black hat USA 2023

AUGUST 9-10, 2023

BRIEFINGS

Bad io_uring: A New Era of Rooting for Android

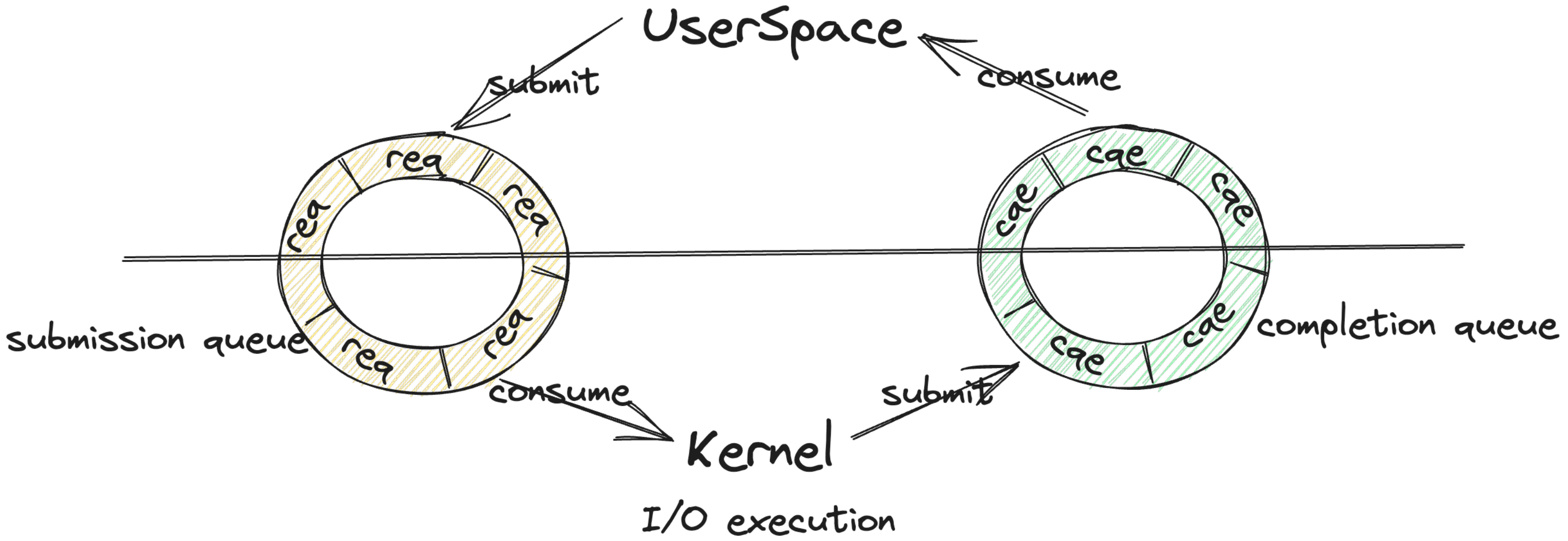
Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, Kang Li

Who We Are

- **Zhenpeng Lin**
  - Ph.D. from *Northwestern University*
  - Specialized in ***kernel security***
- **Xinyu Xing**
  - Associate Professor at *Northwestern University*
- **Zhaofeng Chen**
  - Principle Researcher at *Certik*
- **Kang Li**
  - Chief Security Officer at *Certik*

The io_uring

- Efficient I/O operations
- Less Syscalls
- Under ***ACTIVE*** development



The BAD io_uring

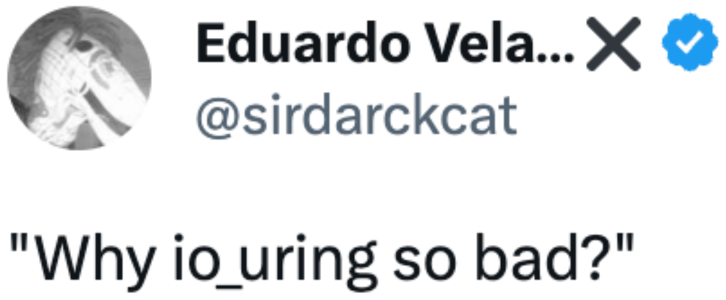
Eduardo Vela... ✕

@sirdarckcat

"Why io_uring so bad?"

