CVE-2024-26926

by Maher Azzouzi

Bug Overview

CVE-2024-26926 is a Linux kernel bug in the Binder component primarily affecting Android devices. The commit `6d98eb95b450` introduced the vulnerability by making changes to how binder objects are copied. In the function `binder_get_object()`, a userspace pointer was added to copy objects directly from it using `copy_from_user` with an arbitrary offset.

```diff
diff --git a/drivers/android/binder.c b/drivers/android/binder.c
index 7cec5840cfcd..73ae3ced72fb 100644
--- a/drivers/android/binder.c
+++ b/drivers/android/binder.c
@@ -1626,10 +1632,16 @@ static size_t binder_get_object(struct binder_proc *proc,
       size_t object_size = 0;

       read_size = min_t(size_t, sizeof(*object), buffer->data_size - offset);
-       if (offset > buffer->data_size || read_size < sizeof(*hdr) ||
-           binder_alloc_copy_from_buffer(&proc->alloc, object, buffer,
-                                         offset, read_size))
+       if (offset > buffer->data_size || read_size < sizeof(*hdr))
               return 0;
+       if (u) {
+               if (copy_from_user(object, u + offset, read_size))
+                       return 0;
+       } else {
+               if (binder_alloc_copy_from_buffer(&proc->alloc, object, buffer,
+                                                 offset, read_size))
+                       return 0;
+       }

       /* Ok, now see if we read a complete object. */
       hdr = &object->hdr;
```

Prior to this commit, all calls to `binder_get_object()` were handled by `binder_alloc_copy_from_buffer()`, which calls `check_buffer()` to ensure the offset is 4-byte aligned. As such, an alignment check in `binder_get_object()` was unnecessary since it would be redundant. However, now that we are using `copy_from_user()`, no alignment check is performed on the offset, which could cause issues if parts of the Binder code expect a 4-byte aligned offset.

To make the code more robust, a check was added in commit `aaef73821a3b`:

```diff
diff --git a/drivers/android/binder.c b/drivers/android/binder.c
index bad28cf42010..dd6923d37931 100644
--- a/drivers/android/binder.c
+++ b/drivers/android/binder.c
@@ -1708,8 +1708,10 @@ static size_t binder_get_object(struct binder_proc *proc,
       size_t object_size = 0;

       read_size = min_t(size_t, sizeof(*object), buffer->data_size - offset);
-       if (offset > buffer->data_size || read_size < sizeof(*hdr))
+       if (offset > buffer->data_size || read_size < sizeof(*hdr) ||
+           !IS_ALIGNED(offset, sizeof(u32)))
               return 0;
+
       if (u) {
               if (copy_from_user(object, u + offset, read_size))
                       return 0;
```

Environment Setup

The vulnerability was introduced in kernel version 5.16.3 and fixed in 6.6.29. I built kernel version 6.6.28, which is just one patch level behind, meaning it still contains the unpatched vulnerability. Since this vulnerability is not Android-specific, I used an Ubuntu 20.04 image along with the vulnerable kernel.

```
root@ubuntu-syzkaller:~# uname -a
Linux ubuntu-syzkaller 6.6.28 #4 SMP PREEMPT_DYNAMIC
```

This is what I enabled in my .config for Binder IPC support:

```
CONFIG_ANDROID_BINDER_IPC=y
CONFIG_ANDROID_BINDERFS=y
CONFIG_ANDROID_BINDER_DEVICES="binder,hwbinder,vndbinder"
```

We can see that the Binder driver is correctly registered:

```
root@ubuntu-syzkaller:~# cat /proc/devices | grep binder
248 binder
```

All that's left for us now is to mount the binderfs, which will allow us to communicate directly with the driver and attempt to trigger the vulnerability.

```
sudo mkdir /mnt/binderfs
mount -t binder binder /mnt/binderfs
ln -s /mnt/binderfs/binder /dev/binder
```

Making sure everything works:

