Racing Against the Lock: Exploiting Spinlock UAF in the Android Kernel

Moshe Kol
The JSOF Research Lab

Abstract

This paper presents an exploit for a unique Binder kernel use-after-free (UAF) vulnerability which was disclosed recently (CVE-2022-20421). Through this vulnerability, we examine the exploitability of a spinlock use-after-free, containing no other memory corruption primitive. We devised an innovative and generic technique for exploiting such limited use-after-free vulnerabilities, assuming a queued spinlock implementation (the default implementation on Android since kernel version 4.19).

Our technique includes constructing a primitive to corrupt a kernel pointer. This corruption is then further developed into a type confusion and eventually, arbitrary kernel read/write, including kASLR bypass and all other relevant mitigations. We successfully demonstrated a robust and stable exploitation on 3 Android devices (Samsung Galaxy S21 Ultra, Samsung Galaxy S22, and Google Pixel 6), assuming code execution from the `untrusted_app` SELinux context.

1 Introduction

Nowadays, modern Android smartphones are shipped with the strongest kernel mitigations and security measures. These measures reduce the likelihood of a successful and reliable local privilege escalation (LPE) exploit involving a kernel vulnerability. To cope with that, attackers typically chain multiple vulnerabilities or rely on a strong one. Specifically, kernel use-after-free (UAF) bugs continue to be one of the most exploited bugs in modern Linux/Android systems, despite the wide range of security mitigations deployed on such systems.

Typically, a use-after-free bug contains an explicit memory corruption primitive that is then used to construct a full local privilege escalation (LPE) exploit. Depending on the vulnerability, the attacker, who wishes to utilize the memory corruption primitive, has to bypass any code flows that access the reused memory in ways that hinder successful exploitation. For instance, attackers usually bypass kernel synchronization primitives, such as spinlocks or mutexes, since they have the potential to detain execution of the intended code flow (best case) or crash the kernel in the worst case.

A primary example can be seen in the exploit for the Bad Binder (CVE-2019-2215) vulnerability [7]. There, the exploit ensures `iovec[10].iov_base` is set to a value that appears as an unlocked spinlock. The spinlock is bypassed to reach the interesting function `__remove_wait_queue()`, that is used further in the exploit. Another example is the exploit for `xt_qtaguid` vulnerability (CVE-2021-0399) [4]. The exploit ensures that `eventfd_ctx.spinlock` and `seq_file.mutex.spinlock` overlap to prevent crashing the kernel and utilize the `eventfd` object for subsequent exploit operations.

In this work, we consider the exploitability of a spinlock use-after-free, containing *no other memory corruption primitive*. We encountered this unusual situation while developing an exploit for a Binder kernel use-after-free vulnerability which was disclosed recently (CVE-2022-20421). During development, we constructed an innovative and generic technique for exploiting such limited use-after-free vulnerabilities, assuming a queued spinlock implementation (Android's default implementation since kernel 4.19).

Specifically, we constructed a primitive that can zero-out the 2 least significant bytes of a kernel pointer, which we develop in to a type confusion. We then exploit this type confusion to bypass kASLR and all other relevant mitigations and gain arbitrary kernel read/write capabilities. Using our technique, we successfully exploited the Binder vulnerability on 3 Android devices (Samsung Galaxy S21 Ultra, Samsung Galaxy S22 and Google Pixel 6), assuming code execution from the `untrusted_app` SELinux context. Our exploit succeeds 4 out of 5 times, depending on the device and any background activity.

Our contributions in this paper are:

1. Full description of our new spinlock UAF exploitation technique.

2. Root cause analysis of the Binder vulnerability, including techniques to widen the race window.

3. Source code[1] of an exploit that uses our technique and obtains arbitrary R/W primitives on 3 Android devices (Samsung Galaxy S21 Ultra, Samsung Galaxy S22 and Google Pixel 6) when running from `untrusted_app` SELinux context.

2 Background

2.1 Binder Overview

Processes in the Android operating system use the *Binder framework* for inter-process communication (IPC). At the heart of the Binder framework lies the Binder kernel-mode driver, which facilitates message passing from one process to another. Processes interact with the driver by opening the */dev/binder* device (or */dev/hwbinder*, depending on the context) and issuing commands to it using the `ioctl()` system call.

Binder is a central piece in the Android operating system, both in terms of security and performance. From the security perspective, it ensures (with the help of the kernel driver) that the identity of the source process (UID, PID) is securely conveyed to the destination. The destination process can then use this identity for permission checking at the platform level. Performance-wise, Binder supports multi-threading, lazy memory allocation and scatter-gather features.

In binder, the messages transmitted between processes are called *transactions*. For a process to receive transactions, it must designate some portion of its address space for their storage, up to 4MB (this is done via the `mmap()` system call on the binder device). This virtual memory area is managed by the kernel driver and is marked read-only.

When a new transaction is issued, the kernel driver allocates space within the destination process' memory area and copies the transaction data to it. The driver then selects a thread in the destination process to receive the transaction (these threads are called *loopers* in binder parlance) and inserts the transaction to the selected thread's work queue. The selected thread then reads the transaction and lets user-space handle it. When user-space is done with the transaction, it issues an `ioctl()` to free-up the space allocated for the transaction, allowing future transactions to re-use it.

The Binder device (either */dev/binder* or */dev/hwbinder*) is accessible by all applications running under the `untrusted_app` SELinux domain. Hence, it is a very attractive attack surface for privilege escalation. Our focus is on the kernel-mode driver, not the user-mode components of the framework. Familiarity with Linux kernel concepts is assumed.

2.2 Binder Device Lifecycle

Open. When a process wants to talk using Binder, it needs to open the Binder device:

```
binder_fd = open("/dev/binder", O_RDONLY);
```

Under the hood, in the kernel, the function `binder_open()` is invoked. This function begins with allocating (with `kzalloc()`) a structure called `binder_proc`. After initializing it, the structure is linked with the open file descriptor using the `file`'s `private_data` field.

The `binder_proc` structure contains the following fields (among others):

`threads`: A red-black tree of `binder_thread` objects belonging to the process.
`nodes`: A red-black tree of `binder_node` objects belonging to the process.
`refs_by_desc, refs_by_node`: Red-black trees storing `binder_ref` objects, sorted by different keys.
`tmp_ref`: Refcount. When it reaches 0, the structure is freed.
`inner_lock, outer_lock`: Spinlocks, for thread-safety.

Close. When the user-mode done with Binder's service, it closes its binder file descriptor:

```
close(binder_fd);
```

In the kernel, the function `binder_release()` is invoked. `binder_release()` does not immediately cleans-up the `binder_proc` structure. Instead, it uses a deferred work mechanism for that. The actual clean-up code is in `binder_deferred_release(proc)`. This function:

1. Removes `proc` from the system-wide list of `binder_procs`.
2. Increments `proc`'s refcount and sets its `is_dead` flag to `true`.
3. Releases all `binder_threads` associated with `proc`. Done by calling `binder_thread_release(proc, thread)` on each thread.
4. Releases all `binder_nodes` owned by the process. Done by calling `binder_node_release(node)` on each node.
5. Releases all `binder_refs` owned by the process. Done by calling

[1] https://github.com/0xkol/badspin

`binder_cleanup_ref_olocked(ref)` and `binder_free_ref(ref)` on each reference.

6. Releases all work items:

```
binder_release_work(proc, &proc->todo);
binder_release_work(proc,
     &proc->delivered_death);
```

7. Finally, decrement the `proc`'s refcount and free it if it reaches 0. This is done by calling the function `binder_proc_dec_tmpref(proc)`.

2.3 Binder Transactions

Recall that the messages exchanged by Binder are called transactions. Transactions can optionally contain *objects*, alongside raw data. Binder supports 7 types of objects, identified by the constants `BINDER_TYPE_*` (see *include/uapi/linux/android/binder.h*).

There are objects to support the submission of file-descriptors between processes (`FD`, `FDA`), an object to support scatter-gather functionality (`PTR`), and, most importantly, objects that support the binder framework itself (`BINDER`, `HANDLE` and their "weak" variants).

When the kernel driver handles the transaction, it processes each embedded object and performs all the necessary operations to convey it to the other end. This is known as *object translation*. Each object type mandates different operations to be taken for its translation. For example, say that process $\mathcal{A}$ wants to pass file descriptor 10 to process $\mathcal{B}$. To do that, process $\mathcal{A}$ prepares a new transaction containing an object of type `FD` specifying fd number 10. During transaction processing, the kernel will allocate a new unused fd in process $\mathcal{B}$ (say 54) and will link the new fd to the same underlying `file` as that of process $\mathcal{A}$. When process $\mathcal{B}$ reads the transaction, it will see an object of type `FD` consisting of fd number 54, ready for further interaction. Notice how the driver changes details of the object being translated (in this case, the fd number), so that they will make sense when they are read by the other side.

Object translation might fail for numerous reasons, including insufficient permissions (SELinux checks), incorrect object's specification by the user or internal errors (e.g. failure to allocate memory or some other logic discrepancy). When some object fails to be translated, the entire transaction is considered erroneous and not delivered to the other side. In the error code flow, the driver cleans up the transaction buffer, effectively undoing the translation of the objects that were translated successfully. The offending object, however, is not cleaned-up. This makes sense, as the function responsible for the translation should clean-up for itself if it returns with an error (the outer code flow does not know which step in the translation failed).

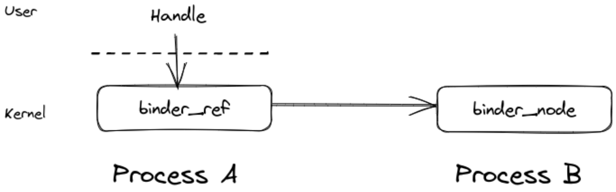


Figure 1: Binder objects and references.

2.4 Binder Objects and References

To talk with a process, the sending process must have a *reference* to a *Binder object* owned by the destination process. (When we say "Binder object", we mean an object of type "Binder".) In this manner, the Binder object acts as an endpoint for Binder transactions.

Binder objects are identified differently depending on the process that refers them. From the perspective of the process that owns the Binder object, they are identified using an 8-byte value called `ptr`, which is opaque from the kernel perspective. We will call Binder objects identified in this manner *local Binder objects*. A local Binder object is represented using a `binder_node` structure in kernel-land.

