Bad io_uring: New Attack Surface and New Exploit Technique to Rooting Android

Zhenpeng Lin[1], Xinyu Xing[1], Zhaofeng Chen[2], Kang Li[2]

Abstract

io_uring is a high-performance asynchronous I/O framework that was introduced in version 5.1 of the Linux kernel. Since its introduction, more than 100 vulnerabilities have been discovered in this subsystem. While there is extensive public exploitation against io_uring bugs in desktop Linux, no public research has targeted it in the Android kernel due to its strict restrictions on memory layout manipulation.

During this talk, we will detail our approach to achieving privilege escalation on Google Pixel 6 and Samsung S22 via CVE-2022-20409, a UAF io_uring bug. Firstly, we will briefly introduce io_uring, the vulnerability, and its memory corruption capabilities. We will then delve into the challenges of exploiting the Android kernel and present our novel exploitation techniques with a detailed step-by-step explanation. We will highlight the effectiveness, generality, and stability of our approach and provide insights into how these techniques work, as well as how to generalize them to exploit other use-after-free (UAF) bugs in both Android and desktop Linux.

Furthermore, we will discuss Samsung's KNOX RKP and showcase a new approach to bypassing its security protection. Through this talk, we hope to provide valuable insights into io_uring exploitation and the challenges associated with exploiting the Android kernel.

1. Introduction

io_uring is a high-performance asynchronous I/O framework in the Linux kernel. It enables efficient, scalable, and low-latency I/O operations for both disk and network I/O, while minimizing context switches and CPU usage. However, despite great kernel I/O optimization, it also introduces a large attack surface that must be considered.

Introduced in Linux Kernel v5.1, io_uring starts with around 4000 lines of code, and now has expanded to over 17000 as of kernel v6.3. From the security perspective, more codes mean more bugs. As syzbot [1] suggests, 161 bugs

[1]{zhenpeng.lin, xinyu.xing}@northwestern.edu Northwestern University
[2]{zhaofeng.chen, kangli}@certik.com Certik

have been found and fixed through Syzkaller. In addition, io_uring is enabled in AOSP by default, which brings the new attack surface to Android as well. Although there has been extensive exploitation of the io_uring subsystem in desktop Linux [**? ? ?**], no public research has targeted the Android kernel due to the unique challenges involved. (the only one is demonstrated by us and demo could be found here [2]).

In this paper, we will show details of how we exploit CVE-2022-20409 (an io_uring bug) on Android to achieve privilege escalation. Specifically, we will first briefly describe the bug detail, the memory corruption capability, and then analyze the challenges of exploiting the Android kernel, and finally present our exploitation strategy and techniques. In addition, we will present a design bypass to Samsung's KNOX. We further generalize the techniques and discuss how they could be applied to exploiting other UAF bugs in the Android kernel.

In summary, this research makes the following contributions: First, we showcase an android kernel exploitation against an io_uring bug, with detailed approaches to tackling different challenges. Second, we present novel exploitation techniques that empower exploiting other Android kernel bugs. Third. our research uncovered a design flaw in Samsung's KNOX RKP and we show how it can be bypassed.

2. CVE-2022-20409

In this section, we briefly describe the root cause and the memory corruption capability of CVE-2022-20409. Readers could jump to Section 2.2 for the discussion of the memory corruption capability of the bug. More details will be presented in the talk if this submission is accepted.

2.1. The root cause

The following layouts the structure of *io_uring_task*, which is an object storing task-specific data for each kernel thread. Normally, the identity field points to the nested structure *__identity*.

```
struct io_uring_task {
    /* submission side */
    struct xarray       xa;
    struct wait_queue_head  wait;
    struct file     *last;
    struct percpu_counter   inflight;
    struct io_identity  __identity;
    struct io_identity  *identity;
    atomic_t        in_idle;
    bool            sqpoll;
};

static inline void io_req_init_async(struct io_kiocb *req)
{
```

```
    struct io_uring_task *tctx = current->io_uring;
    ...
    /* Grab a ref if this isn't our static identity */
    req->work.identity = tctx->identity; [1]
    if (tctx->identity != &tctx->__identity)   [2]
        refcount_inc(&req->work.identity->count); [3]
}

static void io_put_identity(struct io_uring_task *tctx, struct io_kiocb *req)
{
    if (req->work.identity == &tctx->__identity)
        return;
    if (refcount_dec_and_test(&req->work.identity->count)) [4]
        kfree(req->work.identity);                          [5]
}
```

The above shows the function of initializing and freeing the *identity*. Specifically, if the asynchronous request is from the current task, the kernel does nothing. But if it is from a different task, the kernel will increase the reference count, meaning that the identity is allocated from an object. Looking at the *io_put_identity* function, the kernel examines the *identity* with the current task to determine if it belongs to itself. If not, a reference will be decreased.