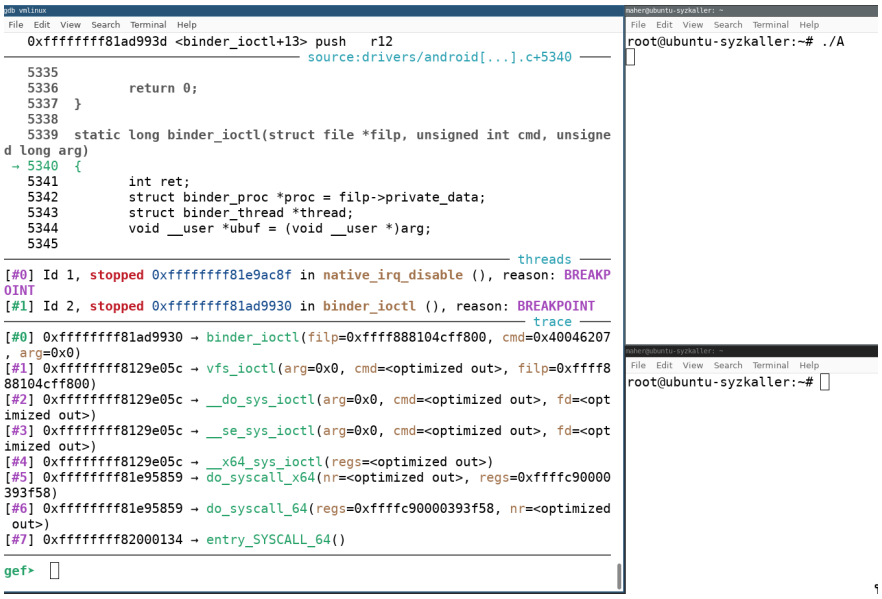

Figure 1: Hitting a breakpoint in `binder_ioctl` to make sure the setup is good.

Static/Dynamic Analysis

We know that the vulnerable function is `binder_get_object()`, so first of all, let's see what are the Xrefs to this function in the binder code:

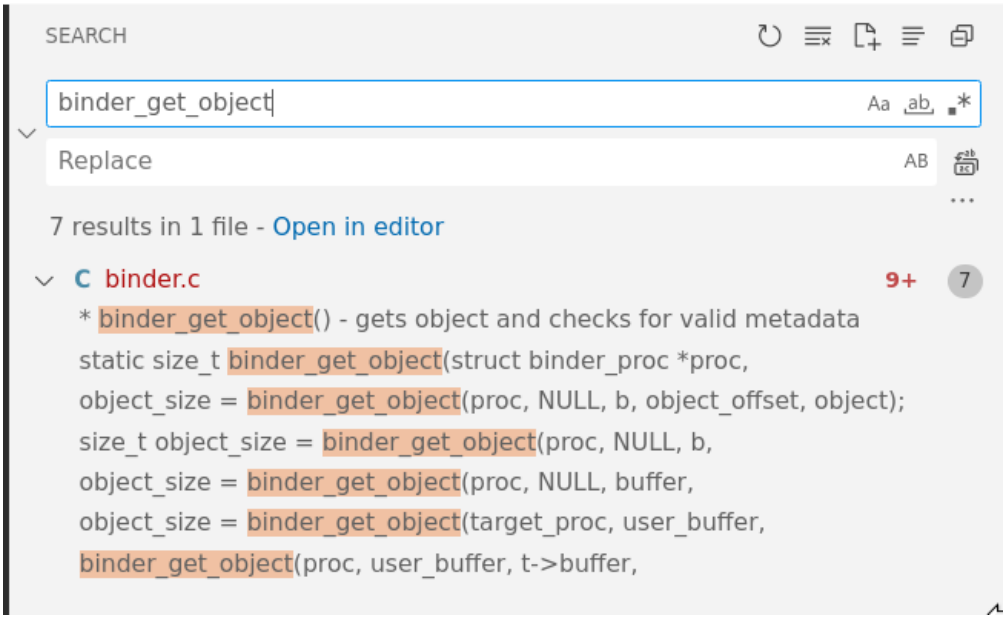

Figure 2: Xrefs to `binder_get_object()`

We can see that this function is called in 5 different places. What interests us more is having the sender's user pointer (`const void __user *u`) to be non-NULL, so we can enter `copy_from_user` and not parse the object using the buffer. This is what CVE-2024-26926 is about, and there are two references that pass a valid user pointer to our vulnerable function. Both entries belong to the same function, `binder_transaction()`, which is a core part of the Binder framework where the actual transaction of data between processes happens.