Other processes refer to the destination's Binder object using a 32-bit value called *handle*. When identified in this manner, they are referred to as *remote Binder objects*. Internally, the handle value is mapped to a kernel object called `binder_ref` that in turn references the node. See Figure 1.

Binder objects are unique and unforgeable. The kernel ensures their uniqueness by maintaining their identity. In particular, the kernel ensures that only one local Binder object with a given `ptr` exists per process. In addition, the kernel ensures that a process can have only one reference for a particular Binder object.

A process can *communicate* a Binder object to another process, thereby giving the other process access to the object. The communication is done by embedding special objects within a Binder transaction. The steps taken in this kind of communication change depending on the way the Binder object is identified (local or remote), but the result is the same: A remote Binder object is created on the other side to refer to the Binder object being passed. It is impossible to forge a reference to a Binder object without having other process passing it.

Passing a remote Binder object. As already discussed, a remote Binder object is identified using a handle value. To pass a remote Binder object to another process, a new transaction has to be prepared, containing an object of type HANDLE with the relevant handle value specified.

When the transaction is processed by the kernel and the object is being translated, a new Binder reference (binder_ref) in the destination process is being created, if one not already exists in the destination for this particular Binder object. During the creation of the new Binder reference, a new handle value is being allocated for it (to let user-space refer to it). When the destination process reads the transaction, it sees an object of type HANDLE with the *new* handle value just allocated. See Figure 2. (If a Binder reference already exists for the Binder object being handed over, the kernel just uses the handle value of the existing Binder reference.)

Passing a local Binder object. A local Binder object is identified using a ptr value. To pass a local Binder object, an object of type BINDER is embedded in a transaction, containing the ptr value of the Binder object being passed.

During transaction processing, a new Binder node (binder_node) is created in the sending process (if not already exists) and a new reference (binder_ref) is created for it in the receiving process (if not already exists). The kernel driver also rewrites the transaction's object type from BINDER to HANDLE, specifying the handle value to use when referring to this Binder object.

Bootstrapping. As Binder objects are handed over using transactions, and transactions are sent to Binder objects, this begs the question: How communication is bootstrapped between processes?

In Binder, a special process is designated to be the *context manager*. This process owns a special Binder object, accessible by all processes using the handle value 0. To bootstrap communication, processes *register* with the context manager. In the registration message, processes state their name and pass a BINDER object. A new reference (and handle) is created in the context manager, which maintains a mapping between names and handles. Other processes can then *lookup* a particular name with the context manager. If the name exists and a permission check passes, the context manager will transfer the handle stored by it back to the requester (by embedding an object of type HANDLE in the reply transaction). Through the newly received reference, the requester can interact with the original process (i.e. sending transactions to it).

2.4.1 Strong and Weak References

There are two types of references, strong and weak. A *strong reference* to a Binder object is needed to send transactions to it. Strong references are passed using the object types BINDER (local) and HANDLE (remote).

A *weak reference* is used in situations where there is no need to send transactions. For instance, it is often used for receiving death notifications (see below). Weak references are passed using the object types WEAK_BINDER (local) and WEAK_HANDLE (remote).

A strong reference can be demoted to a weak reference. However, a weak reference cannot be promoted to a strong one, unless a strong reference is received from a peer. (The BC_ATTEMPT_ACQUIRE command supports such promotion, but it is not implemented in Android currently.)

2.4.2 Refcounting

Generally speaking, if an entity references some object, that object counts how many references it got. When no references exists, the object may be released. This is called refcounting, and it is a commonly used technique to handle automatic object de-allocation in the Android platform (sp<>, wp<>).

Since in Binder we have references to Binder objects that live in a different process, the concept of reference counting is expanded across process boundaries:

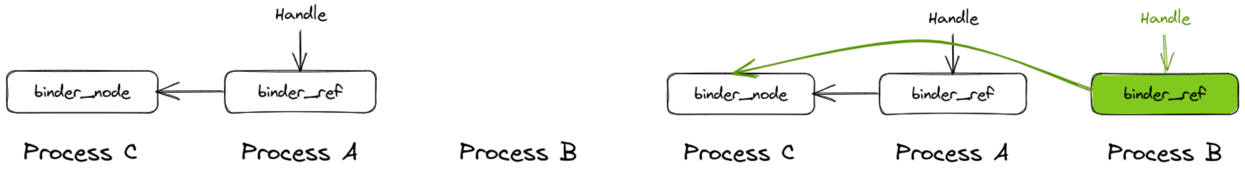
1. User-space object references a binder_ref kernel object. User-space informs about refcounting changes via the commands: BC_ACQUIRE and BC_RELEASE for strong increment or decrement, and BC_INCREFS and BC_DECREFS for weak increment or decrement. These strong/weak reference counts are stored in the binder_ref structure.

2. binder_ref references a binder_node kernel object. The reference count is maintained by the kernel and called "internal" by the driver code. The internal strong reference count is maintained in binder_node's field internal_strong_refs. The internal weak reference count is implicitly tracked via a list tracking all of the binder_ref objects referencing this node.

3. binder_node references an object in user-space. To inform about changes in refcounting, the kernel return commands for user-space to perform: BR_INCREFS, BR_ACQUIRE, BR_DECREFS and BR_RELEASE. User-space acknowledges the change using BC_INCREFS_DONE and BC_ACQUIRE_DONE. These reference counts are partly tracked by the kernel via the fields local_[strong|weak]_refs of binder_node.

2.4.3 Death Notifications

Death notifications is a unique Binder feature, allowing a process to be notified when a Binder object dies. This is useful for de-allocating resources associated with the remote Binder object and used throughout the Android platform.



(a) Before

(b) After

Figure 2: Passing remote Binder object from $\mathcal{A}$ to $\mathcal{B}$.

3 The Vulnerability

The Binder vulnerability (CVE-2022-20421) is a use-after-free vulnerability on `binder_proc` object. It is triggered by sending a transaction with an object of type `HANDLE` or `WEAK_HANDLE`, and having the target process die in-parallel while the transaction is in-flight. This results in a new `binder_ref` object that is created for the (dying) target process, which is not cleaned up by the driver. After the target's `binder_proc` object is released, the newly created `binder_ref` is left with a kernel pointer pointing to the freed `binder_proc`. Later, when the `binder_node` referenced by the moot `binder_ref` is released, the freed `binder_proc` is locked using `spin_lock()`, resulting in a use-after-free.

In this section, we thoroughly describe all of the steps needed to trigger the vulnerability.

3.1 Setup

To trigger the vulnerability we need to setup binder IPC between 3 processes, named $\mathcal{A}$, $\mathcal{B}$ and C, all of which under our control. To do that, we abuse the *ITokenManager* service that is reachable from the */dev/hwbinder* context. This is a known technique and it is documented in [1].

3.2 Creating a Reference in a Dying Process

In this step, we will have process $\mathcal{B}$ receive a new reference (`binder_ref`) after it closes its binder file descriptor; i.e. after a cleanup procedure has been done to its internal binder structures. As we will see below, this is not enough to trigger the vulnerability, but it is a crucial step.

To create new binder references in process $\mathcal{B}$, a transaction containing special objects must be sent to it. As discussed in § 2, a new Binder reference can be created in two ways: by passing a local Binder object or a remote Binder object. The vulnerability can be triggered only by sending a remote Binder object, using the object types `HANDLE` and `WEAK_HANDLE`. In this section, we will concentrate on triggering it with `WEAK_HANDLE`. See Appendix B on how the vulnerability could be triggered with `HANDLE`.

Since new references can be created only as a result of a transaction, and we cannot send a new transaction to a process that has already died, our objective seems impossible. This is where we exploit a race condition in the driver. We will have process $\mathcal{A}$ sending process $\mathcal{B}$ a transaction containing a `WEAK_HANDLE` object, and we will arrange process $\mathcal{B}$ to close its Binder file descriptor at the same time. Since process $\mathcal{A}$ already begun processing the transaction, thinking $\mathcal{B}$ is alive, it will not be aware that $\mathcal{B}$ has died in the meanwhile. It will recognize that $\mathcal{B}$ has died only at the very last step – when it will try to place the new transaction on the work queue of one of $\mathcal{B}$'s Binder threads. At this point, though, all of the objects carried by the transaction were already translated, which is enough for our purposes.

The race condition window is fairly large. It begins with the calls:

```
ref = binder_get_ref_olocked(proc,
        tr->target.handle, true); ❶
if (ref)
    target_node = binder_get_node_refs_for_txn(
                    ref->node, &target_proc,
                    &return_error); ❷
```

In ❶, the handle value corresponding to $\mathcal{B}$'s node (the target of the transaction) is looked-up in $\mathcal{A}$'s data structures. If found, it returns a `binder_ref` structure. In ❷, $\mathcal{B}$'s `binder_proc` is verified to be alive (indirectly by testing if `node->proc` is non-NULL). If so, a refcount is taken on the target node and $\mathcal{B}$'s `binder_proc` to ensure they are no freed for the duration of the transaction. The race window ends by the time the `WEAK_HANDLE` object is translated by the driver. So, to win the race, we must have process $\mathcal{B}$ alive before the call to ❷, and it must be already dead (i.e. `binder_deferred_release()` completed) by the time the `WEAK_HANDLE` object is translated. Appendix C describes how we managed to widen the race window so we win the race every time.

To sum it up: If this race is won, process $\mathcal{A}$ will reach the code that translates binder objects *after* `binder_deferred_release()` was completed for process $\mathcal{B}$. The transaction carries an object of type WEAK_HANDLE, so a new Binder reference will be created in the context of process $\mathcal{B}$ after it died.

3.3 Causing Translation Error

We have not arrived at a use-after-free condition yet. Closing the Binder file descriptor does not necessarily free the internal `binder_proc` structure, as it is reference counted. When process $\mathcal{A}$ sends a transaction to process $\mathcal{B}$, it checks that it is alive and increment the reference count on $\mathcal{B}$'s `binder_proc`. When process $\mathcal{A}$ done with the transaction, success flow or error flow, it will drop this reference, potentially causing the `binder_proc` structure to be `kfree()`'d.