When there is an inconsistency in calling those two functions, the bug will occur. Assuming we have two tasks: task_a and task_b. In *io_req_init_async*, req belongs to task_a, current is task_b. So, req(task_a)->work.identity will be initialized with task_b- io_uring- identity in [1]. Since tctx->identity by default reference &tctx->__identity, [3] will not be executed.

Later on in *io_put_identity*, the function is executed by task_a, so req(task_a)->work.identity does not equal to &tctx(task_a)->__identity, the function will continue and go to [4]. Because, the reference count is never being increased, this will trigger the free of *identity* in [5], causing an invalid free of the *identity* field of an *io_uring_task* object.

2.2. The memory corruption

Figure 1 reveals the memory layout before and after triggering the free of the *io_uring_task* object. In (a), because of the inconstancy, the identity from another task references the vulnerable io_uring_task without increasing the reference count properly. Therefore, we could free the io_uring_task and the memory layout becomes like (b), where the identity becomes a dangling pointer.

At this point, we could either craft a fake identity object at the freed memory to confuse kernel, or we could trigger the free of the identity object, causing an invalid-free on the kernel heap. Noted that the identity references the middle of a slab object. The invalid-free would result in a memory overlap between two freed objects.

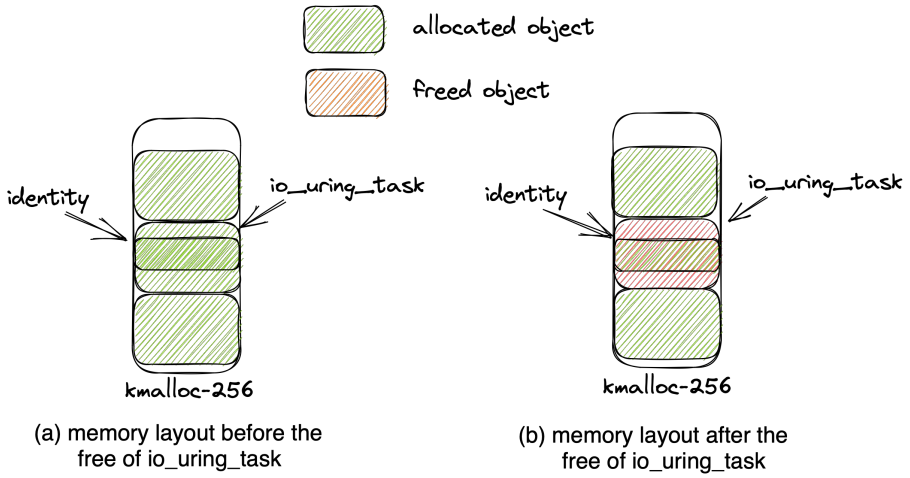

Figure 1: The memory layout before and after triggering the free of io_uring_task

3. The Exploitation Challenges on Android

Normally, exploiting this use-after-free bug in a Linux distro is simple, as there are many available spray objects that facilitate information leaking and control flow hijacking. In the following, we discuss the challenges in detail.

First, compared to Linux distros, Android has fewer functionalities and less available exploit components. For example, the very popular *msg_msg* is not available in the Android kernel. Additionally, many of the components that are available have been hardened with additional security measures, making them more difficult to leverage.

Second, the Android kernel has more mitigation enabled. One example is Kernel Control-Flow Integrity (KCFI), which verifies the control-flow of the kernel at runtime to detect and prevent exploits that manipulate the kernel's execution flow. While in Linux distros, this is not enabled.

Third, the Android operating system uses a permission-based model to control access to system resources, including the kernel. This means more restricted access to the kernel. One example is that the user namespace is not available in Android but is accessible in Linux distros.

Overall, the combination of strict access control, mitigation techniques, and limited functionalities makes it harder to exploit the Android kernel.