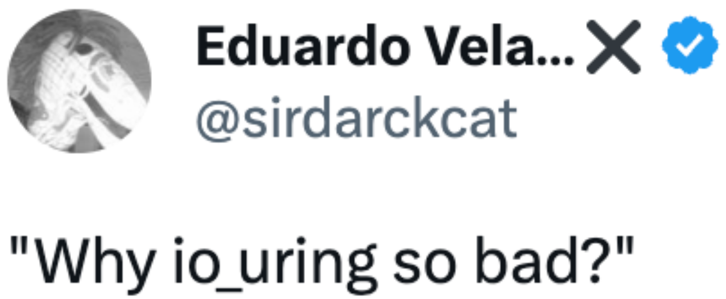
The BAD io_uring

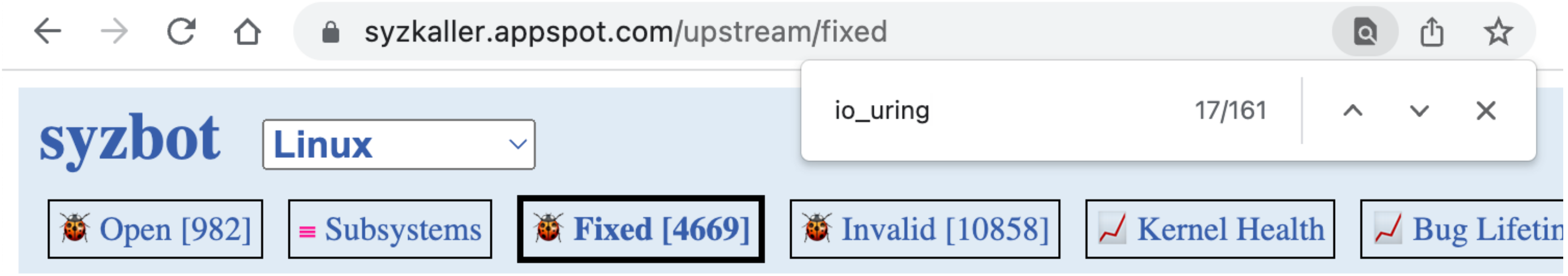
- Very buggy



The BAD io_uring

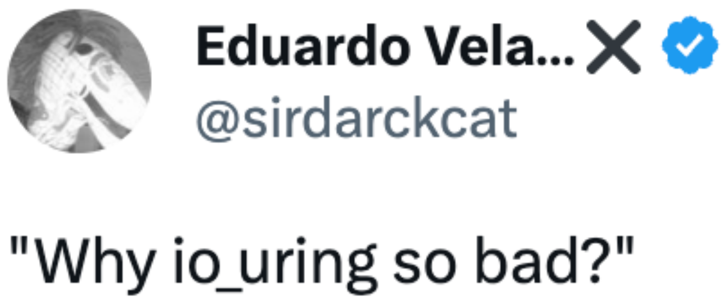
- Very buggy

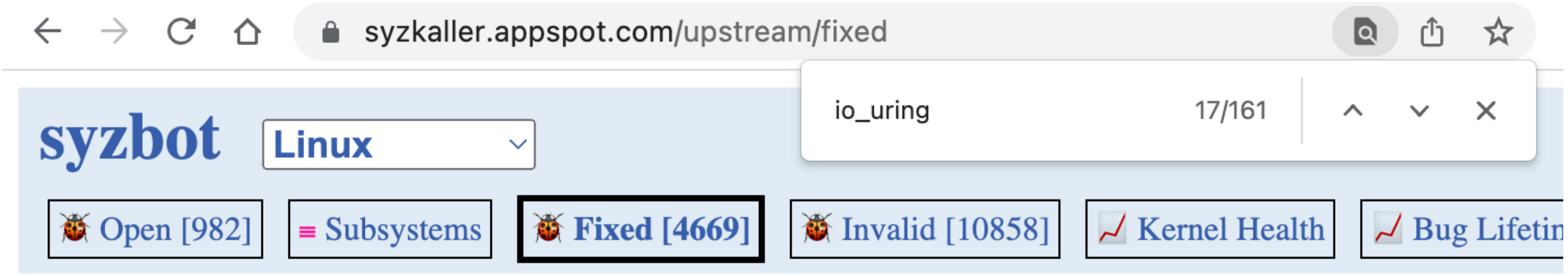




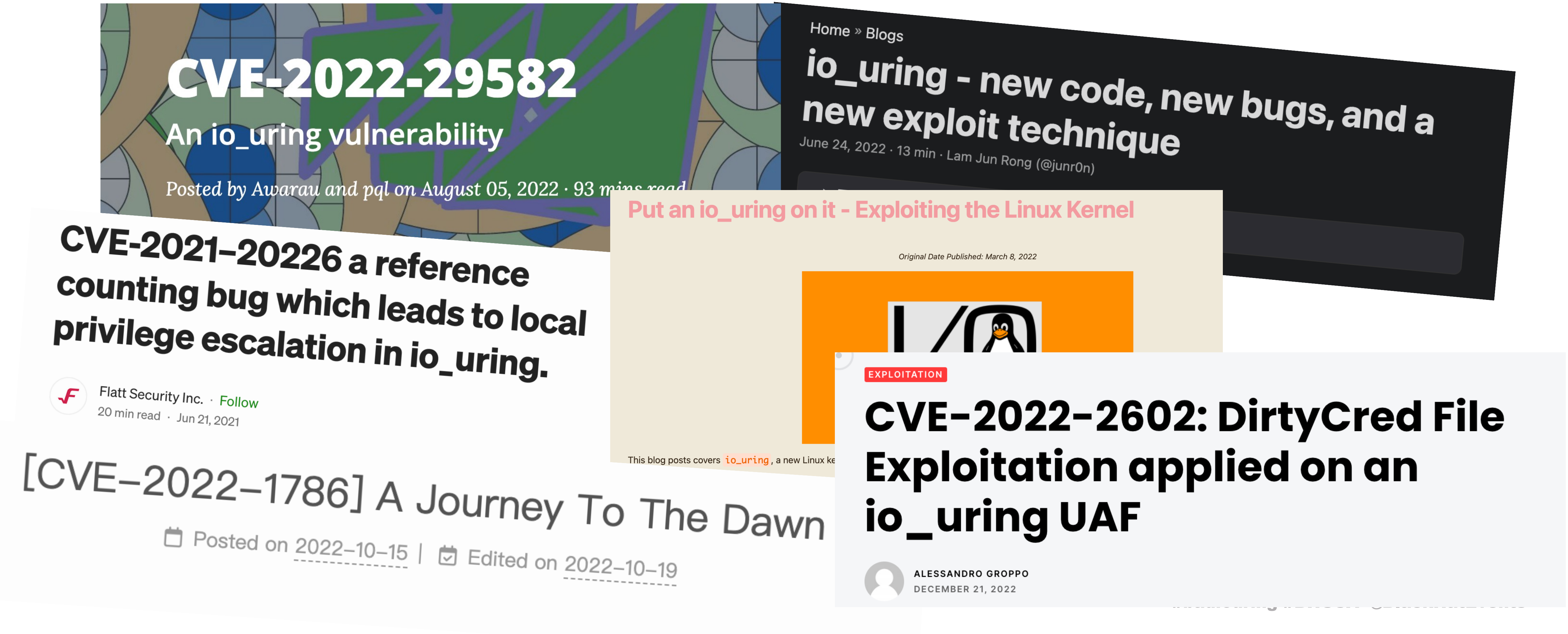
The BAD io_uring

- Very buggy
- Active development, and ***ACTIVE exploitation***





Exploitation Against io_uring



Exploitation Against io_uring

- 60% submissions to KCTF VRP exploited io_uring as of June 2023
- Around 1 million USD paid out for those bugs
- All public exploits targeted desktop Linux kernel

Exploitation Against io_uring

- 60% submissions to KCTF VRP exploited io_uring as of June 2023
- Around 1 million USD paid out for those bugs
- All public exploits targeted desktop Linux kernel

- Measures taken by Google
  - ChromeOS: io_uring disabled
  - Google servers: io_uring disabled
  - GKE AutoPilot: investigating disabling io_uring by default
  - Android: io_uring ***restricted***

Exploitation Against io_uring

- 60% submissions to KCTF VRP exploited io_uring as of June 2023
- Around 1 million USD paid out for those bugs
- All public exploits targeted desktop Linux kernel

- Measures taken by Google
  - ChromeOS: io_uring disabled
  - Google servers: io_uring disabled
  - GKE AutoPilot: investigating disabling io_uring by default
  - Android: io_uring ***restricted***
    - still accessible from ***privileged*** context (e.g., adb)

Exploiting io_uring on Android

- A lot of bugs, a lot of potential!

Exploiting io_uring on Android

- A lot of bugs, a lot of potential!
- 🤓 Fun and profit!

Code execution reward amounts

Description	Maximum Reward
Pixel Titan M with Persistence, Zero click	Up to $1,000,000
Pixel Titan M without Persistence, Zero click	Up to $500,000
Local App to Pixel Titan M without Persistence	Up to $300,000
Secure Element	Up to $250,000
Trusted Execution Environment	Up to $250,000
Kernel	Up to $250,000
Privileged Process	Up to $100,000

Exploiting io_uring on Android

- A lot of bugs, a lot of potential!
- 🤓 Fun and profit!
- ☹️ No public writeup for exploiting it on Android

Code execution reward amounts

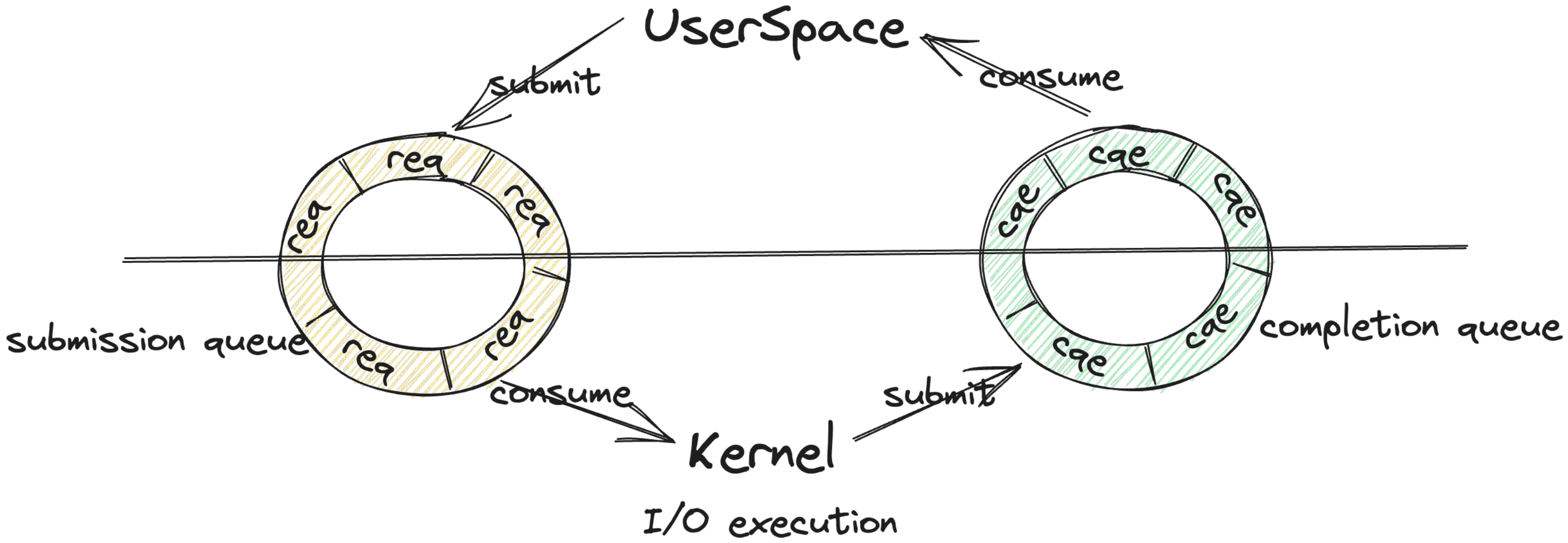
Description	Maximum Reward
Pixel Titan M with Persistence, Zero click	Up to $1,000,000
Pixel Titan M without Persistence, Zero click	Up to $500,000
Local App to Pixel Titan M without Persistence	Up to $300,000
Secure Element	Up to $250,000
Trusted Execution Environment	Up to $250,000
Kernel	Up to $250,000
Privileged Process	Up to $100,000

CVE-2022-20409

- No difference than other io_uring bugs
- A stable **invalid-free** bug
- The bug I used to *pwn* ***Google Pixel 6*** and ***Samsung S22*** *in 2022*
- Fixed on 7/29/2022

io_uring's AsyncIO

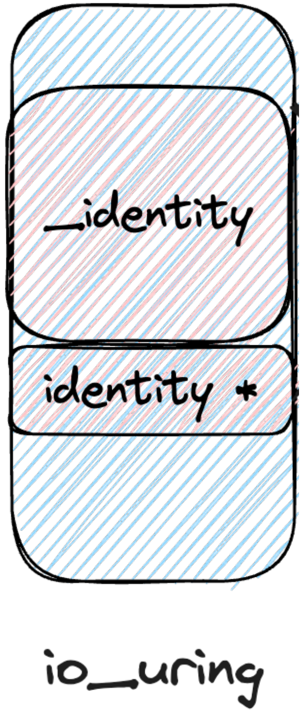
- Each I/O operation is a *req* in the submission queue
- Each req can be processed *asynchronously*
- Each req has its *identity*



Initializing identity

- *identity* stores in *io_uring*

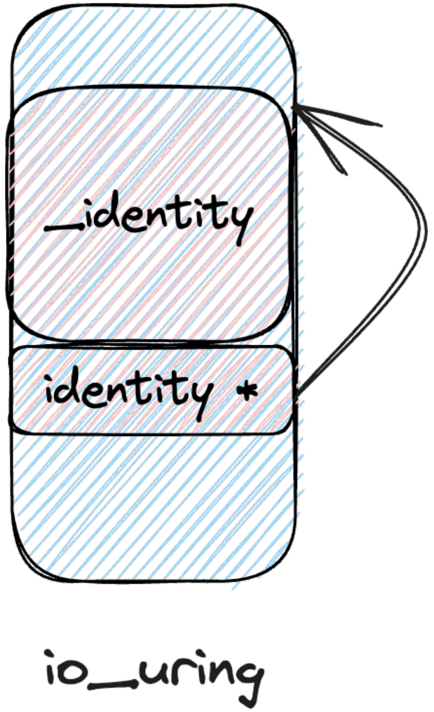
```c
int io_uring_alloc_task_context(struct task_struct *task)
{
    struct io_uring_task *tctx;
    tctx = kmalloc(sizeof(*tctx), GFP_KERNEL);
    ...
    io_init_identity(&tctx->__identity);
    tctx->identity = &tctx->__identity;
    task->io_uring = tctx;
}
```



Initializing identity

- *identity* stores in *io_uring*
- *identity* references to the nested *__identity*

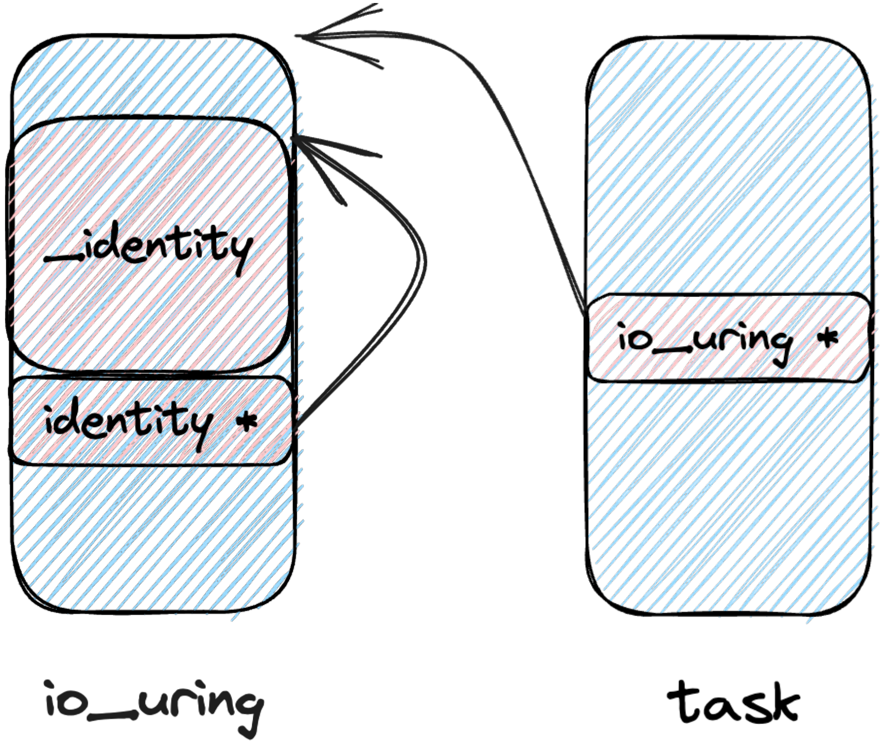
```
int io_uring_alloc_task_context(struct task_struct *task)
{
    struct io_uring_task *tctx;
    tctx = kmalloc(sizeof(*tctx), GFP_KERNEL);
    ...
    io_init_identity(&tctx->__identity);
    tctx->identity = &tctx->__identity;
    task->io_uring = tctx;
}
```