If we won the race described above, we are guaranteed to reach the error flow of the transaction processing. Before dropping the last reference to $\mathcal{B}$'s `binder_proc`, the error flow ensures to cleanup the artifacts of the translated objects – essentially undoing the effect of each translation made. In this cleanup procedure, performed in `binder_transaction_buffer_release()`, the newly created reference in $\mathcal{B}$ (as a result of the WEAK_HANDLE translation) is deleted. This is perfectly fine, so no memory corruption, yet.

The problem arises if a translation error occurs during the translation of the weak handle object, i.e. if `binder_translate_handle()` returns with error. In such a case, the objects that will be cleaned-up in the error flow will be the the objects that were successfully translated *so far*. So, the weak handle object will not be cleaned-up in the error flow.

On top of that, `binder_translate_handle()` function itself fails to clean-up the reference if it returns with an error in a certain condition. The lack of clean-up by the function `binder_translate_handle()` is exactly the root cause of the vulnerability. Combining this fact with the race condition, we can create a new Binder reference that will never be cleaned-up by the driver. This is bad news, as will be explained in the next section § 3.4.

Weak handle's translation error. To better understand how a weak handle's translation can result in an error and lack of clean-up, we need to go through the steps taken when translating remote Binder objects:

1. The handle value specified by the sender (process $\mathcal{A}$) is mapped to a Binder reference (`binder_ref`), from which a pointer to the Binder object's `binder_node` structure is obtained.

2. If the transaction's target process (process $\mathcal{B}$) differs from the owner of the Binder object (process $\mathcal{C}$):

    (a) A reference to the Binder object being communicated is looked-up in the target process (process $\mathcal{B}$ in our case). If one is not already present, it is allocated and inserted to the relevant data structures of the target process.

    (b) A reference count is taken on the `binder_ref`, whether it was newly created or not.

If the last step (taking a reference count) resulted in an error, the whole translation is considered unsuccessful, and the newly created `binder_ref` is never cleaned-up. It remains to show how the last step can fail, and how this failure could be triggered.

The last step is performed by the function `binder_inc_ref_for_node()`, invoked with a pointer to the newly created `binder_ref`. In `binder_inc_ref_olocked()`, a weak reference is taken on the new reference. Since it is a *new* reference, a weak reference is also taken on the node it points to. This is where the error condition will be injected:

```
static int binder_inc_node_nilocked(struct
     binder_node *node, int strong,
                     int internal,
                     struct list_head *target_list)
{
    struct binder_proc *proc = node->proc;

    if (strong) {
        ...
    } else {
        if (!internal)
            node->local_weak_refs++;
        if (!node->has_weak_ref ❶ &&
            list_empty(&node->work.entry) ❷) {
            if (target_list == NULL) {
                return -EINVAL; ❸
            }

            binder_enqueue_work_ilocked(&node->work,
      target_list);
        }
    }
    return 0;
}
```

In our flow, `strong` is false and `target_list` is NULL. So for this function to return -EINVAL error ❸, we must have `node->has_weak_ref == 0` ❶ and an empty node's work list ❷. This can be done *synchronously* when passing weak Binder references, and this is where a third process, process $\mathcal{C}$, comes in.

Creating the faulty Binder reference. Before the race condition is triggered (by processes $\mathcal{A}$ and $\mathcal{B}$), process $\mathcal{A}$ will ensure that it has a Binder reference that can cause this error flow. The faulty Binder reference will point to a node in process $\mathcal{C}$. The following steps are taken by process $\mathcal{C}$ to make the Binder reference faulty:

1. $\mathcal{C}$ spawns a new thread, call it $\mathcal{T}$.

2. $\mathcal{C}$ sends $\mathcal{A}$ a transaction containing an object of type `BINDER` (strong local Binder object). When this object is translated, a new `binder_node` is created in $\mathcal{C}$ and a new `binder_ref` is created in $\mathcal{A}$.

3. Process $\mathcal{A}$ accepts the new reference and take a strong reference on it (to keep it alive).

4. At this point, the `has_weak_ref` field of the new `binder_node` equals 0. However, as part of its creation process, the new `binder_node` created in $\mathcal{C}$ is inserted to a work list of the binder thread that created it, namely $\mathcal{T}$. When $\mathcal{T}$ processes its work list (by invoking `BINDER_READ_WRITE` ioctl with a read buffer), it sets `node->has_weak_ref` to 1. We wish to avoid that and also remove the node from any list. To do this, $\mathcal{T}$ invokes an `ioctl` command `BINDER_THREAD_EXIT`. This ioctl causes the work list of $\mathcal{T}$ to be released, deleting the node from it without ever setting `has_weak_ref`. Exactly what we wanted.

Wrap up. The communication between processes $\mathcal{A}$ and $\mathcal{C}$ results in a faulty Binder reference created in $\mathcal{A}$. This faulty reference is then passed to process $\mathcal{B}$, which, at the same time, closes its file descriptor. If the race condition is won, a new Binder reference (`binder_ref`) referring to $\mathcal{C}$'s node will be inserted to process $\mathcal{B}$, after its death (after `binder_deferred_release()` completed). Further, this reference is never cleaned-up by the driver.

3.4 The use-after-free

The `binder_ref` structure contains a field called `proc` that points to the owing `binder_proc`. In our case, the stale Binder reference is owned by process $\mathcal{B}$, so `ref->proc` points to $\mathcal{B}$'s `binder_proc`. $\mathcal{B}$'s `binder_proc` gets `kfree()`'d when the last reference count for it is dropped. After `binder_deferred_release()` was completed for $\mathcal{B}$, the last reference count of $\mathcal{B}$'s `binder_proc` is taken by process $\mathcal{A}$ (as it is in a middle of a transaction directed at $\mathcal{B}$). Therefore, when process $\mathcal{A}$ finish processing the transaction to $\mathcal{B}$, it will drop the last reference on $\mathcal{B}$'s `binder_proc` and free it. This means that `ref->proc` contains a stale pointer to a freed location.

It is not a use-after-free vulnerability if `ref->proc` is not used. Indeed, there is one code flow that uses it – it happens when process $\mathcal{C}$ closes its Binder file descriptor (or exits).

When process $\mathcal{C}$ closes its Binder file descriptor, the function `binder_deferred_release()` will be called for it. As discussed in § 2.2, this function releases all of the Binder nodes owned by $\mathcal{C}$, by invoking `binder_node_release()` on each one. During node release, all of the node's references are enumerated when death notifications are handled:

```
static int binder_node_release(struct binder_node *
    node, int refs) {
   ...
   hlist_for_each_entry(ref, &node->refs,
    node_entry) {
       binder_inner_proc_lock(ref->proc); ❶
       if (!ref->death) {
           binder_inner_proc_unlock(ref->proc); ❷
           continue;
       }
       ...
   }
   ...
}
```

We see that the first use-after-free is in ❶, as `ref->proc` points to the already-freed $\mathcal{B}$'s `binder_proc` structure. Note that the stale reference cannot possibly have a non-NULL `ref->death`, since its insertion was racy with its owner death.

The "inner lock" of `binder_proc` is implemented as a spinlock, so the use-after-free is only ❶:

```
spin_lock(&ref->proc->inner_lock);
```

... followed by an immediate ❷:

```
spin_unlock(&ref->proc->inner_lock);
```

3.5 CVE-2022-20421

CVE-2022-20421 is a race condition vulnerability in the Binder kernel driver that results in a use-after-free on the `binder_proc` object. It is reachable from `untrusted_app` SELinux domain, and affects all Android devices running on kernels 3.18 to 5.10 with security patch level below October 2022.

The vulnerability has few advantages. First, the `binder_proc` structure is allocated in the general-purpose slab `kmalloc-1k`. Compared with other slabs, like `kmalloc-128`, this is a relatively inactive slab, so there is high likelihood to reliably reallocate the memory with an object of interest. Second, we managed to win the race condition 100% of the times (Appendix C), so it is suitable for reliable exploitation. A third useful property of this vulnerability is that the timing of the "use" is under our control, as we control process $\mathcal{C}$. This gives us more time to prepare the reallocating object for the use-after-free.

On the negative side, the primitive seems weak: we can flip a bit from 0 to 1 for very short amount of time. There is no other memory corruption inside the critical section that can benefit us. This is a very unique and challenging situation.

On top of that, the offset of binder_proc's inner_lock field varies between devices. During development we encountered the following offsets: 512 (on an old Samsung Galaxy S21 device), 520 (Samsung Galaxy S20), 544 (Samsung Galaxy S21) and 576 (Samsung Galaxy S22 and Google Pixel 6). Ideally, we want an exploit technique that will work universally, with little or no changes per device.

4 The Primitive

To exploit the use-after-free vulnerability, we will construct useful exploit primitives by taking the following steps:

Free the vulnerable object: The first step is to trigger the vulnerability and free the vulnerable object *while keeping a pointer to it.* In our case, the vulnerable object is binder_proc and the steps taken to trigger the vulnerability were described in § 3.

Reallocate the object: After the vulnerable object has been freed, we reallocate the memory for a different object or data structure. This allows us to create a type confusion vulnerability, in which the software is manipulated to interpret data as a different type.

Exploit the type confusion: Finally, we exploit the type confusion to extract useful exploit primitive. In our case, if we reallocate the memory with an object Foo, that overlaps binder_proc's inner_lock with the field bar, then bar will be interpreted as a spinlock when we trigger the use-after-free. This can lead to serious consequences depending on the original type of bar (pointer, length, refcount, etc).

In this section, we closely examine the spinlock implementation in the Android kernel. To our surprise, the implementation is more complex than what one might think, and we managed to extract powerful memory corruption primitives off of it.

4.1 Spinlocks

Spinlocks are a key synchronization mechanism in the Linux kernel, used to protect shared resources from concurrent access. When a thread needs to access a shared resource, it first acquires the spinlock associated with that resource. If the spinlock is already held by another thread, the requesting thread spins (loops) until the lock is released.

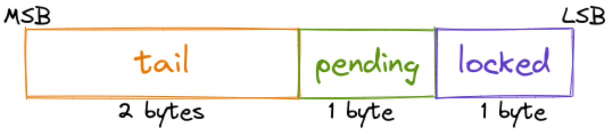


Figure 3: The structure of qspinlock

Spinlocks are typically used in situations where the critical section is short. The reason is that spinlocks implicitly disable CPU preemption, so a long critical section induces more latency on the system. Spinlocks are also used when other types of locks (e.g. sleeping locks) are not suitable (e.g. IRQ handler). See Appendix A for more detailed information about spinlocks.