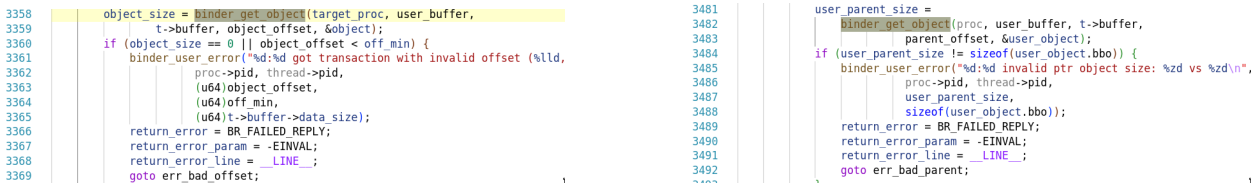

Figure 3: binder_get_object() entries

The `binder_get_object` function at line 3358 is used to parse and get the `struct binder_object object` on the stack.
The `binder_get_object` function at line 3482 is used to retrieve the file descriptor array (FDA) from the parent object.

Both seem like good candidates to reach with our PoC, but we cannot use the `binder_get_object` used to retrieve the FDA because its offset (parent_offset) is checked to be 4 bytes aligned at line 3464, like this:

`binder_validate_fixup() -> binder_alloc_copy_from_buffer() ->`

`binder_alloc_do_buffer_copy() -> check_buffer()`

```
3464            if (!binder_validate_fixup(target_proc, t->buffer,
3465                            off_start_offset,
3466                            parent_offset,
3467                            fda->parent_offset,
3468                            last_fixup_obj_off,
3469                            last_fixup_min_off)) {
3470                binder_user_error("%d:%d got transaction with out-of-o
3471                        proc->pid, thread->pid);
3472                return_error = BR_FAILED_REPLY;
3473                return_error_param = -EINVAL;
3474                return_error_line = __LINE__;
3475                goto err_bad_parent;
3476            }
```

Figure 4: Alignment check

So, we are only left with one call into `binder_get_object` with a controlled offset that cannot be aligned. The `object_offset` is parsed from our buffer and used directly into our target function. This `object_offset` is also tainting some other variables, but they don't seem to cause any problems.

```
3327        if (binder_alloc_copy_from_buffer(&target_proc->alloc,
3328                        &object_offset,
3329                        t->buffer,
3330                        buffer_offset,
3331                        sizeof(object_offset))) {…
3339
3340        /*
3341         * Copy the source user buffer up to the next object
3342         * that will be processed.
3343         */
3344        copy_size = object_offset - user_offset;
3345        if (copy_size && (user_offset > object_offset ||
3346                binder_alloc_copy_user_to_buffer(
3347                    &target_proc->alloc,
3348                    t->buffer, user_offset,
3349                    user_buffer + user_offset,
3350                    copy_size))) {…
3358        object_size = binder_get_object(target_proc, user_buffer,
3359                t->buffer, object_offset, &object);
```

Figure 5: Highlight of `object_offset` being passed into `binder_get_object()` at line 3327.