4. Jump out of the slab – Page Spray

Most kernel exploitation strategies leverage spray objects to overwrite kernel memory. For example, for use-after-free memory corruption, spraying data objects whose content is fully controlled by attackers could overwrite the data in the UAF object. This technique is known as heap spray. Some elastic objects

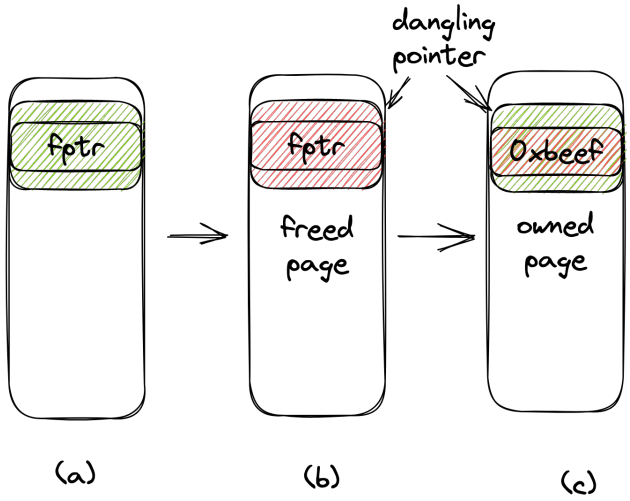

Figure 2: The memory layout before and after triggering the free of io_uring_task

are used very often due to their "clean and stable" memory allocation capability. However, object spray is not always available, it requires the kernel to enable some specific subsystem. When we first tried to find a spray object to manipulate the memory layout for the bug in Android, we failed because we did not find a suitable object. Either due to restricted access, or the functionalities is just not available.

In the end, we tackle this challenge from a different perspective – instead of spraying objects, we overwrite the target memory by spraying the memory pages. The page is the basic unit of memory management in the Linux kernel. The slab, which is known as the allocator for kmalloc, is built on top of it. By overwriting the memory page of the object in the slab, we could also overwrite the memory of the UAF object.

4.1. The Page Spray Technique

As is discussed in AUTOSLAB [3], the memory page of a slab will be recycled to the buddy allocator when all the objects in the slab are freed. Therefore, by reclaiming the freed page, we could overwrite the data to craft a fake object to the dangling pointer.

Figure 2 demonstrates this idea. Initially, in (a), the UAF object contains a function $fptr$, a PC control could be obtained by overwriting it. To do so, we could trigger the vulnerability, free the UAF object, and then free all the other objects on the same slab. As is shown in (b), the page for the slab eventually gets freed, the dangling pointer refers to the freed page. Next, we reclaim the freed page by spraying kernel pages, which contain crafted data and tamper the function pointer.

As we will describe below, there is no limitation for spraying pages in the Linux kernel. Therefore, the technique gives us a reliable way to craft any data for the *io_identity* object, and eventually contributes to the success of the exploitation.

4.2. Page Candidates

The page spray technique requires allocating pages that are shared with the slab allocator. Through manual analysis, we found that there are some APIs accessible in the Android kernel without the limitation of allocation, as such, we could implement the page spray.

```
static void *io_mem_alloc(size_t size)
{
  gfp_t gfp_flags = GFP_KERNEL | __GFP_ZERO | __GFP_NOWARN |
                                  __GFP_COMP | __GFP_NORETRY;

  return (void *) __get_free_pages(gfp_flags, get_order(size));
}
```

One API function we manually found is in the *io_uring* subsystem. The *io_uring* has a ring buffer shared between userspace and kernel space. The ring buffer is allocated through the buddy allocator to obtain kernel pages (function *io_mem_alloc* above). As a shared memory, userspace could directly modify the page content, this means of the page is overlapped with the kernel object, we could tamper the kernel object directly from the userspace. In addition, users can control the size of the page to reclaim the freed slab page as long as the size does not exceed the system memory limit.

```
static ssize_t
pipe_write(struct kiocb *iocb, struct iov_iter *from)
{
  ...
  if (!page) {
    page = alloc_page(GFP_HIGHUSER | __GFP_ACCOUNT);
    if (unlikely(!page)) {
      ret = ret ? : -ENOMEM;
      break;
    }
    pipe->tmp_page = page;
    }
  ...
}
```

Another API function we found is in the *pipe* subsystem. The page allocated is to hold to buffer written to the pipe. By writing crafted data to the pipe, we could control the content stored on the memory page. Besides, the page could