In some cases, multiple threads may attempt to acquire the same spinlock at the same time. To prevent these threads from starving (spinning indefinitely and wasting CPU cycles) and improve cache utilization, Linux implements queued spinlocks, which are more efficient version of spinlocks compared to traditional implementations.

4.2 Queued Spinlocks

Queued spinlocks are the default implementation in the Android kernel since version 4.19 (controlled by the kernel configuration CONFIG_QUEUED_SPINLOCKS). They work by maintaining a queue of threads that are waiting to acquire the lock. When a thread attempts to acquire a spinlock that is already held by another thread, it is placed at the end of the queue. The thread at the head of the queue is then granted the lock when it becomes available, and all other threads continue to spin until it is their turn to acquire the lock.

The queued spinlock is implemented using the qspinlock structure, which is a 4-byte value that is used to store the lock state. It is broken into 3 sub-fields, as depicted in Figure 3. The lock state can be in one of three states:

Unlocked: The lock is not currently held by any thread. All sub-fields are 0.

Locked: The lock is currently held by a thread. In this case, the locked sub-field is non-zero.

Contended: The lock is currently held by a thread, and there are other threads waiting to acquire the lock. In this case, $(\texttt{tail}, \texttt{pending}) \neq (0, 0)$.

We now look at the implementation in more detail.

4.2.1 Queued Spinlocks Implementation

The generic function spin_lock() begins by disabling preemption on the CPU, and then calling the queued spinlock specific function:

```
void queued_spin_lock(struct qspinlock *lock);
```

Similarly, `spin_unlock()` calls the queued spinlock specific implementation

```
void queued_spin_unlock(struct qspinlock *lock);
```

... and finishes with re-enabling CPU preemption.

The queued spinlock implementation in Linux uses atomic operations and memory barriers to ensure correct behavior. Instead of confusing ourselves with such low-level details, we are going to focus on the core logic of the implementation, with the goal of extracting useful exploitation primitives. The following describes the core logic of the queued spinlock implementation. It is heavily edited from the original and should be considered as pseudo-code:

```
void queued_spin_lock(struct qspinlock *lock) {
    if (*(u32 *)lock == 0) { ❶
        lock->locked = 1;
        return;
    }

    if (lock->tail != 0 || lock->pending != 0) ❷
        goto queue;

    lock->pending = 1; ❸
    while (lock->locked != 0); ❹
    lock->pending = 0;
    lock->locked = 1; ❺
    return;

queue: ...
}

void queued_spin_unlock(struct qspinlock *lock) {
    lock->locked = 0; ❻
}
```

When the lock is in UNLOCKED state ❶, the `locked` byte is set to 1 and the function returns – indicating the lock has been acquired successfully. The thread continues with executing the critical section.

Say that while the critical section is executing and the lock is held, another CPU tries to grab the lock. In this case, the new CPU will see $(\texttt{tail}, \texttt{pending}, \texttt{locked}) = (0, 0, 1)$, so steps ❶ and ❷ will be skipped. The new CPU will raise the `pending` bit in step ❸, and will spin at step ❹ until the `locked` byte becomes 0.

When the first thread finishes its critical section, it will release the lock by calling `spin_unlock()`, which sets the `locked` byte to 0 ❻. At this point, the spinning CPU will see that the `locked` byte became 0 and will quit busy-looping. Before returning, it will clear the `pending` bit and set the `locked` byte to 1 ❺, indicating lock ownership.

Note that if a CPU tries to grab the lock when it is CONTENDED (i.e. currently held by a CPU and there are other CPUs waiting for it), the locking procedure will proceed with the `queue` label ❷. This queuing logic will set `tail` to last CPU id waiting for the lock, and will enter busy-looping until the CPU the precedes it in the queue releases it. This is the least useful state for us, so we do not discuss it any further.

4.3 Extracting Primitives

We make the following observations, from an exploit development perspective:

- Unlocking only affects the `locked` byte. Other bytes are left unchanged.
- Spinning at step ❹ is performed only on the `locked` byte. Other lock bytes are not checked at this point.
- If the lock is CONTENDED, i.e. $(\texttt{tail}, \texttt{pending}) \neq (0, 0)$, we jump to the `queue` label in step ❷. Entering the queuing logic with arbitrary `tail` bytes leads to an uncontrolled memory corruption that is likely to result in a crash. For this reason, we should avoid jumping to `queue`.

By utilizing these observations, we managed to come up with 3 exploit primitives, presenting different levels of strength. For the description of the primitives, we will denote the lock value by the tuple `(tail, pending, locked)`.

Semi-Increment Primitive. For this primitive, we setup[2] the lock value to be $(0, 0, x)$ with $x > 0$ (i.e. the lock is in LOCKED state). Then, we trigger the "use", which will try to acquire the lock. Since the lock appears in LOCKED state, the pending bit will be set, and the value of the lock becomes $(0, 1, x)$. This is a *semi-increment* primitive, as the value that overlaps the lock bumped by `0x100`.

It should be noted that the CPU which acquires the lock spins until the `locked` value (x) becomes 0. This is a double-edged sword: We can benefit from this situation because it gives enough time for the rest of the exploit to proceed. However, if the situation is not rectified, then after 10-20 seconds the watchdog will crash the kernel.

Semi-Decrement Primitive. Consider what happens when some other code flow sets the `locked` field to 0, while a CPU is spinning on it. In this case, the CPU will cease spinning and will acquire the lock, which will become $(0, 0, 1)$. It will stay this way for the duration

[2] The setup phase is performed by reallocating the freed `binder_proc` with different object.

of the critical section, which in our case is very short. `spin_unlock()` leaves us with (0, 0, 0). This is a *semi-decrement* primitive, as the initial value was $x > 0$, and we managed to drop it to 0.

The semi-decrement primitive can be good for decrementing a refcount field. The idea is: we will catch the field with an object that have a refcount field exactly in the offset of the `inner_lock`. We will increment the refcount to `0xff`. Next, we will call `spin_lock()` and the refcount will change to `0x1ff` (as the `pending` bit has been set). We will increment the refcount one more time so it becomes `0x200`. The least significant byte (LSB) becomes 0, so the CPU stops spinning, and we get a refcount of `0x1` for a brief moment. The unlock function sets the refcount field to 0. Now, to free the object, we will do another increment + decrement operation. This gives us a use-after-free on the new object.

There are two issues with this approach. The first is that we need to find a structure that have a refcount field exactly at the offset `offsetof(binder_proc, inner_lock)`. On top of that, we must have the ability to increment and decrement it at will. As it so happens, there are handful number of objects with a refcount-like field at our desired offset in `kmalloc-1k`. Although it is possible to do a cross-cache attack to increase the number of available objects to use, this can reduce the reliability of the exploit. Also, regardless of reliability, if the offset of `inner_lock` changes between devices, the exploit technique would not generalize.

The second issue is with the last step: incrementing a refcount when it is 0. To get a use-after-free on the new object, we need to be able to increment the refcount when it is 0, and then decrement it so it will be freed. The problem is that `refcount_t` checks that the refcount is not 0 when `CONFIG_REFCOUNT_FULL` is configured. So we might only seek objects that do not use `refcount_t` and increment the refcount even if it is zero. `CONFIG_REFCOUNT_FULL` was removed from kernels ≥ 5.7, so it is a valid approach for newer devices, but then there is the offset and generalization issue.

Nullifying 2 LSBs Primitive. This primitive begins like the previous: The initial lock value is $(0, 0, x)$ with $x > 0$. After triggering the “use", the lock becomes $(0, 1, x)$ and the CPU is spinning until the least significant byte becomes 0 (the `locked` byte).

Suppose that other CPU overwrites this memory location with $(z, y, 0)$, where z and y are completely arbitrary. In this case, the LSB is 0, so the spinning CPU will stop looping and try to acquire the lock, which becomes $(z, 0, 1)$ (notice how y was overridden to 0). The `spin_unlock()` leaves us with $(z, 0, 0)$.

This primitive is very attractive from exploitation point of view, as it has the potential to nullify the 2 LSBs of a pointer.

5 The Exploit

Based on the 2 LSBs nullifying primitive, we have constructed a robust and stable exploitation technique that is generic enough to work on any 8-aligned spinlock offset. Using our technique, we were able to bypass modern Android kernel security mitigations and achieve arbitrary kernel read-write capabilities. Further, we successfully tested our technique on 3 Android devices, including multiple vendors (Samsung, Google), various Android versions (12, 13), and various kernel versions (5.4.x, 5.10.x).

Limitations. Despite its strengths, it is important to note that this technique does have its drawbacks and limitations. For instance, our technique mandates a preemptive kernel running on multiple cores (SMP). This is necessary to execute the 2 LSBs nullifying primitive, as discussed below. Further, our exploit assumes that `GFP_KERNEL_ACCOUNT` and `GFP_KERNEL` allocations are serviced from the same `kmem_cache`. This assumption is true on 5.10 kernels [9], and on 5.4 kernels with `CONFIG_MEMCG_KMEM` disabled or booted with `cgroup.memory=nokmem` command line parameter. The latter was the case on several Samsung devices we surveyed, including the tested device Samsung Galaxy S21 Ultra running on kernel version 5.4.129. Additionally, our technique may be more challenging to implement and execute than other exploitation techniques.

5.1 Our Target Allocation

To make the exploit somewhat resistant to changing offsets of the `inner_lock`, an ideal object is an array of pointers. This way, we can adjust the exploit so that we always corrupt a pointer (i.e. zeroing out its 2 LSBs), assuming that the `inner_lock` offset aligns to 8.

There are multiple code flows in the kernel that allocate an array of pointers. An interesting one is the *file descriptor table* (fdtable). The file descriptor table is an array of `struct file` pointers, where each `struct file` represents an open file (containing information about its position, flags, inode, etc). The file descriptor table is linked to the process' `files_struct` with the `fdt` field. File descriptors (fd) are just indices to the process' file descriptor table.

In Linux, the `files_struct` itself contains an inline version of the file descriptor table, capable of storing 64 open files without allocating a larger file descriptor table. This is an optimization to reduce the memory footprint for processes that use several file descriptors. If a process uses more than 64 files, the file descriptor table is allocated as a different structure, by calling the function `alloc_fdtable(nr)`.