Initializing identity

- *identity* stores in *io_uring*
- *identity* references to the nested *__identity*
- *io_uring* is referenced by *task*

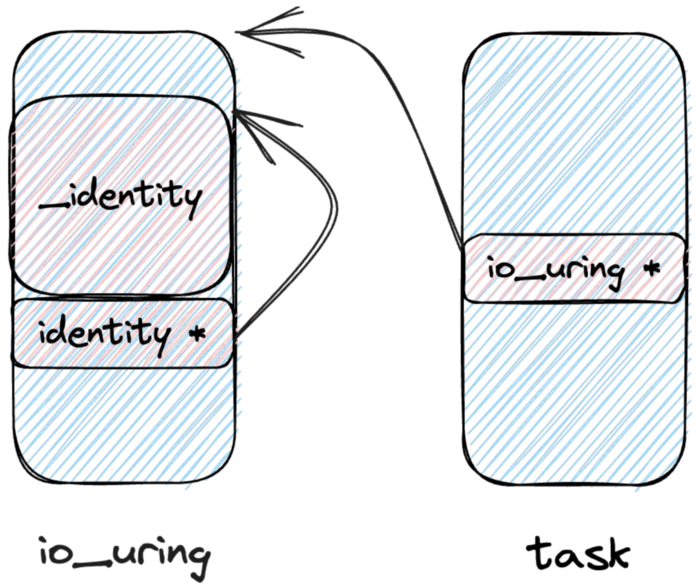
```
int io_uring_alloc_task_context(struct task_struct *task)
{
    struct io_uring_task *tctx;
    tctx = kmalloc(sizeof(*tctx), GFP_KERNEL);
    ...
    io_init_identity(&tctx->__identity);
    tctx->identity = &tctx->__identity;
    task->io_uring = tctx;
}
```



identity COW

- If *identity* changes (e.g., cred changes), new *identity* is created

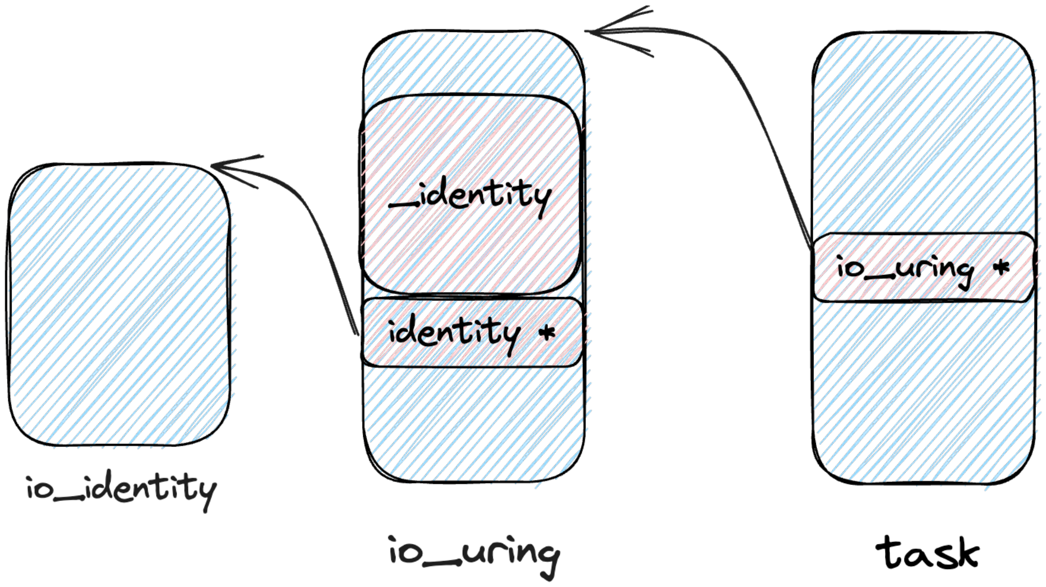
```
static bool io_identity_cow(struct io_kiocb *req)
{
    struct io_uring_task *tctx = current->io_uring;
    struct io_identity *id;
    ...
    id = kmemdup(req->work.identity, sizeof(*id),
GFP_KERNEL);
    io_init_identity(id);
    ...
    req->work.identity = id;
    tctx->identity = id;
}
```



identity COW

- If *identity* changes (e.g., cred changes), new *identity* is created
- *identity ** will reference to the new *identity* on heap

```
static bool io_identity_cow(struct io_kiocb *req)
{
    struct io_uring_task *tctx = current->io_uring;
    struct io_identity *id;
    ...
    id = kmemdup(req->work.identity, sizeof(*id),
GFP_KERNEL);
    io_init_identity(id);
    ...
    req->work.identity = id;
    tctx->identity = id;
}
```



The BUG

```
static bool io_identity_cow(struct io_kiocb *req)
{
    struct io_uring_task *tctx = current->io_uring;
    ...
    /* drop tctx and req identity references, if needed */
    if (tctx->identity != &tctx->__identity &&
        refcount_dec_and_test(&tctx->identity->count))
        kfree(tctx->identity);



    if (req->work.identity != &tctx->__identity &&
        refcount_dec_and_test(&req->work.identity->count))
        kfree(req->work.identity);


    req->work.identity = id;
    tctx->identity = id;
    return true;
}
```

The BUG

```
static bool io_identity_cow(struct io_kiocb *req)
{
    struct io_uring_task *tctx = current->io_uring;
    ...
    /* drop tctx and req identity references, if needed */
    if (tctx->identity != &tctx->__identity &&
        refcount_dec_and_test(&tctx->identity->count))
        kfree(tctx->identity);



    if (req->work.identity != &tctx->__identity &&
        refcount_dec_and_test(&req->work.identity->count))
        kfree(req->work.identity);


    req->work.identity = id;
    tctx->identity = id;
    return true;
}
```

The BUG

```
static bool io_identity_cow(struct io_kiocb *req)
{
    struct io_uring_task *tctx = current->io_uring;
    ...
    /* drop tctx and req identity references, if needed */
    if (tctx->identity != &tctx->__identity &&
        refcount_dec_and_test(&tctx->identity->count))
        kfree(tctx->identity);



    if (req->work.identity != &tctx->__identity &&
        refcount_dec_and_test(&req->work.identity->count))
        kfree(req->work.identity);


    req->work.identity = id;
    tctx->identity = id;
    return true;
}
```

The BUG

thread A

thread B

```
static bool io_identity_cow(struct io_kiocb *req)
{
    struct io_uring_task *tctx = current->io_uring;
    ...
    /* drop tctx and req identity references, if needed */
    if (tctx->identity != &tctx->__identity &&
        refcount_dec_and_test(&tctx->identity->count))
        kfree(tctx->identity);


    if (req->work.identity != &tctx->__identity &&
        refcount_dec_and_test(&req->work.identity->count))
        kfree(req->work.identity);


    req->work.identity = id;
    tctx->identity = id;
    return true;
}
```

The BUG

thread A

```c
static bool io_identity_cow(struct io_kiocb *req)
{
```

thread B

```c
    struct io_uring_task *tctx = current->io_uring;
```

This is false

```c
    ...
    /* drop tctx and req identity references, if needed */
    if (tctx->identity != &tctx->__identity &&
        refcount_dec_and_test(&tctx->identity->count))
        kfree(tctx->identity);


    if (req->work.identity != &tctx->__identity &&
        refcount_dec_and_test(&req->work.identity->count))
        kfree(req->work.identity);


    req->work.identity = id;
    tctx->identity = id;
    return true;
}
```

The BUG

thread A

thread B

This is false

thread A->io_uring->identity

```
static bool io_identity_cow(struct io_kiocb *req)
{
    struct io_uring_task *tctx = current->io_uring;
    ...
    /* drop tctx and req identity references, if needed */
    if (tctx->identity != &tctx->__identity &&
        refcount_dec_and_test(&tctx->identity->count))
        kfree(tctx->identity);

    if (req->work.identity != &tctx->__identity &&
        refcount_dec_and_test(&req->work.identity->count))
        kfree(req->work.identity);


    req->work.identity = id;
    tctx->identity = id;
    return true;
}
```

The BUG

thread A

thread B

This is false

```
static bool io_identity_cow(struct io_kiocb *req)
{
    struct io_uring_task *tctx = current->io_uring;
    ...
    /* drop tctx and req identity references, if needed */
    if (tctx->identity != &tctx->__identity &&
        refcount_dec_and_test(&tctx->identity->count))
        kfree(tctx->identity);

    if (req->work.identity != &tctx->__identity &&
        refcount_dec_and_test(&req->work.identity->count))
        kfree(req->work.identity);

    req->work.identity = id;
    tctx->identity = id;
    return true;
}
```

thread A->io_uring->identity

This is true

The BUG

thread A

```
static bool io_identity_cow(struct io_kiocb *req)
{
```

thread B

```
    struct io_uring_task *tctx = current->io_uring;
```

This is false

```
    ...
    /* drop tctx and req identity references, if needed */
    if (tctx->identity != &tctx->__identity &&
        refcount_dec_and_test(&tctx->identity->count))
        kfree(tctx->identity);
```

thread A->io_uring->identity

```
    if (req->work.identity != &tctx->__identity &&
        refcount_dec_and_test(&req->work.identity->count))
        kfree(req->work.identity);
```