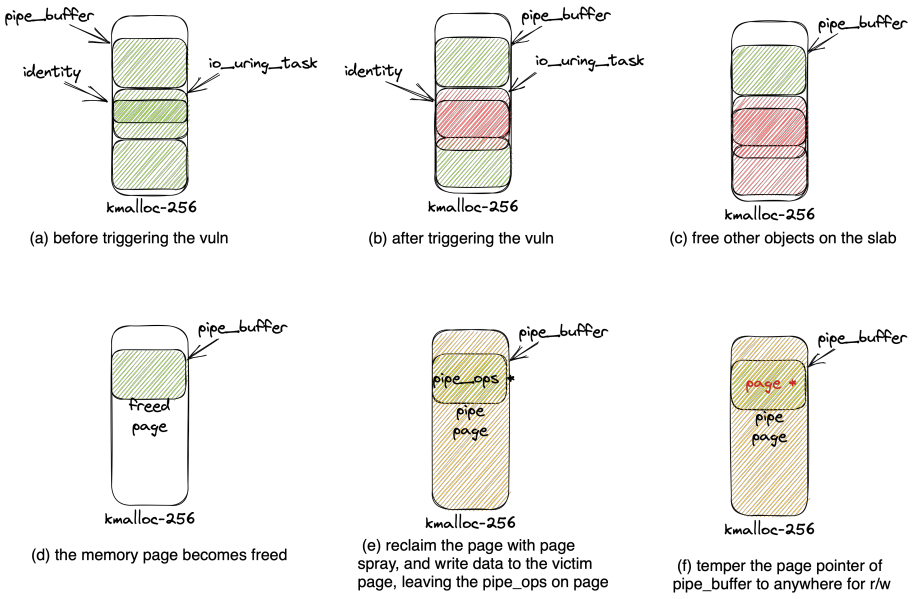

Figure 3: The high-level exploitation flow

be freed by draining the buffer in the pipe. Compared to the page allocation in *io_uring*, the allocation here is more restricted – we cannot control the size of the page. In addition, we cannot modify the data on the page from userspace directly. Modifying data can only be done through reading and writing to the pipe.

5. Achieving Arbitrary Read/Write

With the page spray technique, we could tamper the UAF object with anything we want. However, the UAF object itself contains nothing that can be sent to userspace. Additionally, interacting with it will dereference a series of pointers. Without an information leak, we could not leverage any memory corruption to achieve the goal. In this section, we detail how to address the challenge to achieve the arbitrary read and write capability in Android.

To address the challenge, one thing we could do is to pivot the vulnerability capability. Originally, the UAF object is the $io_identity$ which is hard to leverage due to its nature. By pivoting the vulnerability capability, we could make another object in the same slab as the vulnerable object. Step (a) - (c) from Figure 3 depict the process of such a capability pivot.

In step (a), we allocate an object called $pipe\_buffer$ in the same slab of the UAF object, the figure shows the memory layout of the slab. Next, in step (b), we trigger the vulnerability, which will free the io_uring_task and the UAF object *identity*. This triggering will make the allocator think two objects on

the slab are freed. Later on, we free the other objects on the slab and keep the *pipe_buffer* intact (step c). Since the whole slab is freed, the memory page for the page will be marked as freed as well. As such, the pointer to the *pipe_buffer* becomes a dangling pointer to a freed memory, as is shown is step (d). With the help of the page spray technique, now we could reclaim the freed slab page and overwrite the *pipe_buffer* object in step (e).

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

5.1. Why pipe_buffer

The code above is the definition of the *pipe_buffer* object, which contains a *page* field that indicates the page where the buffer for the pipe is stored, as well as a function table pointer *ops*, which depicts the API functions the pipe is used. Those two fields can be leveraged to leak kernel information and achieve arbitrary read and write. When the users write data to the pipe, the pipe subsystem will check if the *pipe_buffer* is initialized or not. If not, the *pipe_buffer* will be filled with a *pipe_ops*, which is a global function table pointer indicating the operators of the pipe. Since we have reclaimed the page through the page spray technique in the userspace, we now have read and write capability to the page. By reading the content from the page, we could leak the pointer, thus bypassing KASLR (step (e)). Additionally, the page pointer is the destination that the pipe will read and write, by tampering the page pointer to anywhere, we could achieve arbitrary read and write by simply reading and writing to the pipe, as is shown in step (f).

With the function pointer leaked and arbitrary read and write capability on hand, there is not more mitigation we need to bypass in AOSP. Therefore, to exploit Google's Pixel, we traverse the kernel task finding our current process and cred, and then change the uid to escalate our privilege. We could also disable SELinux by simply overwriting the SELinux state.