There are two cases in which the file descriptor table is allocated. The first is when a new process is being created using `fork()` (or `clone()` without `CLONE_FILES`). If the parent process has $nr > 64$ file descriptors, `alloc_fdtable(nr)` is called for the child. Otherwise, the inline fdtable is used.

The second case is when `exapnd_files()` is called to expand a process' file descriptor table. We can reach this function by using the `dup2()` system call:

```
int dup2(int oldfd, int newfd);
```

The `dup2()` system call creates a new file descriptor (`newfd`) that refers to the same open file as an existing descriptor (`oldfd`). If `newfd` was already open, it will be closed before being reused.

If the process started its life with ≤ 64 file descriptors, then when we invoke `dup2(oldfd, newfd)` with `new_fd` ≥ 64, its fdtable must be expanded to accommodate `newfd`, so a larger file descriptor table is allocated.

Regardless of the reason, if a larger allocation of an fdtable has to be made, it is performed by the `alloc_fdtable(nr)` function:

```
struct fdtable *alloc_fdtable(unsigned int nr)
{
    struct fdtable *fdt;
    void *data;

    nr /= (1024 / sizeof(struct file *));
    nr = roundup_pow_of_two(nr + 1);
    nr *= (1024 / sizeof(struct file *)); ❶

    ...

    fdt = kmalloc(sizeof(struct fdtable),
     GFP_KERNEL_ACCOUNT);
    if (!fdt)
        goto out;
    fdt->max_fds = nr;
    data = kvmalloc_array(nr, sizeof(struct file *),
      GFP_KERNEL_ACCOUNT); ❷
    if (!data)
        goto out_fdt;
    fdt->fd = data;

    ...

    return fdt;

out_fdt:
    kfree(fdt);
out:
    return NULL;
}
```

When `nr` < 128 it is rounded up to 128 at ❶. An array of 128 `struct file` pointers (1024 bytes) is then allocated in ❷. The allocation will use the general purpose cache `kmalloc-1k`. This is perfect for us, as this allocation uses the same cache as our vulnerable object, `binder_proc`.

For the rest of the exploit we will use the `dup2()` function to reach this allocation. Using this method we can support `inner_lock` offsets ≥ 512. This is acceptable for us, as we did not encountered a smaller offset. Otherwise, we would have used the `fork()` method.

5.2 Nullifying 2 LSBs of `file` pointer

In this section, we adapt the nullifying 2 LSBs (least significant bytes) primitive to corrupt a `struct file` pointer.

Recall that to trigger this primitive, we need to enter `spin_lock()` with a lock value $(0, 0, x)$ where $x > 0$. To do that, we will reallocate `binder_proc` with a TTY write buffer that contains the value `0x00000041` repeatedly. A TTY write buffer is allocated using `kmalloc()` when we first write data to a pseudo-terminal (PTY) we opened. One of its useful properties is that we can store arbitrary binary information in it. The following snippet results in an allocation of a TTY write buffer in the `kmalloc-1k` general cache:

```
char data[1024];
int pty_fd = open("/dev/ptmx", O_RDWR);
write(pty_fd, data, 1024);
```

After reallocation, we will trigger the use-after-free, by closing process C, so that `spin_lock()` will be called. One of the `0x00000041` values in the TTY write buffer will overlap with the lock value and will change to `0x00000141` (the `pending` bit is set). The CPU will now spins until the LSB, which is currently `0x41`, becomes `0`.

At this point, from another CPU, we will free the TTY write buffer and reallocate them as file descriptor table using the `dup2()` technique. There is one obstacle in this approach: the allocation of the file descriptor table, despite being a plain `kmalloc()` (not `kzalloc()`), is initialized to 0 as a result of the `init_on_alloc` policy (which is enabled by default in Android kernels). That is too bad, since a value of 0 will cause the spinning CPU to prematurely exit the loop, before any `struct file` pointer got a chance to be written at that location.

To cope with that, we will stress the spinning CPU with interrupts, attempting to slow it down significantly *while it spins*. By doing so, we hope to win a tiny race condition where the allocation of the fdtable is initialized to 0 and a struct file pointer is written, without the spinning CPU noticing that a value of 0 was written there for a brief moment. We found that the `timerfd` technique of Jann Horn is suitable to accomplish this goal [3].

Even if we managed to win this tiny race using interrupts, there is no guarantee that the `struct file` pointer that overlaps the lock will have LSB 0. And yet, having LSB 0 is crucial for the success of the primitive, otherwise the CPU keeps spinning indefinitely.

To solve this issue, we repeatedly perform `dup2(random_fd, target_fd)` where `target_fd` is calculated as `binder_proc`'s `inner_lock` offset divided by 8 (i.e. so that `fdt[target_fd]` overlaps with the lock).

Depending on the size of `filp_cache` and the size of `struct file`, we can calculate the probability of a random `struct file` to have LSB 0. For example, if the `filp_cache` is 2 pages in size and the size of `struct file` is 0x140 bytes, we will have 7 out of 25 `struct files` with LSB 0, per slab. So, repeating `dup2(random_fd, target_fd)` 16 times guarantees a success rate $> 99\%$.

As soon as a `struct file` with LSB 0 is placed, the spinning CPU will exit busy-looping, nullifying the 2nd LSB. Exactly the corruption we were shooting for.

To summarize, the following steps will be taken to nullify the 2 LSBs of a `struct file` pointer:

1. Trigger the vulnerability, so that the `binder_proc` structure of "Process *B*" is freed and a pointer to it is kept in `ref->proc`. Avoid triggering the "use" yet (i.e. do not close "Process *C*").

2. Reallocate the `binder_proc` structure by spraying TTY write buffers. Make the TTY write buffers contain `0x00000041` repeatedly (i.e. LSB non-zero and the rest are zero).

3. Trigger the "use" on CPU 4: close "Process *C*" so that the `spin_lock()` function will be called on one of the `0x00000041`. This will change the value to `0x00000141` and CPU 4 will spin until the LSB becomes 0.

4. Raise `timerfd` interrupts on CPU 4 *while it spins* in `spin_lock()` function.[3] This will slow down CPU 4 significantly, allowing us to win a tiny race with the initialization to 0 of the allocated file descriptor table.

5. Free the TTY write buffers and reallocate them with the fdtable allocation (the target allocation) using the `dup2()` technique.

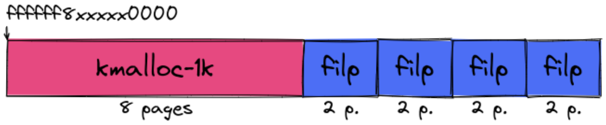


Figure 4: Physical memory shaping

6. Continue making `dup2(random_fd, target_fd)` until a `struct file` pointer with LSB zero is placed in this offset. When this happens, CPU 4 will stop spinning and exit the `spin_lock()` function. After the `spin_unlock()` function, the `struct file` pointer that was placed there will be corrupted (i.e. its 2 LSBs will be zero).

5.3 Exploit Strategy

Our assumption is that we can corrupt (i.e. zeroing out 2 LSBs) a `struct file` pointer at a known fd number. We shall now take the following steps to achieve arbitrary R/W capabilities:

1. Shape physical memory so that beginning at an address aligned to 16 pages we will have the `kmalloc-1k` slab (containing object at our control), followed by 4 slabs of `filp`, as depicted in Figure 4. This way, if we corrupt any `struct file` pointer residing on one of these `filp` caches, then we will land on the first object in the `kmalloc-1k` cache.

2. Fill the `kmalloc-1k` slab with TTY write buffers, so that we can fill it with arbitrary data. Each TTY write buffer will contain a fake `struct file`.

3. Invoke `close()` on the corrupted fd. This will end up freeing the TTY write buffer.

4. Catch the freed TTY write buffer with an array of `struct pipe_buffers`.

5. Leak a `pipe_buffer` to bypass kASLR.

6. Fake a `pipe_buffer` and use the pipe to achieve arbitrary R/W primitive to the linear mapping.

7. Override `addr_limit` to gain arbitrary R/W primitive, bypassing AArch64 UAO feature.

Below we describe each step in detail.

5.4 Shaping Physical Memory

The rest of the exploit assumes that the corrupted `struct file` pointer points to a TTY write buffer that is under our control. In this step we will shape physical

[3] To make the interrupts raise on CPU 4, we need to setup the `timerfd` on CPU 4. We cannot do this when the "use" already happened (CPU 4 is non-preemptible from this point onwards). Therefore, we setup the `timerfd` before triggering the use-after-free, which requires some tuning.

memory to make this assumption true, with high likelihood. (Do not confuse this TTY write buffer with the one used to trigger the nullifying 2 LSBs primitive.)

To see if such an assumption is even feasible, we need to look at the physical properties of the slabs used for the allocation of `struct file` and the TTY write buffer.

`struct files` are allocated from a dedicated memory pool called `filp_cache`. The `filp_cache` consumes 2 physical pages per slab and can store up to 25 `struct files` per slab[4].

TTY write buffers are allocated from the `kmalloc-1k` general purpose cache. The `kmalloc-1k` cache contains up to 32 objects per slab. The size of each object is 1024 bytes, so a total of 8 pages are used per slab.

Our general plan is to spray many `struct files` and TTY write buffers in a particular way, so that at the beginning of a 16-pages aligned address we have TTY write buffers with controlled content, and `struct files` are positioned after them, as in Figure 4.