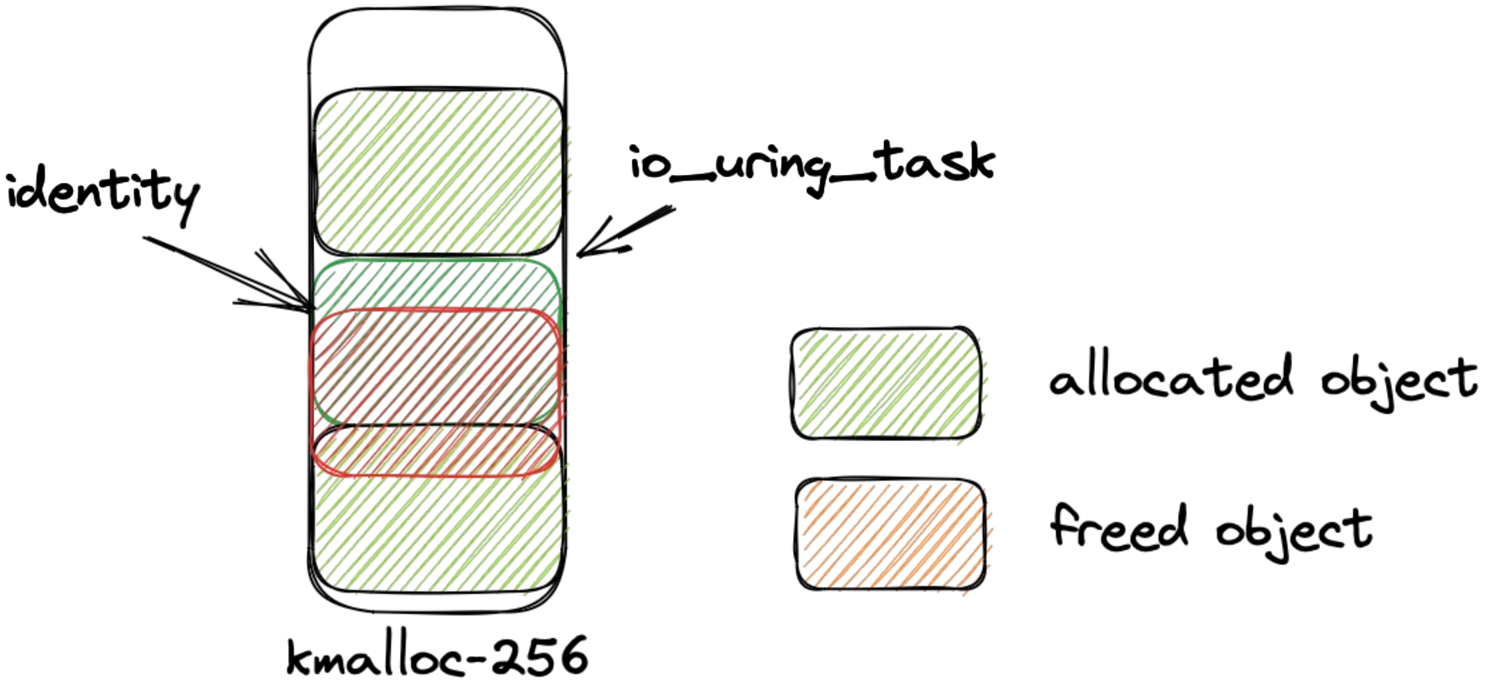
invalid free

This is true

```
    req->work.identity = id;
    tctx->identity = id;
    return true;
}
```

The Memory Corruption Capability

- Invalid-free a *kmalloc-256* object in the middle



Exploitation on Android

- Restricted Access
  - No user_ns
  - No FUSE, userfaultfd
  - No msg_msg, user_key_payload, etc.
  - Very limited choice of syscalls

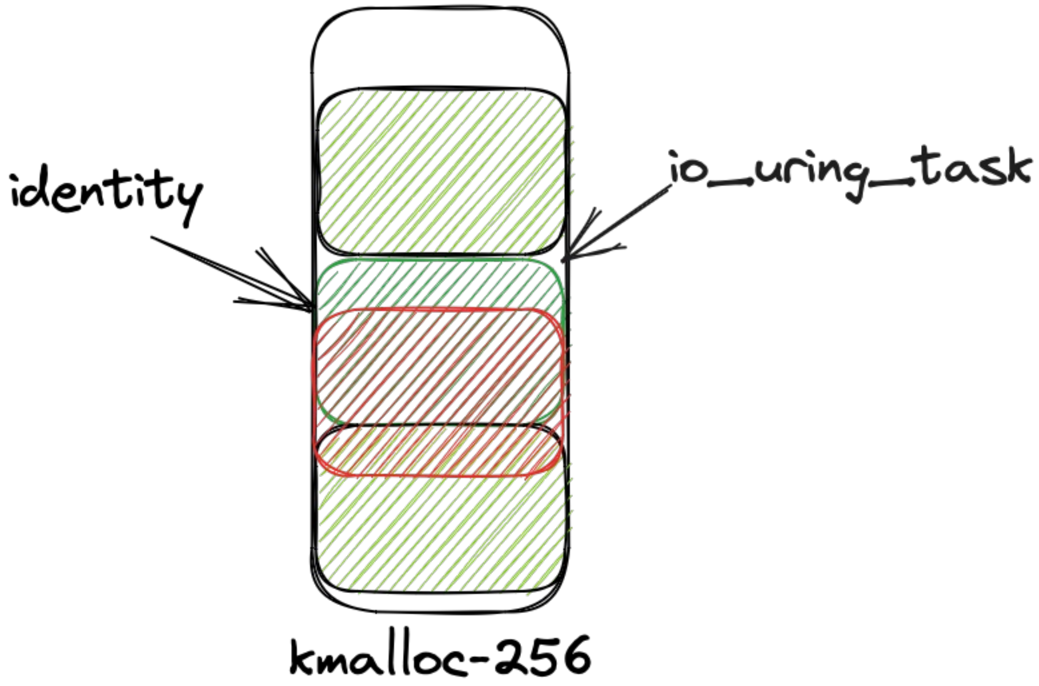
Exploitation on Android

- Restricted Access
  - No user_ns
  - No FUSE, userfaultfd
  - No msg_msg, user_key_payload, etc.
  - Very limited choice of syscalls
- But we have ***pipe*** 🧐
  - *pipe_buffer* is an ***elastic object*** --- good for spraying
  - *pipe_buffer* contains a global pointer --- good for leaking

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

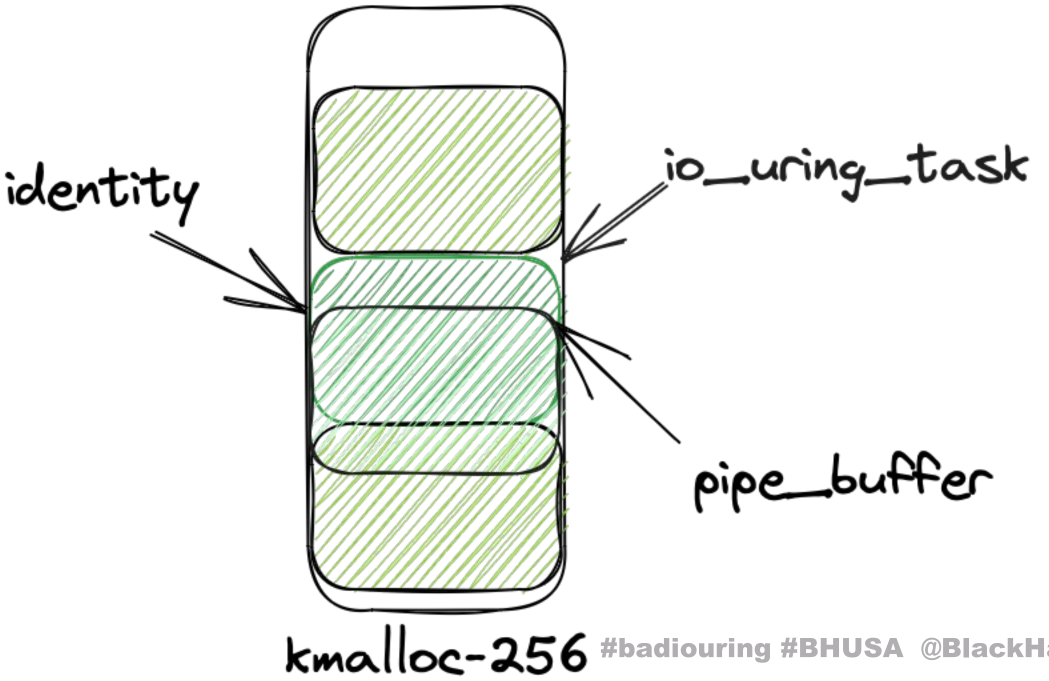
UAF from *identity* to *pipe_buffer*

- Trigger the invalid-free of *identity*, which frees *io_uring_task* in the middle



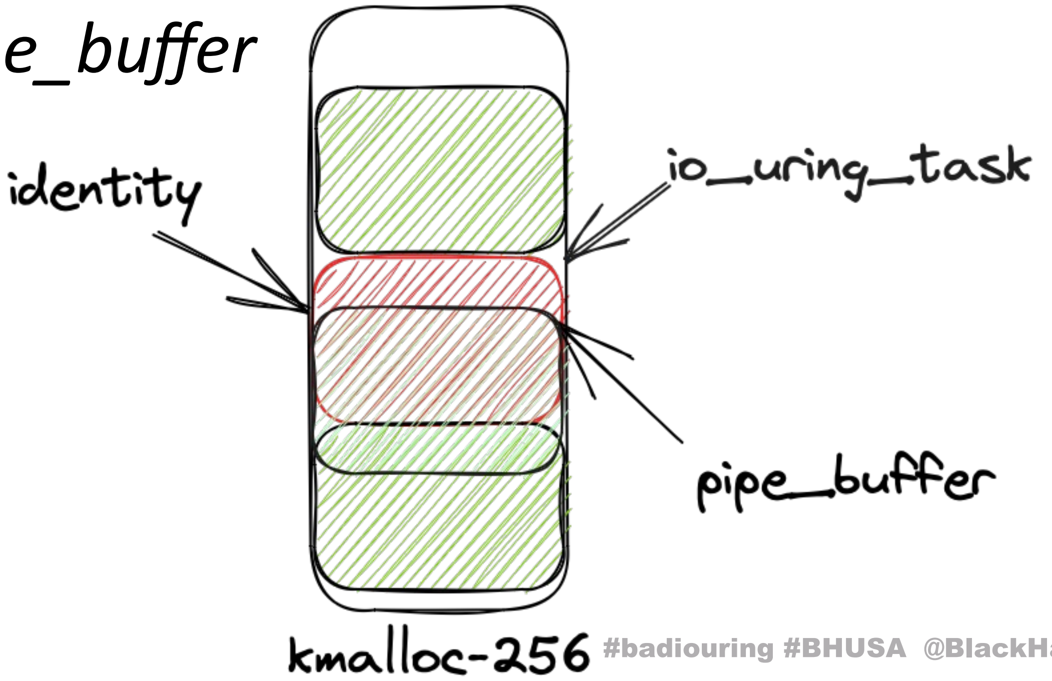
UAF from *identity* to *pipe_buffer*

- Trigger the invalid-free of *identity*, which frees *io_uring_task* in the middle
- Spray *pipe_buffer* in **kmalloc-256**