6. Bypassing Samsung's KNOX RPK

Samsung's KNOX provides additional protection on the Android kernel. Utilizing a security monitor, KNOX protects some kernel properties from being tampered. For example, the cred is protected. So, even with an arbitrary write capability, we could not directly overwrite the uid to escalate privilege. Most of the previous works bypass this by hijacking the kernel control flow to do ROP [4], this is inefficient when generalizing the exploit to different devices and could potentially be killed with enhanced CFI protection. In this white paper, we reveal a weak spot of the KNOX's design and demonstrate a complete bypass

against it. More details will be shared through the talk if the submission is accepted.

6.1. Samsumg's KNOX RPK Design

KNOX protects critical properties in read-only memory. Separating from normal memory space, the protected read-only memory is in a dedicated memory space. When the kernel wants to modify the read-only memory, the kernel communicates to the hypervisor, changing the memory permission to writable, then makes changes to the protected memory. After the modification is done, the kernel signals the hypervisor to change the permission back. In this way, attackers could not modify the cred directly.

Since our goal is to escalate privilege, we need to overwrite the cred object. To prevent attackers from forging cred, KNOX validates the cred through LSM hooks. The function *security_integrity_current* below shows how this is is done. Specifically, it checks if the cred is in the protected memory region, and if the mapping dependency is correct. If anything is broken or does not match the record, the kernel will panic to prevent attacks.

```
int security_integrity_current(void)
{
  const struct cred *cur_cred = current_cred();

  rcu_read_lock();
  if (kdp_enable &&
    (is_kdp_invalid_cred_sp((u64)cur_cred, (u64)cur_cred->security)
    || cmp_sec_integrity(cur_cred, current->mm)
#ifdef CONFIG_KDP_NS
    || cmp_ns_integrity()
#endif
    )) {
      rcu_read_unlock();
      panic("KDP CRED PROTECTION VIOLATION\n");
    }
    rcu_read_unlock();
    return 0;
}

static inline bool is_kdp_invalid_cred_sp(u64 cred, u64 sec_ptr)
{
  struct task_security_struct *tsec = (struct task_security_struct *)sec_ptr;
  ...
  if (!is_kdp_protect_addr(cred) ||
    !is_kdp_protect_addr(cred + cred_size) ||
    !is_kdp_protect_addr(sec_ptr) ||
    !is_kdp_protect_addr(sec_ptr + tsec_size)) {
    printk(KERN_ERR, "[KDP] cred: %d, cred + sizeof(cred): %d, sp: %d, sp + sizeof(tsec): %d
```

```
    is_kdp_protect_addr(cred),
    is_kdp_protect_addr(cred + cred_size),
    is_kdp_protect_addr(sec_ptr),
    is_kdp_protect_addr(sec_ptr + tsec_size));
      return true;
    }

    if ((u64)tsec->bp_cred != cred) {
      printk(KERN_ERR, "[KDP] %s: tesc->bp_cred: %lx, cred: %lx\n",
      __func__, (u64)tsec->bp_cred, cred);
      return true;
    }

    return false;
}
int is_kdp_protect_addr(unsigned long addr)
{
  ...
  if ((addr == ((unsigned long)&init_cred)) ||
  (addr == ((unsigned long)&init_sec)))
    return PROTECT_INIT;

    page = virt_to_head_page(objp);
    s = page->slab_cache;
    if (s && (s == cred_jar_ro || s == tsec_jar))
      return PROTECT_KMEM;
    return 0;
}
```

However, the way KNOX validates cred is vulnerable, so the cred is possible to be forged with sophisticated techniques. The function *is_kdp_protect_addr* shows how KNOX validates if the memory belongs to the protected region. Specifically, it checks if the memory object is coming from the slab *cred_jar_ro* and *tsec_jar*. Because the page of the object is not protected, meaning that we could tamper the metadata of the page, we could simply change the slab cache of it to either *cred_jar_ro* or *tsec_jar*, then we would be able to bypass the check of the function and forge a fake privileged cred for our process. With this, we can bypass KNOX without having to hijack the control flow. A more detailed analysis of Samsung's KNOX will be presented in the talk.

References

[1] syzbot, `https://syzkaller.appspot.com/upstream`.

[2] Pixel 6 exploitation demo, `https://zplin.me/pixel6_demo.mp4`.

[3] How autoslab changes the memory unsafety game, https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game.

[4] Defeating samsung knox with zero privilege, https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege-wp.pdf.