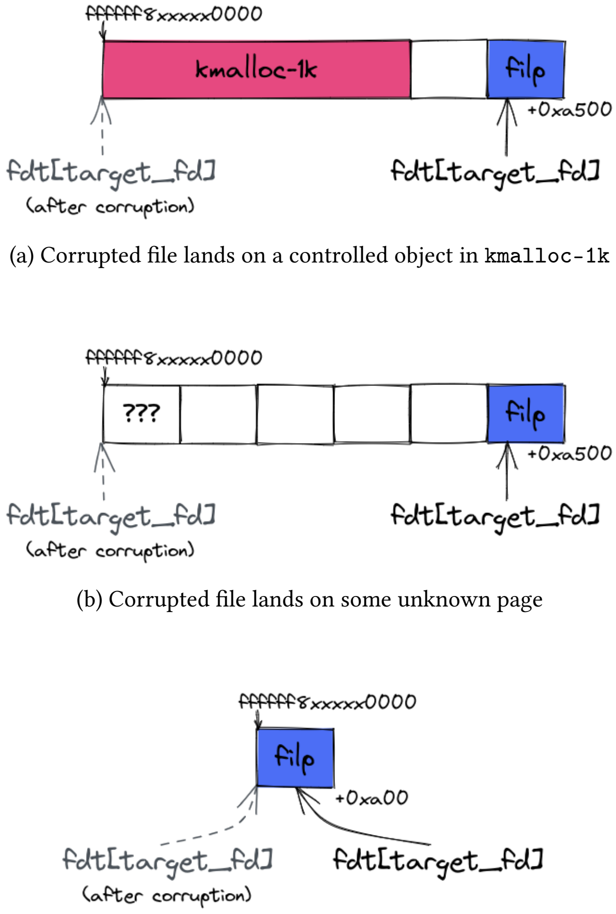
Observe that if we corrupt any `struct file` positioned in any of the `filp` caches shown on Figure 4, the resulting pointer will point to the beginning of the `kmalloc-1k` slab (which is filled with TTY write buffers with controlled content). For this reason, we architect our exploit to use random files from these `filps` when performing the `dup2(random_fd, target_fd)` operations.

Spraying strategy. Our spraying strategy is to repeat the following as much as we can: allocate 32 TTY write buffers, followed by opening $25 \times 4 = 100$ files. By doing this, we wish that the allocation of the first 32 objects will create a new slab for `kmalloc-1k`. Additionally, we expect that the next 100 allocations of files will create new 4 slabs for `filp_cache`.

It is important to note that there is a system-wide limitation on the number of open pseudo-terminals (4096 by default on Android; see `/proc/sys/kernel/pty/max`). For this reason, we can repeat the above up to $4096/32 = 128$ times.

Clearly, wishes not always come true. To improve statistics and to ensure that new slabs are allocated when we spray, we do some "warm-up" rounds. In the warm-up, we allocate many objects in `filp_cache` and `kmalloc-1k`. In this phase we do *not* use TTY write buffers for the spray – as these are a limited and precious resource. Instead, we simply open */dev/hwbinder* devices – this allocates `struct file` from `filp_cache` and `binder_proc` from `kmalloc-1k`.

For the warm-up phase, it should be noted that there is a per-process limitation on the number of open file-descriptors (32768 on Android; see `/proc/[pid]/limits`).[5] Consequently, we distribute the warm-up phase across multiple processes.



(a) Corrupted file lands on a controlled object in `kmalloc-1k`

(b) Corrupted file lands on some unknown page

(c) Corrupted file lands on other `struct file`

Figure 5: Possible situations after shaping

Handling shaping failures. To make the exploit more reliable, we must detect whether the shaping succeeded or not. For this, we need to differentiate between these situations (Figure 5):

1. The corrupted file pointer points to our TTY write buffer in `kmalloc-1k`. This is the desired situation.
2. The corrupted file pointer points to some unknown page. In this case, any operation done on the file might create havoc on the system. Therefore, we are careful to block the process holding the corrupted

[4] The actual number might change depending on the kernel version.

[5] There is also a system-wide limitation on the number of open file descriptors for *all* processes – see `/proc/sys/fs/file-max`. On Samsung Galaxy S22 it is 583277, which is sufficiently large for our purposes.

file, until the exploit succeeds and we can rectify the situation.

3. The corrupted file pointer points to some `struct file`. We could have keep the corrupted file's process alive in this case too. However, to minimize load on the system that might add-up on each failed attempt, we send the file to a designated "graveyard" process (we do this using UNIX sockets). The file will live in the graveyard process until the exploit succeeds and the situation can be corrected.

We classify each situation by *indirectly* deduce the value of bits from the corrupted file pointer. The idea is as follows: we invoke an operation (system call) on the corrupted fd and, from its result, deduce the value of various bits. For example, one technique we used was to call `timerfd_gettime()` to infer whether `fdget()` succeeds on the struct file or not:

```
bool fdget_succeed(int fd) {
    int ret = timerfd_gettime(fd, NULL);
    if (ret == -1 && errno == EBADF)
        return false;
    else if (ret == -1 && errno == EINVAL)
        return true;
    /* Unreachable. "fd" is not timerfd file. */
}
```

Assuming the corrupted fd has not landed on a `timerfd` type of file by chance, this method will tell us whether `fdget()` passes on the corrupted file. If it passes, it tells us that the file's count is greater than 0 and that the file's mode do not have the `FMODE_PATH` bit set. This gives us two bits of information. We proceed in similar ways to extract more bits of information about the file.

Note that any operation that we invoke on the corrupted `file` *must* avoid dereferencing any pointer of it, since we can be in the unknown page situation. This greatly limits the number of system calls we can call to extract information. For example, this rules out trivial methods such as reading `/proc/[pid]/fdinfo/[fd]`. Additionally, any system call's implementation that calls LSM hooks (e.g. `fcntl()`) are out of the game, since the LSM hooks dereference `f_security`.

All in all, in cases of failure, we restart the exploit and try again. On success, we proceed to the next step, closing the corrupted file descriptor.

5.5 Closing the fd

Our strategy will be to call `close()` on the corrupted fd, so that we will free the TTY write buffer. To pull this off, we need to craft a fake `struct file`, so that the kernel will happily free the object for us.

When calling `close()`, the function `filp_close()` is ended up being called (*fs/open.c*):

```
int filp_close(struct file *filp, fl_owner_t id)
{
    int retval = 0;

    if (!file_count(filp)) { ❶
        printk("VFS: Close: file count is 0\n");
        return 0;
    }

    if (filp->f_op->flush) ❷
        retval = filp->f_op->flush(filp, id);

    if (likely(!(filp->f_mode & FMODE_PATH))) { ❸
        dnotify_flush(filp, id);
        locks_remove_posix(filp, id);
    }
    fput(filp); ❹
    return retval;
}
```

Our goal is to reach the function `fput(filp)` ❹, as it decreases the file count by 1, and if it reaches 0, will free the `file` structure. To reach it, we need to satisfy three conditions:

1. *The file count must be 1.* This ensures that we pass the first condition ❶ and also makes sure the file struct will be freed by `fput()` (instead of having its count field decreased by 1).

2. *The file operations pointer (*`f_op`*) must be a valid kernel pointer, and* `f_op->flush` *must equal 0.* This is required to make sure we pass the second condition ❷. We can do this by having `f_op` pointing to a fixed address in the kernel that is known to contain 8 bytes of zeros.

3. *The file mode must have the* `FMODE_PATH` *bit set.* This is required to avoid entering the third condition ❸.

If the above conditions are satisfied, `fput()` will call `__fput()`, the actual function that frees the `file` struct:

```
static void __fput(struct file *file)
{
    struct dentry *dentry = file->f_path.dentry;
    struct vfsmount *mnt = file->f_path.mnt;
    struct inode *inode = file->f_inode;
    fmode_t mode = file->f_mode;

    if (unlikely(!(file->f_mode & FMODE_OPENED))) ❶
        goto out;

    ...
out:
    file_free(file); ❷
}
```

If the FMODE_OPENED bit is not set ❶, we immediately reach file_free(file) on ❷ (*fs/file_table.c*):

```
static inline void file_free(struct file *f)
{
    security_file_free(f);
    if (!(f->f_mode & FMODE_NOACCOUNT))
        percpu_counter_dec(&nr_files);
    call_rcu(&f->f_u.fu_rcuhead, file_free_rcu);
}
```

security_file_free(f) ensures that f_security is not NULL before freeing it, so, setting f->f_security to NULL makes it return immediately. Then, after an RCU grace period, file_free_rcu() will be called:

```
static void file_free_rcu(struct rcu_head *head)
{
    struct file *f = container_of(head, struct file,
      f_u.fu_rcuhead);

    put_cred(f->f_cred);
    kmem_cache_free(filp_cachep, f); ❶
}
```

The call to kmem_cache_free() will free the file, pointed by f. Importantly, f will be freed back to the slab cache it allocated from, *not* filp_cachep as supplied in the first argument ❶. In our situation, f is a fake struct file, originally allocated from kmalloc-1k cache. The call to kmem_cache_free(filp_cachep, f) will free f as if the call was plain kfree(f), except for a *warning* message outputted on kmsg:

```
cache_from_obj: Wrong slab cache. filp but object is
      from kmalloc-1k
```

The reason for this behavior is that Linux's SLUB allocator determines the target slab from the virtual address of the object being freed, not the argument passed to kmem_cache_free().

The end result is that after the corrupted fd is closed, a TTY write buffer is freed, while still having other reference (the TTY fd), pointing to the same (freed) location. In other words, we were upgraded to a use-after-free situation on the TTY write buffer.

5.6 Leaking Pipe Buffer

We are going to leverage this use-after-free situation by leaking the content of struct pipe_buffer:

```
/**
 * struct pipe_buffer - a linux kernel pipe buffer
 * @page: the page containing the data for the pipe
   buffer
 * @offset: offset of data inside the @page
 * @len: length of data inside the @page
 * @ops: operations associated with this buffer
 * @flags: pipe buffer flags
 * @private: private data owned by the ops
 **/
struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    const struct pipe_buf_operations *ops;
    unsigned int flags;
    unsigned long private;
};
```

This structure interests us since by leaking it we get a pointer to a struct page and, most importantly, a pointer to the pipe operations (ops) which is stored on the kernel data section. This means that ops is offsetted by a constant amount from the beginning of the kernel image. (The "constant amount" may change between different kernel versions and configurations, but would still be the same on a per-device per-kernel basis.) Therefore, having leaked ops, we can calculate the base of the kernel image by subtracting a fixed offset from it, thereby bypassing the kASLR mitigation.

When a pipe is allocated using the system call pipe(), an array of 16 struct pipe_buffer is allocated using kcalloc() (*fs/pipe.c*):

```
struct pipe_inode_info *alloc_pipe_info(void)
{
    struct pipe_inode_info *pipe;
    unsigned long pipe_bufs = PIPE_DEF_BUFFERS;

    pipe = kzalloc(sizeof(struct pipe_inode_info),
     GFP_KERNEL_ACCOUNT);

    ...

    pipe->bufs = kcalloc(pipe_bufs, sizeof(struct
     pipe_buffer), GFP_KERNEL_ACCOUNT); ❶

    ...
}
```

The size of struct pipe_buffer is 40 bytes, so the pipe->bufs array is of size $16 \times 40 = 640$ bytes and as such it will be allocated in kmalloc-1k general cache. This is perfect for us, since we can catch the freed TTY write buffer with this array of pipe buffers.