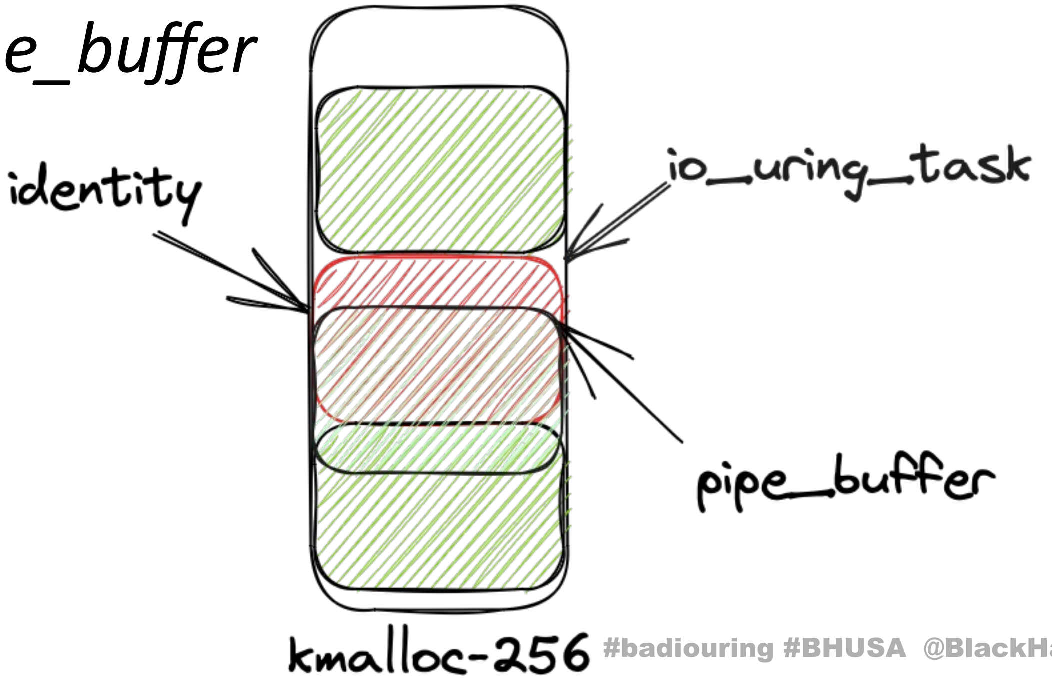
UAF from *identity* to *pipe_buffer*

- Trigger the invalid-free of *identity*, which frees *io_uring_task* in the middle
- Spray *pipe_buffer* in **kmalloc-256**
- Free *io_uring_task*, which frees *pipe_buffer*



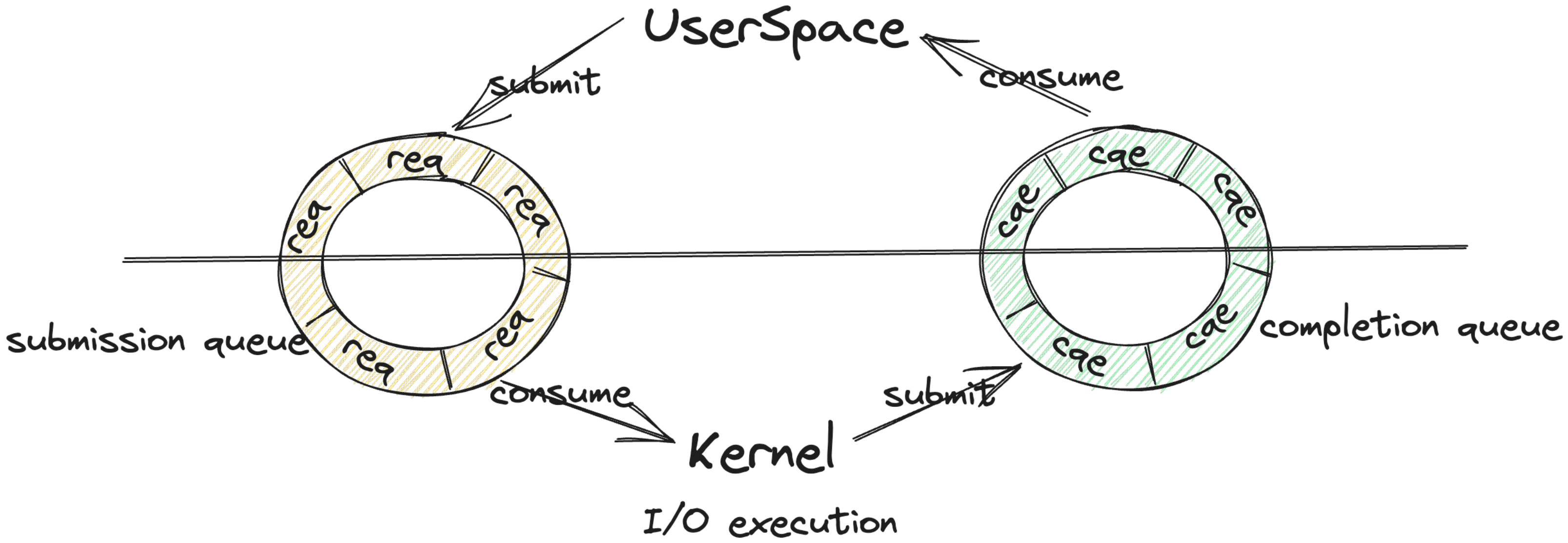
UAF from *identity* to *pipe_buffer*

- Trigger the invalid-free of *identity*, which frees *io_uring_task* in the middle
- Spray *pipe_buffer* in **kmalloc-256**
- Free *io_uring_task*, which frees *pipe_buffer*
- How to **leak** *pipe_buffer* out?



Recap of The io_uring Design

- The ***ring buffer*** is accessible to both userspace and kernel



The Shared Ring

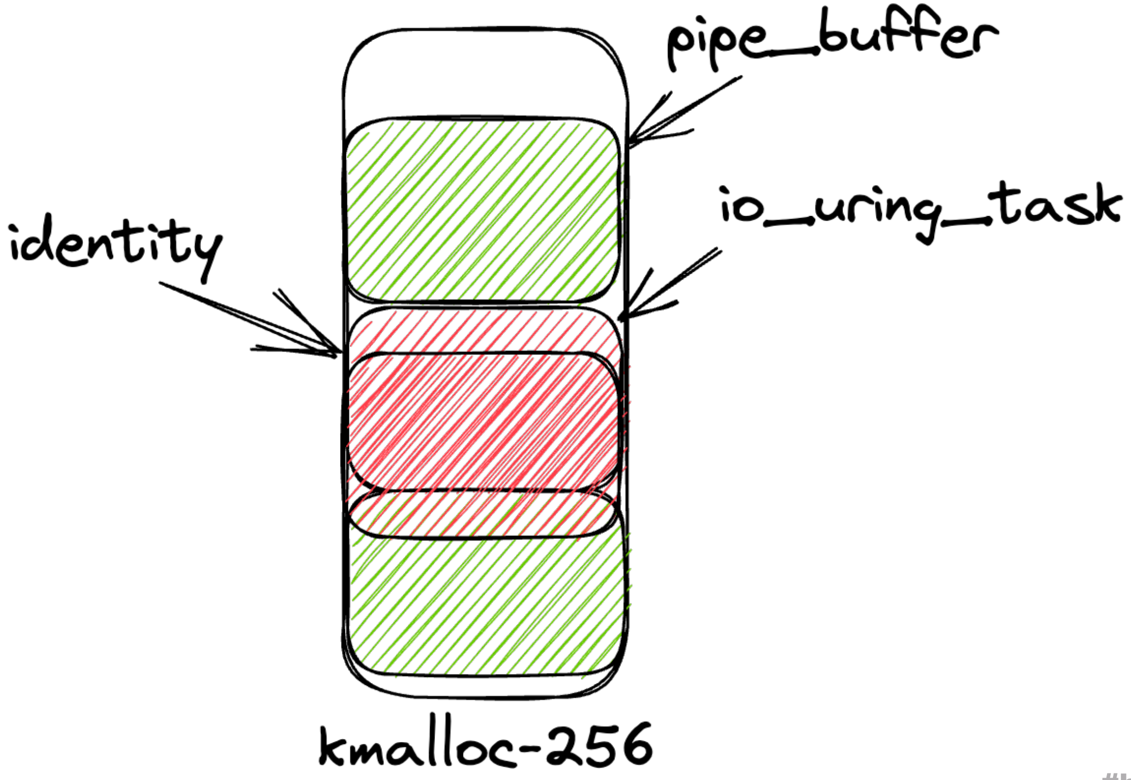
- **User** pages ***shared*** between kernel and userspace
- The memory is allocated by ***buddy allocator*** and mapped to userspace
- No copy_to/from_user is needed
- Date can be transported directly without copying
  - Read/write kernel memory from userspace
  - Read/write userspace memory from kernel