Now the goal is to have a small PoC to reach the vulnerable function with a non-aligned offset. It should involve two processes trying to communicate using the same handle, sending an object for parsing. The execution path should be like this:

`binder_ioctl() -> binder_ioctl_write_read() -> binder_thread_write() ->`

`binder_transaction() -> binder_get_object()`

Notice the offset having `0x9`, which is not 4-bytes aligned, and we will see whether that can cause any problems later.

```
gdb vmlinux
File  Edit  View  Search  Terminal  Help
   0xffffffff81acd756 <binder_get_object+6> push   rbx
   0xffffffff81acd757 <binder_get_object+7> mov    rax, QWORD PTR [rdx+0x40]
   0xffffffff81acd75b <binder_get_object+11> mov    rdx, rax
──────────────────────────────────────── source:drivers/android[...].c+1710 ────
   1705  {
   1706          size_t read_size;
   1707          struct binder_object_header *hdr;
   1708          size_t object_size = 0;
   1709
 → 1710          read_size = min_t(size_t, sizeof(*object), buffer->data_size - offset);
   1711          if (offset > buffer->data_size || read_size < sizeof(*hdr))
   1712                  return 0;
   1713
   1714          if (u) {
   1715                  if (copy_from_user(object, u + offset, read_size))
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, stopped 0xffffffff81e9ac8f in native_irq_disable (), reason: BREAKPOINT
[#1] Id 2, stopped 0xffffffff81acd750 in binder_get_object (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0xffffffff81acd750 → binder_get_object(proc=0xffff888100c83c00, u=0x7ffe03498fe0, buffer=0xffff888103b64c00,
offset=0x9, object=0xffffc90000337c08)
[#1] 0xffffffff81ad5d8a → binder_transaction(proc=0xffff888104e9f400, thread=0xffff888103a35200, tr=0xffffc9000033
7d08, reply=0x0, extra_buffers_size=<optimized out>)
[#2] 0xffffffff81ad8aad → binder_thread_write(proc=0xffff888104e9f400, thread=0xffff888103a35200, binder_buffer=0x
7ffe03498f90, size=<optimized out>, consumed=0xffffc90000337e60)
[#3] 0xffffffff81adad2b → binder_ioctl_write_read(thread=0xffff888103a35200, arg=0x7ffe03498f60, filp=0xffff888102
7ac700)
[#4] 0xffffffff81adad2b → binder_ioctl(filp=0xffff8881027ac700, cmd=0xc0306201, arg=0x7ffe03498f60)
[#5] 0xffffffff8129e05c → vfs_ioctl(arg=0x7ffe03498f60, cmd=<optimized out>, filp=0xffff8881027ac700)
[#6] 0xffffffff8129e05c → __do_sys_ioctl(arg=0x7ffe03498f60, cmd=<optimized out>, fd=<optimized out>)
[#7] 0xffffffff8129e05c → __se_sys_ioctl(arg=0x7ffe03498f60, cmd=<optimized out>, fd=<optimized out>)
[#8] 0xffffffff8129e05c → __x64_sys_ioctl(regs=<optimized out>)
[#9] 0xffffffff81e95859 → do_syscall_x64(nr=<optimized out>, regs=0xffffc90000337f58)
──────────────────────────────────────────────────────────────────────────────
gef➢

maher@ubuntu-syzkaller: ~
File  Edit  View  Search  Terminal  Help
root@ubuntu-syzkaller:~# ./A
Binder context manager set.

maher@ubuntu-syzkaller: ~
File  Edit  View  Search  Terminal  Help
root@ubuntu-syzkaller:~# ./B
```

Figure 6: Breakpoint hit in `binder_get_object()` with a misaligned offset.

In the function, this offset is being added to the userspace pointer provided, the object should be one of seven types. The check at line `binder.c:1744` is just making sure that the correct struct is being initialized (mitigating non-initialized memory use/disclosure). We don't want this function to return 0 to continue execution in `binder_transaction()`.

From now on, we will be following the `object_offset` variable as we continue in `binder_transaction()`.

At line `binder.c:3375`, `user_offset` is getting tainted by our non-aligned offset, but that doesn't matter at all. At line `binder.c:3378`, `off_min` variable is also getting tainted, but it's not very important as it is not causing problems. Now, we enter a switch case based on the object type; there are six cases and a default case.

I will try to show that the transaction will return with an error before anything damaging happens with the non-aligned offset (OOB read/write).