Now, the idea is to ensure that the page and ops fields of single pipe buffer are populated, and then read the content of the pipe buffer using the second reference we have from the TTY fd.

Populating one pipe buffer can be easily done by writing to the pipe, say 0x1000 bytes. We ensure that this operating does not block by having the write-end of the pipe entering non-blocking mode.

Reading from the overlayed TTY write buffer is more involved than simply invoking read() on the TTY fd. When writing data to a TTY write buffer, the data is written to a TTY write buffer (see do_tty_write() in *drivers/tty/tty_io.c*). The write buffer is then passed to the function `n_tty_write()` (in *drivers/tty/n_tty.c*), which internally calls `pty_write()` (in *drivers/tty/pty.c*). In `pty_write()`, the data of the write buffer is *copied* to other end of the TTY. In our case, the other end is under our control, so we can read the *copied* data by invoking `read()` on the TTY fd.

We conclude that we cannot *directly* read the contents of the TTY write buffer, since we read a copy a data previously written. However, consider the case where we *block* the writer after data is written to the write buffer, but before the data in the write buffer is copied to the other end. In such a case, any data written to this memory location while the writer is blocking will be copied to the other end when the writer unblocks.

So, to leak a pipe buffer, we write zeroes to the TTY write buffer and block the writer thread before the write buffer is copied. While the writer thread is blocking, we use another thread to write data to the corrupted pipe (say, 0x1000 bytes). Recall that this operation will populate the `struct page` and ops pointers of one of the `pipe_buffers`. We then unblock the writer thread and let the data being held in the write buffer be copied to the other end of the TTY – this data will now contain the `pipe_buffer` contents. We proceed by reading from the TTY fd, thereby leaking the `pipe_buffer`.

It remains to show how can we block the writer thread exactly at the wanted position. To accomplish that, we use the software flow control capabilities of the TTY driver. It is possible to suspend transmission or reception of data using the `ioctl` command `TCXONC` invoked on the TTY fd. If the argument is `TCIOFF` then this will cause the writer thread to block on `n_tty_write()`. To unblock the writer thread, we use `TCION` instead of `TCIOFF`. Exactly the functionality we seeked.

5.7 Arbitrary R/W to the Linear Mapping

Up until now, the situation is as follows: we bypassed kASLR (by leaking the content of a pipe buffer) and we have two references to the same memory location: TTY write buffer and the array of pipe buffers. By writing to the TTY fd, we write data to the TTY write buffer – which modifies the contents of the pipe buffer array.

We can achieve write-what-where and arbitrary-read primitives using the corrupted pipe. The idea is to use the TTY fd to overwrite one of the pipe buffers with the address of the physical page we want to read from/write to. Then, we invoke a read/write operation on the pipe, which results in reading/writing the contents of the target page to a supplied userspace buffer.

To execute this idea, we need to convert a linear mapping's kernel virtual address to a `struct page` address. To understand how this conversion is done, we need to talk a bit about how physical memory is managed in the kernel.

The kernel describes a single physical page using `struct page`. The mapping between PFNs to `struct page` is one-to-one. However, architectures might need to reserve some regions of memory from use by the kernel. This is supported with Linux's `SPARSEMEM` memory model. On AArch64's Linux with `CONFIG_SPARSEMEM_VMEMMAP` enabled, when the kernel boots, it designates a contiguous virtual memory area to store an array of `struct pages`, one for each page of physical memory that can be used by the kernel. This area is called `vmemmap`.[6]

This is done so to make the conversion between pages and virtual addresses fast. Given a virtual address in the linear mapping `x`, we can calculate its corresponding `struct page` address using:

```
struct page *virt_to_page(void *x) {
    u64 index = ((u64)x - PAGE_OFFSET) / PAGE_SIZE;
    u64 addr = VMEMMAP_START + index * sizeof(struct
      page);
    return (struct page *)addr;
}
```

`PAGE_OFFSET` is `0xffffff8000000000` on AArch64 Linux ≥ v5.4, configured with 39-bit virtual addresses [2]. `VMEMMAP_START`, defined in *arch/arm64/include/asm/memory.h*, is the *fixed* virtual address pointing to the start of the `vmemmap`.

5.8 Arbitrary R/W

We upgrade to arbitrary R/W on every kernel virtual address by overwriting our task `addr_limit` and bypass UAO (user access override).

Finding our task struct. To modify `addr_limit`, we need to find our task struct pointer first. We will take the naive approach of traversing the task's list, starting from `init_task`, located in the kernel image's data section.

With the R/W primitive achieved so far, we can reliably read and write to any address in the linear mapping, which contains the kernel's heap (and our task struct). However, a read/write on kernel image's address (e.g. `init_task`) requires a conversion to the linear mapping.

This conversion can change per-device, depending on whether the kernel image is physically randomized. On Samsung devices tested, the virtual and physical kASLR slides are equal. Since we know the virtual kASLR slide, the conversion amounts to subtracting a constant. On

[6] The `vmemmap` virtually covers the reserved regions but does not map them, so accessing the `vmemmap` on reserved regions will raise a fault.

Google Pixel 6, there is no physical kASLR, so the conversion becomes:

```
u64 kimg_to_lm(u64 x){
    return PAGE_OFFSET + (x - kimg_base);
}
```

UAO. A brief introduction to UAO is in order. UAO (User Access Override) is a feature introduced on Armv8.2 that controls the behavior of unprivileged load-/store instructions (`ldtr*`/`sttr*`). When those instructions are executed in EL1 (kernel mode), they behave as if they were executed in EL0. This allows the kernel to access userspace memory without temporarily disables PAN (Privilege Access Never). Consequently, these instructions are used by the kernel user accessor functions `copy_[from|to]_user()`.

There are situations[7] where the kernel uses `copy_[from|to]_user()` with *kernel* buffer address instead of user buffer address. To support this use-case, the kernel cannot use the unprivileged load/store instructions (as access to a kernel address will generate a fault). UAO provides a solution to this problem. When UAO is enabled, the behavior of the unprivileged load/store instructions is overriden, i.e. they behave as normal load/store instructions (`ldr*`/`str*`). Hence, when UAO is enabled, userspace accessor functions can operate on kernel addresses but not on user addresses since PAN is enabled. In Linux, UAO is enabled when the task's `addr_limit` is set to `KERNEL_DS`[8], and is disabled otherwise.

UAO bypass. To upgrade to full R/W primitives we use the following strategy: Spawn two threads, T1 and T2, and set T2's `addr_limit` to `KERNEL_DS` using the R/W primitives already obtained. T1 and T2 will have a shared memory region and a pipe for read/write kernel information. For a `kernel_read(addr, size)` operation, T2 will write the data from the kernel address `addr` to the pipe buffer by executing `write(pipe[1], addr, size)`. This operation succeeds since UAO is enabled for T2. T1 will read the contents of `addr` from the pipe using normal `read()` syscall. A `kernel_write(addr, size)` operation is implemented analogously. The shared memory is used for synchronizing T1 and T2 and passing messages without T2 having to make system calls (as it can only provide kernel addresses for buffers). Please refer to [6] and [8] for more information about the UAO feature.

5.9 Escalating Privileges

There are multiple techniques to execute code as the root user and to disable/bypass SELinux and other mitigations. Generally, the techniques vary between vendors, kernel versions, Android versions, etc.

On the Samsung Galaxy S22 and Google Pixel 6 devices the `selinux_state` structure contains the `enforce` field (kernel is configured with `SECURITY_SELINUX_DEVELOP`), so bypassing SELinux on these devices amounts to override this field to 0.

To get root credentials on the Pixel 6, we override our `task_struct`'s `cred` and `real_cred` fields with the credentials of the `init` process.

6 Results

We tested our exploit on the following 3 Android devices:

- Samsung Galaxy S22 with kernel version 5.10.81 running Android 12 with June 2022 Security patch level.
- Google Pixel 6 with kernel version 5.10.66 running Android 12 with May 2022 security patch level. Also tested with kernel version 5.10.107, Android 13, September 2022 security patch level.
- Samsung Galaxy S21 Ultra with kernel version 5.4.129 running Android 12 with March 2022 security patch. (Tested from rooted device only.)

We have not tested our technique on an Android device running kernel 4.19. However, manual code inspection suggests that it is possible to adapt the exploit to such kernels as well.

Our testing shows that our exploit successfully executes 4 out of 5 times. The main source of instability is a failure to shape physical memory, which results in more iterations of the vulnerability. Each vulnerability invocation risk in a failed reallocation of the `binder_proc` structure, which might lead to a kernel crash when the `spin_lock()` function is called on random bits. We believe that the exploit can be fine-tuned to rectify these issues and improve statistics. No further efforts were made to improve stability.

Our implementation takes 5-60 seconds to complete, depending on the number of iterations performed to succeed in shaping, and the number of `dup2()` steps taken.

7 Conclusion

This exploit shows that even a seemingly weak and constrained initial primitive can be used to completely com-

[7] For example, syscall emulation within the kernel (kernel_setsockopt, kernel_recvmsg, etc).

[8] `KERNEL_DS` is used to represent the `addr_limit` of kernel threads. Therefore, UAO can be seen as a security defense against invalid userspace accesses in kernel threads.

promise a system, despite all security mitigations deployed.

8 References

[1] G. Beniamini. Issue 1404: Android: Hardware Service Manager Arbitrary Service Replacement due to getpidcon. URL https://bugs.chromium.org/p/project-zero/issues/detail?id=1404.

[2] S. Capper. arm64: mm: Introduce 52-bit Kernel VAs (b6d00d4), 2019.

[3] J. Horn. Racing against the clock – hitting a tiny kernel race window, 3 2022. URL https://googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html.

[4] X. Jin and R. Neal. The Art of Exploiting UAF by Ret2bpf in Android Kernel. URL https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf.

[5] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1): 21–65, feb 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. URL https://doi.org/10.1145/103727.103729.

[6] V. Nikolenko. UAO (User Access Override) as a mitigation against addr_limit overwrites. URL https://duasynt.com/blog/android-uao-kernel-expl-mitigation.