The "DirtyPage" Technique

- Some user pages are recycled with slab pages
  - **Spraying pages to reclaim freed slab pages**
  - Spray objects? No! We spray pages now!
  - Candidates: *io_uring, pipe*
- What is the advantage?
  - Powerful 🤓 : Read/write slab objects from userspace
  - Stable 🤓 : Spray once to have persist read/write on victim object
  - Simple 🤓 : Just allocate more

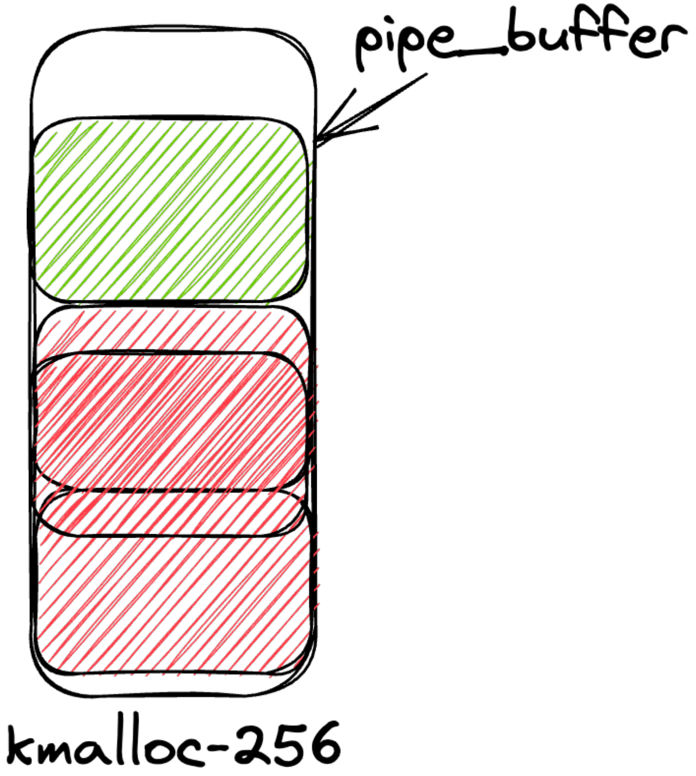
Achieving Read/Write on *pipe_buffer*

- Preparing the memory layout



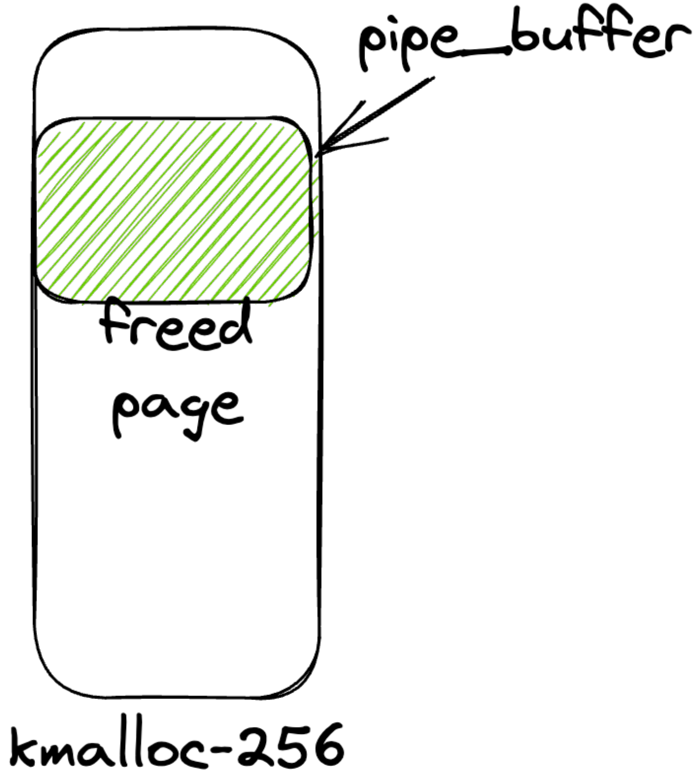
Achieving Read/Write on *pipe_buffer*

- Preparing the memory layout
- Triggering the invalid-free



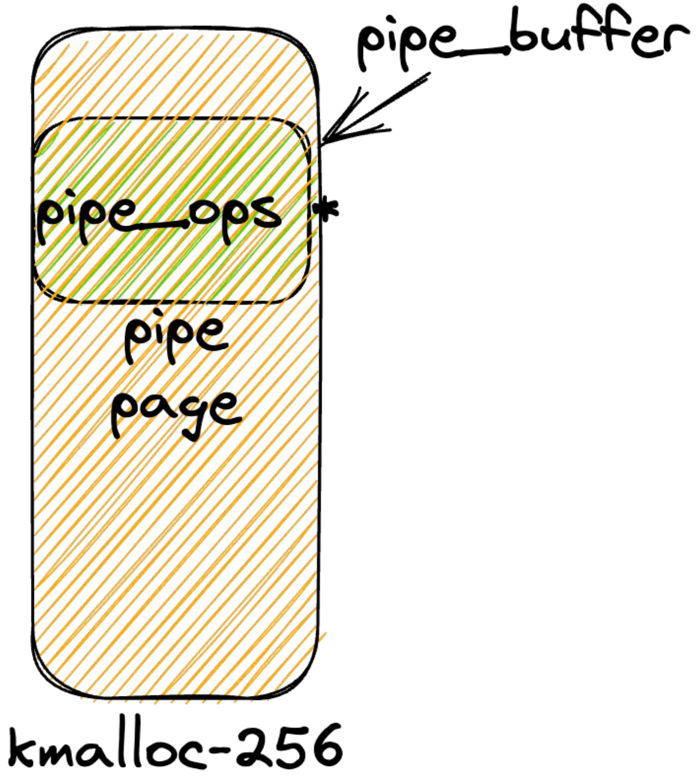
Achieving Read/Write on *pipe_buffer*

- Preparing the memory layout
- Triggering the invalid-free
- Freeing the slab page



Achieving Read/Write on *pipe_buffer*

- Preparing the memory layout
- Triggering the invalid-free
- Freeing the slab page
- Reclaiming the freed slab page



Achieving Read/Write on *pipe_buffer*

- Preparing the memory layout
- Triggering the invalid-free
- Freeing the slab page
- Reclaiming the freed slab page
- Reading *pipe_buffer*
  - *ops* --- **bypass kaslr**

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

Achieving Read/Write on *pipe_buffer*

- Preparing the memory layout
- Triggering the invalid-free
- Freeing the slab page
- Reclaiming the freed slab page
- Reading *pipe_buffer*
  - *ops* --- **bypass kaslr**
- Writing *pipe_buffer*
  - *flags* --- **Dirty Pipe Retro!**

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

Achieving Read/Write on *pipe_buffer*

- Preparing the memory layout
- Triggering the invalid-free
- Freeing the slab page
- Reclaiming the freed slab page
- Reading *pipe_buffer*
  - *ops* --- **bypass kaslr**
- Writing *pipe_buffer*
  - *flags* --- [Dirty Pipe Retro!]
  - *page* --- **arbitrary r/w** on kernel memory?

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

How Pipe Uses Pages

- ***kmap_atomic*** the page
- copy ***in/out*** the page

```
static ssize_t
pipe_read(struct kiocb *iocb, struct iov_iter *to) {
    ...
    // in copy_page_to_iter_iovec
    kaddr = kmap_atomic(page);
    from = kaddr + offset;
    left = copyout(buf, from, copy);
    ...
}

static ssize_t
pipe_write(struct kiocb *iocb, struct iov_iter *to)
{   ...
    // in copy_page_from_iter_iovec
    kaddr = kmap_atomic(page);
    to = kaddr + offset;
    left = copyin(to, buf, copy);
    ...
}
```

How Pipe Uses Pages

- ***kmap_atomic*** the page
- copy ***in/out*** the page
- ***kmap_atomic*** is ***page_address***

```
static inline void *kmap_atomic(struct page *page)
{
        preempt_disable();
        pagefault_disable();
        return page_address(page);
}
```

How Pipe Uses Pages

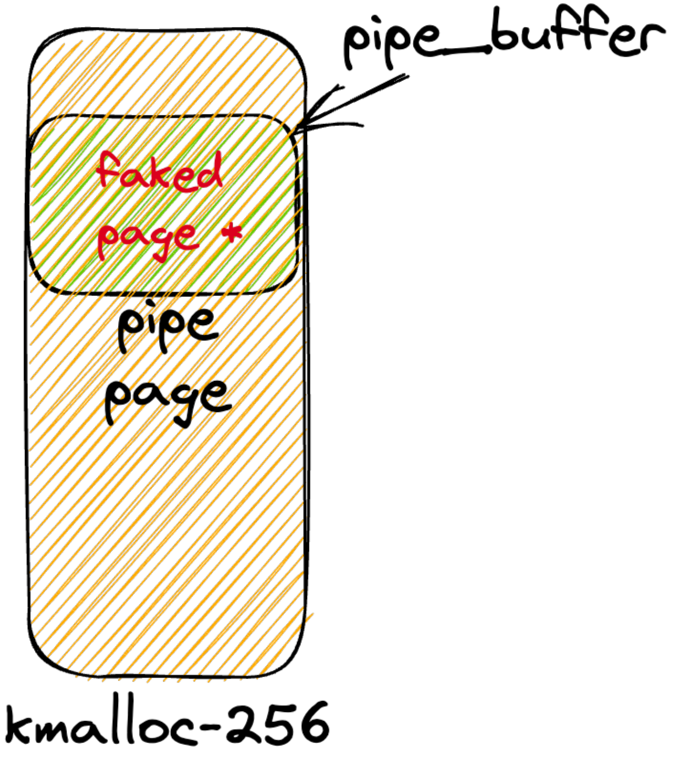
- ***kmap_atomic*** the page
- copy ***in/out*** the page
- ***kmap_atomic*** is ***page_address***
- ***page_address***
  - equals (page<<**SHIFT**)+**OFFSET**
  - **SHIFT** is fixed
  - **OFFSET** is also **fixed** on ARM64

```
#define page_address(x) page_to_virt(x)
#define page_to_virt(x) __va(PFN_PHYS(page_to_pfn(x)))
#define __va(x) ((void *)((unsigned long )(x)+PAGE_OFFSET))
#define PFN_PHYS(x)     ((phys_addr_t)(x) << PAGE_SHIFT)
```

