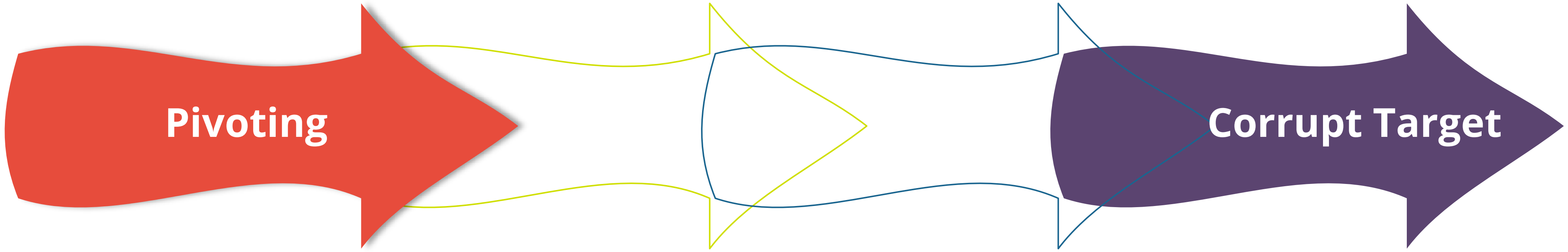


Advantages of PageJack

Typical:



PageJack:



Black Hat Sound Bytes

- **A novel OS kernel data-only exploit technique**

  Bypass CFI

- **Applicable for a variety of vulnerabilities in the real world**

  Linux and Android, vul type: OOB, UAF, double free

- **Bypass mitigations, fewer steps, and improve stability**

  KASLR, SLAB_VIRTUAL

Thank you!

More exploits with PageJack: https://github.com/Lotuhu/Page-UAF

White paper: https://arxiv.org/abs/2401.17618

zhiyunq@cs.ucr.edu

https://www.cs.ucr.edu/~zhiyunq

@pkqzy888

https://github.com/seclab-ucr

UC RIVERSIDE