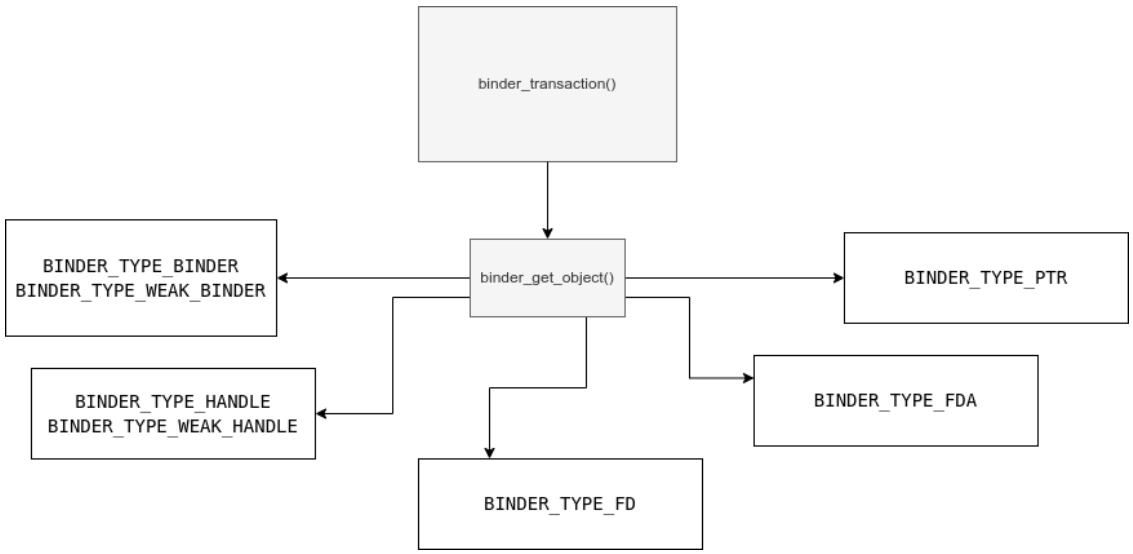

Figure 7: Object Deserialization Switch Case

1. For the cases where the type is either `BINDER_TYPE_BINDER`, `BINDER_TYPE_WEAK_BINDER`, `BINDER_TYPE_HANDLE`, `BINDER_TYPE_WEAK_HANDLE`, the `object_offset` is only used in the call to `binder_alloc_copy_to_buffer()` which will check our offset using `check_buffer()` and return with an error. In case of an error, the switch case will jump into `err_translate_failed`, which will perform cleaning and freeing of the buffer and related objects used for the transaction.

2. For `BINDER_TYPE_FD`, our non-aligned offset is tainting the `fd_offset` variable, and a call to `binder_translate_fd()` will not perform any checks and will link our fd fixup to the `t->fd_fixups` linked-list. It seems like this could be exploitable, but it is not because in `binder_apply_fd_fixups()` it will call `binder_alloc_copy_to_buffer()` which will invoke `check_buffer()`.

3. For `BINDER_TYPE_FDA`, `BINDER_TYPE_PTR`, it's similar; `object_offset` will only be used with the call into `binder_alloc_copy_to_buffer()` which will perform the correct checks.

4. If the type provided is not known, the default case will execute, logging an error message and executing the cleaning routine.

As a result, I believe that a misaligned `object_offset` will be caught and will not lead to an inconsistent state immediately. I don't see a way to cause problems with it.