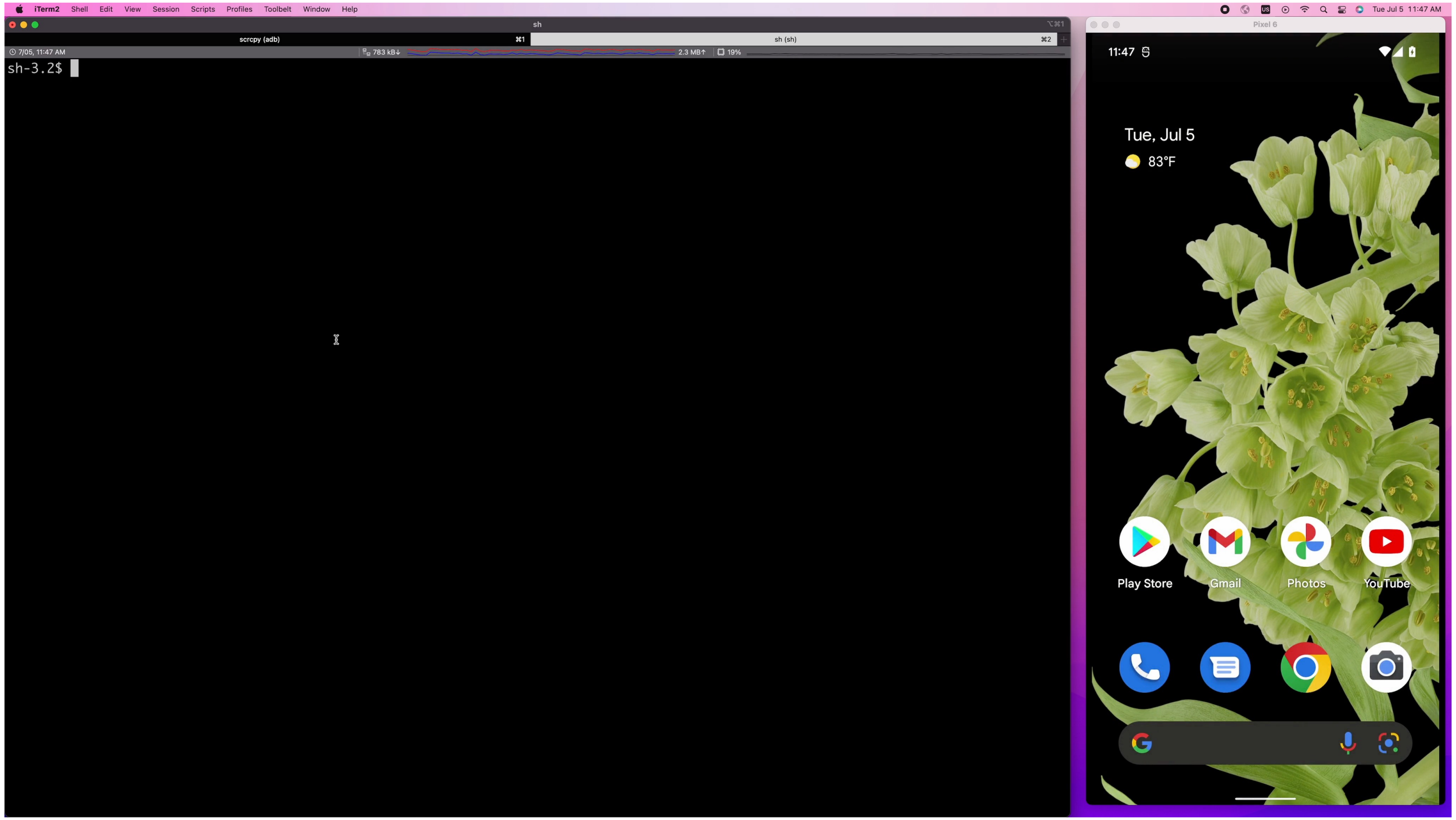
Achieving Kernel Arbitrary R/W

- Given a kernel address
  - Calculate the its page
  - Calculate the offset
  - Overwrite the *pipe_buffer* with calculated data
- *Read/Write* by reading/writing the pipe

```
unsigned long addr_to_page(unsigned long addr)
{ addr = addr & 0xfffffffffffff000ul;
  return ((addr - 0xffffc008000000ul) >> 6);
}
```



Escalating Privilege On Pixel 6



Samsung’s KNOX

- Samsung has customized protection for their kernel --- KNOX
- KNOX protects cred integrity

Samsung's KNOX

- Samsung has customized protection for their kernel --- KNOX
- KNOX protects cred integrity

Samsung's KNOX

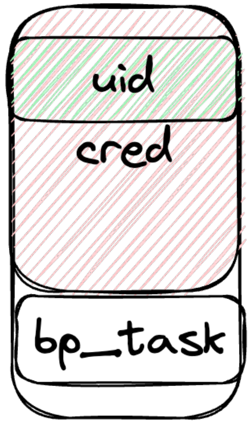
- Samsung has customized protection for their kernel --- KNOX
- KNOX protects cred integrity

```
struct cred_kdp {
    struct cred cred;
    atomic_t *use_cnt;
    struct task_struct *bp_task;
    void *bp_pgd;
    unsigned long long type;
};
```

Samsung's KNOX

- Samsung has customized protection for their kernel --- KNOX
- KNOX protects cred integrity

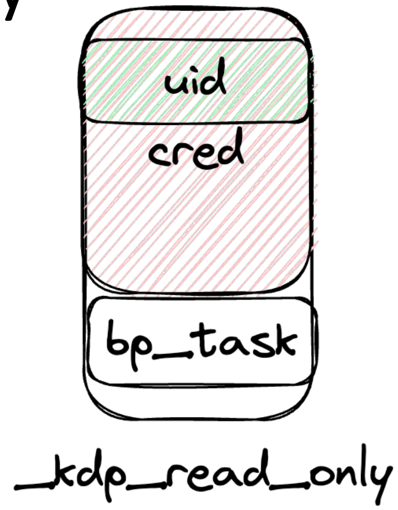
```
struct cred_kdp {
    struct cred cred;
    atomic_t *use_cnt;
    struct task_struct *bp_task;
    void *bp_pgd;
    unsigned long long type;
};
```



Samsung's KNOX

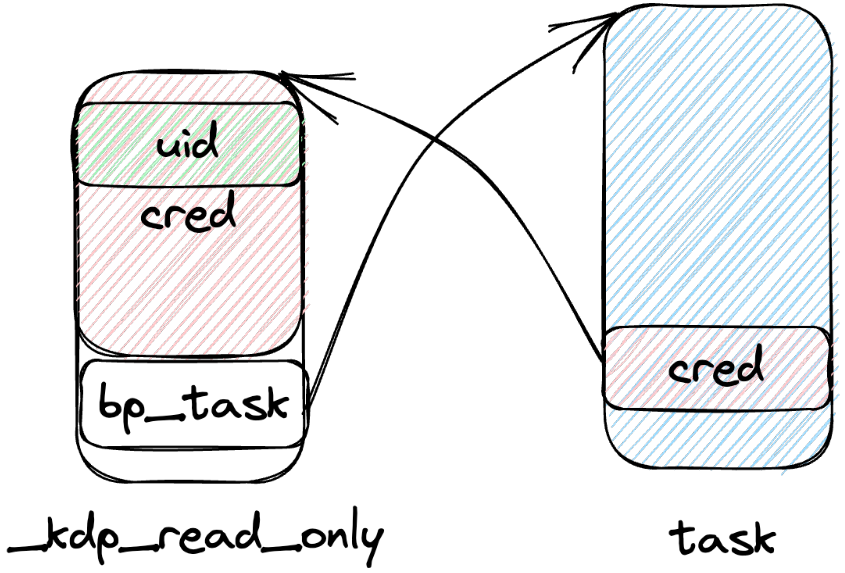
- Samsung has customized protection for their kernel --- KNOX
- KNOX protects cred integrity
- ***cred*** object is read-only, ***uid*** field is read-only

```
struct cred_kdp {
    struct cred cred;
    atomic_t *use_cnt;
    struct task_struct *bp_task;
    void *bp_pgd;
    unsigned long long type;
};
```



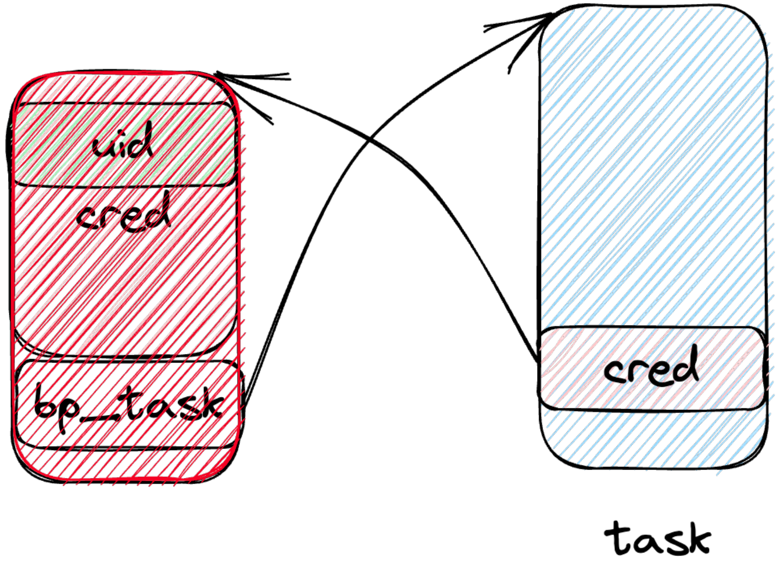
Validating cred Integrity

- Cross-checking between ***task*** and ***cred***
- Integrity is validated at syscall entry



Validating cred Integrity

- Cross-checking between ***task*** and ***cred***
- Integrity is validated at syscall entry
- How to prevent the cred is forged?