[7] M. Stone. Bad binder: Android in-the-Wild Exploit, . URL https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[8] M. Stone. A Very Powerful Clipboard: Analysis of a Samsung in-the-wild exploit chain, . URL https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html.

[9] J. Weiner. mm: memcg/slab: use a single set of kmem_caches for all allocations (10befea9), 7 2020.

A Queued Spinlocks in Detail

Spinlock is a lock that busy waits (*spins*) if it is already-acquired when trying to acquire it. The spinning ends when the lock holder releases the lock. Since waiting for the lock wastes CPU cycles, spinlocks are appropriate when the critical section run for short periods of time. They are also suitable for use in contexts that are not allowed to sleep (e.g. IRQ handler, softirq, tasklet).

Perhaps the simplest implementation of spinlocks is *test-and-set* (TAS). In this implementation, the lock is a single bit, where 0 means that the lock is not acquired. The lock is acquired using an atomic test-and-set instruction which writes 1 to the lock and returns its old value. If the old value is 0, the lock was previously unlocked, so this operation just acquired it. Otherwise, the old value was 1, meaning the lock is held by another thread, so we busy wait by repeatedly trying to acquire the lock until it is released.

This approach has significant drawbacks. When there are multiple waiters for the lock, only one of them win acquiring it. The others still wait. This can cause starvation if a thread always loses. Further, each waiter spins as fast as it can, each time invoking an expensive read-modify-write instruction. On cache-coherent systems with multiple CPUs, this type of instructions might cause many remote cache invalidations, which further degrades performance. There exist spinlock implementations that remedy these issues to some extent. We concentrate on one such implementation called MCS (after the initials of its inventors).

A.1 MCS Lock

MCS locks solves the starvation problem by ensuring that a FIFO queue is maintained so every thread that wants to acquire the lock will eventually acquire it. It solves the caching issue by having each thread spins on a separate, locally-accessible value.

In MCS lock, the lock is a pointer to a structure called "MCS node". This structure contains a `next` pointer to form a linked list of MCS nodes, and a single bit called `locked`. The lock pointer always points to the tail of the queue. If it is `NULL`, the queue is empty and the lock is unlocked.

When a thread wants to acquire the lock, it allocates a new MCS node, `n`, and atomically places itself in the end of the queue. If it is the first node in the queue (tail pointer was `NULL`), then it just acquired the lock. Otherwise, it sets `n->locked` to `1` and busy waits waiting for `n->locked` to become 0. When a thread releases the lock, it removes itself from the queue and "wakes-up" its successor (if it exists) by setting the successor's `locked` value to 0.

Of course, to make this algorithm correct, we must ensure that certain operations are atomic (like insert/delete from the queue), and to account for race conditions between different acquirers. Full treatment can be found in the original paper [5, §2.4].

A.2 The Linux implementation

Spinlocks in the Linux kernel are implemented as (a form of) MCS locking, and called "queued spinlock". Traditionally, spinlocks in the kernel are 4-bytes wide, and engineering effort was put to keep it this way (so the kernel memory footprint would not increase).

The basic observation is that spinlocks disable preemption – the CPU cannot be preempted to run another process while executing a critical section protected by spinlock (also the `spin_lock()` procedure itself is not preemptible).[9]

For this reason, the length of the queue is upper bounded by the number of CPUs in the system. This makes the allocation of MCS nodes simple: they are statically allocated per-CPU. Note that static MCS node allocation is subtle, since spinlocks are used in different contexts in the kernel. For example, consider a task in process context that runs on CPU 0 and uses the MCS node for CPU 0. If hardware interrupt is triggered on CPU 0 at the same time, the interrupt handler might also need to use spinlock for race protection (different spinlock! of course). Which MCS node will be used in this case? Clearly it can't be the same MCS node as the one used in the process context. For this reason, *multiple* MCS nodes are allocated for per-CPU. Currently, 4 MCS nodes are statically allocated for each CPU. 4 nodes account for the different contexts the kernel can be in: Process context, softirqs, hardware interrupts and NMI. As a comment in code states: "4 nodes are allocated based on the assumption that there will not be nested NMIs taking spinlocks". If this is not the case, the code resorts to spin on the lock value itself. So in this rare event, the spinlock is actually reducing to a simple test-and-set lock.

Since MCS nodes are statically allocated, there is no need to store the full pointer in the lock – it suffices to store the CPU id $\in [0, \mathtt{num\_cpus} - 1]$ and the context index $\in [0, 3]$. This enables the developers to keep the spinlock 4-bytes, which are broken into these subfields (ordered from MSB to LSB):

`tail` **(2 bytes):** Stores a pair (cpu, index).

`pending` **(1 byte):** Stores the pending bit. (More on this below.)

`locked` **(1 byte):** Non-zero if the lock is held by the thread in the head of the queue.

The first optimization Linux makes is that the first locker need not allocate any MCS node to tell other threads it holds the lock. It simply communicates this information by setting the `locked` field to 1. This avoids the cache-line miss associated with accessing the per-CPU MCS nodes array.

When a second locker comes along, it sees that the lock is held by examining the `locked` byte. According to the MCS algorithm, it needs to allocate a node and spin on it. Here Linux makes a second optimization: it sets the `pending` field to 1 and spins until `locked` is set to 0. When the first locker releases the lock, it sets the `locked` field to 0, causing the second locker to stop spinning, clear the pending bit and acquire the lock (by setting `locked` to 1). This is called "optimistic spinning", hoping that the first lock holder will quickly release the lock so the second locker will acquire it, avoiding entering a slow-path involving MCS nodes (this also avoids the cache-line miss associated with accessing the per-CPU data). MCS nodes are used in situations where the lock is seriously contended (at least two lockers in the queue).

B Triggering with Strong Binder Reference

It is possible to trigger the vulnerability also by sending a strong remote Binder reference (object of type `HANDLE`) from process $\mathcal{A}$ to $\mathcal{B}$. However, this involves an extra race condition.

When a strong handle is translated, `binder_inc_node_nilocked()` will execute the following code:

```
static int binder_inc_node_nilocked(struct
     binder_node *node, int strong,
                    int internal,
                    struct list_head *target_list)
{
    struct binder_proc *proc = node->proc;

    if (strong) {
        if (internal) {
            if (target_list == NULL && ❶
               node->internal_strong_refs == 0 && ❷
                !(node->proc &&
                  node == node->proc->context->
     binder_context_mgr_node &&
                  node->has_strong_ref)) {
                return -EINVAL; ❸
            }
            node->internal_strong_refs++;
        } else
            node->local_strong_refs++;
        ...
    } else {
        ....
    }
    return 0;
```

[9] This is required to prevent high latency inside the critical section (https://lwn.net/Articles/828616/): if the thread holding the lock got preempted, it is possible for other threads to spin on the same lock, wasting CPU cycles. In addition, disabling preemption has a desirable side-effect of disabling task migration to another CPU on SMP systems. For uniprocessor systems with preemptive kernels, there's no need for spinning at all as it suffices to just disable preemption.

```
}
```

The `strong` argument is now set to 1, `internal` is set to 1 as well and `target_list` is still NULL. We can see that if strong and internal reference is taken (as in our case), the function checks if `target_list` is `NULL` ❶ and `node->internal_strong_refs` is 0 ❷. (The code also checks whether the node is not the context manager node, which it isn't, so no problems there.) `target_list` is `NULL` in our flow, but `node->internal_strong_refs` is not 0. Can we force it to be 0 somehow?

Recall that the node is owned by process C. Process $\mathcal{A}$ obtained a strong reference to it, and process $\mathcal{A}$ passes this reference to the dying process $\mathcal{B}$.

Process $\mathcal{A}$ obtained a strong reference to the node from C. An internal strong reference was taken at the time C sent strong nodes to $\mathcal{A}$. It turns out that $\mathcal{A}$ has the power to decrease the internal strong reference to 0. It can do that by issuing the `BC_RELEASE` driver command, along with the handle it has to C's node. This command calls

```
binder_update_ref_for_handle(𝒜 ,   /* proc */
                             HANDLE,
                             false,  /* increment */
                             true);  /* strong */
```

... which ends up calling

```
binder_dec_ref_olocked(ref, true /* strong */);
```

This function decreases `ref->data.strong`, and if it becomes 0, a strong internal reference is dropped from the node using `binder_dec_node()`.

When sending a strong handle (object of type `HADNLE`), the driver ensures that the sending process actually has a strong reference on this handle. (Otherwise, an error is generated saying "tried to use weak ref as strong ref".) For this reason, the decrement caused by `BC_RELEASE` must happen *after* the handle value specified by $\mathcal{A}$ is mapped to a Binder reference during handle's translation in `binder_translate_handle()` (see § 3.3).

If the race is won, `binder_inc_ref_for_node()` *will* insert a new `binder_ref` to the dying process $\mathcal{B}$ but *will not* be able to take a strong reference on the node, since `node->internal_strong_refs` will be 0, so `-EINVAL` error will be returned.

C Widen the Binder Vulnerability Race Window

Ideally, we would have wanted to include many objects to be translated before the faulty Binder reference is passed as `WEAK_HANDLE` object. This would buy us time to close process $\mathcal{B}$, and have `binder_deferred_release()` completed, before the `WEAK_HANDLE` object is translated.

However, due to a limitation on the size of Binder transactions (1MB), we cannot have unlimited number of objects to translate. So, we need to choose them carefully.

We need to answer the following questions:

1. Which object(s) to use?
2. When to close process $\mathcal{B}$? What will be the deciding moment?

For the first question, we use a single `FDA` object (fd array). An `FDA` object is capable of submitting a vast amount of objects (250,000), while minimizing the space in the transaction needed to specify each fd. Further, this object is particularly attractive as, starting with kernel 5.4, each fd translation results in (only) a fixup entry being allocated and initialized. With 250,000 fds, we delay the race window in about 400ms, which is more than enough to win.

For the second question, we exploit an implementation detail in how transactions are moved to the other process. The implementation copies the transaction buffer to the other process *before* the objects are translated. Therefore, we can include a magic value at the start of the transaction buffer, and have process $\mathcal{B}$ busy-loops until this value is seen in its Binder memory map. As soon as it observes this value, it immediately closes its Binder file descriptor.

With this, we trigger the vulnerability 100% of the times.