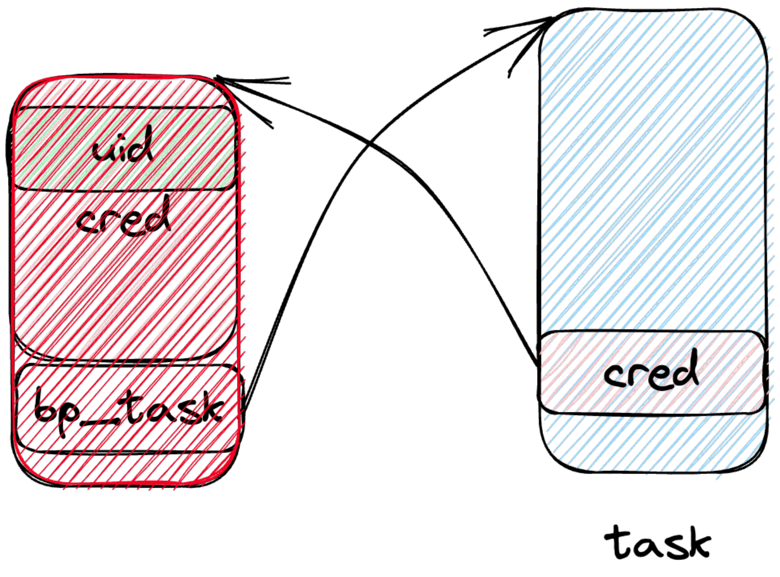


Validating cred Integrity

- How to prevent the cred is forged?
  - Checking if the ***cred*** is from ***cred_jar_ro/tsec_jar*** slab

```
/* Check whether the address belong to Cred Area */
int is_kdp_protect_addr(unsigned long addr)
{
    ...
    page = virt_to_head_page(objp);
    s = page->slab_cache;
    if (s && (s == cred_jar_ro || s == tsec_jar))
        return PROTECT_KMEM;

    return 0;
}
```

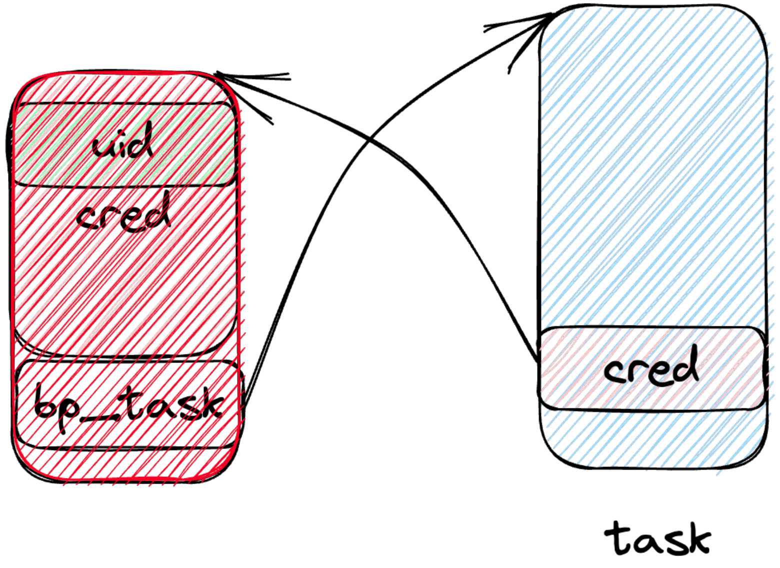


Validating cred Integrity

- How to prevent the cred is forged?
  - Checking if the ***cred*** is from ***cred_jar_ro/tsec_jar*** slab
  - This check is weak which could by bypassed

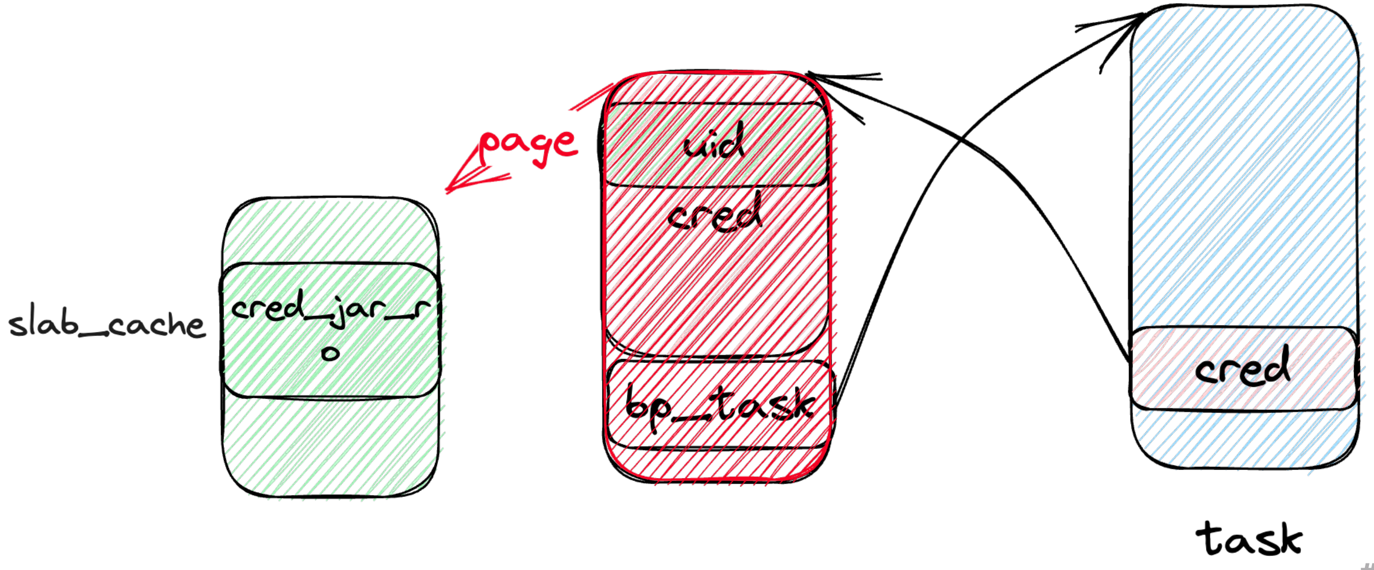
```
/* Check whether the address belong to Cred Area */
int is_kdp_protect_addr(unsigned long addr)
{
    ...
    page = virt_to_head_page(objp);
    s = page->slab_cache;
    if (s && (s == cred_jar_ro || s == tsec_jar))
        return PROTECT_KMEM;

    return 0;
}
```

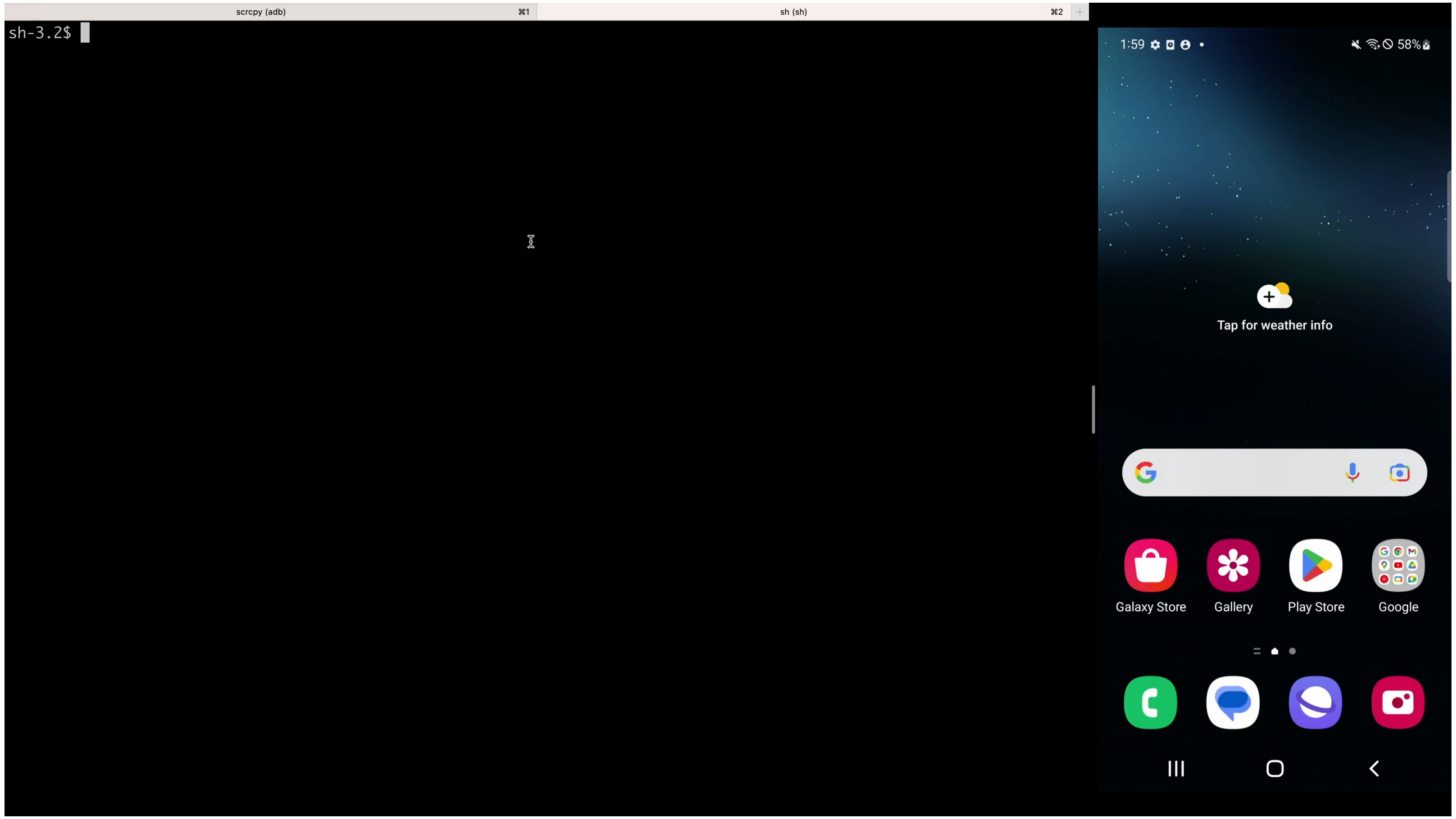


Bypassing KNOX

- Forging a ***root cred*** with correct references
- Tampering the ***slab_cache*** of the forged cred's page



Escalating Privilege On S22



Takeaways

- io_uring is a huge attack surface not only to desktop but also to AOSP
- ***Restricting*** io_uring on Android doesn't seem enough
- Object spray is not the only exploit option, try ***DirtyPage*****(**page spray**)**!
- Android kernel exploitation with ***DirtyPage*** is simple!

https://github.com/Markakd/bad_io_uring

@Markak_

https://zplin.me