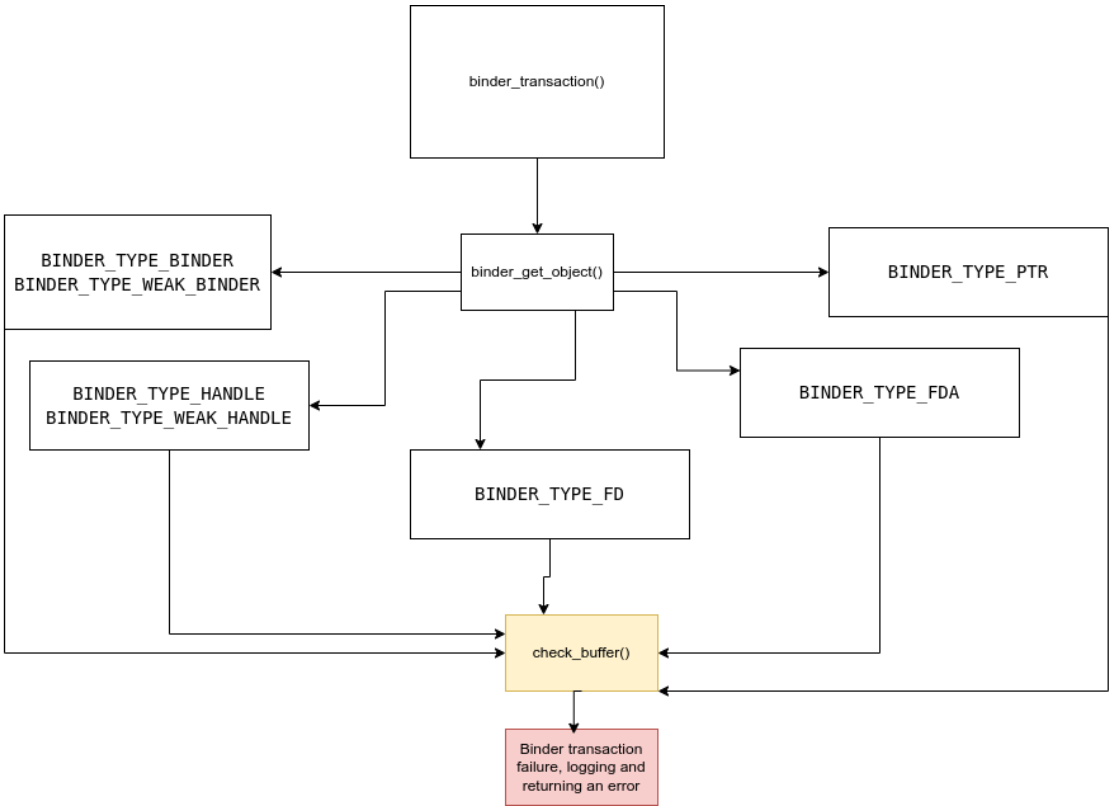

Figure 8: All Execution Paths are Leading to our Offset Being Checked

Conclusion

CVE-2024-26926 is a Linux kernel bug that is not currently exploitable, but enforcing alignment checks is a good measure to prevent future bugs. That justifies the low CVSS score assigned by security entities from Ubuntu, SUSE, and Amazon Linux Security Center.

Instructions

This is a PoC for hitting the vulnerable function with a non-aligned offset. To execute this PoC compile and run A and then B in another terminal.

Process A Code:

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <linux/android/binder.h>
#include <linux/android/binderfs.h>

#define BINDER_DEVICE "/dev/binder"
//#define BINDER_THREAD_EXIT _IO('b', 11)

int main() {
    int binder_fd = open(BINDER_DEVICE, O_RDWR);
    if (binder_fd < 0) {
        perror("Failed to open binder device");
        return EXIT_FAILURE;
    }

    void *mapped = mmap(NULL, 128 * 1024, PROT_READ , MAP_SHARED, binder_fd, 0);
    if (mapped == MAP_FAILED) {
        perror("Failed to mmap binder device");
        close(binder_fd);
        return EXIT_FAILURE;
    }

    if (ioctl(binder_fd, BINDER_SET_CONTEXT_MGR, 0) < 0) {
        perror("Failed to set context manager");
        munmap(mapped, 128 * 1024);
        close(binder_fd);
        return EXIT_FAILURE;
    }

    printf("Binder context manager set.\n");


    struct binder_write_read bwr = {0};
    struct binder_transaction_data txn = {0};

    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;
    bwr.read_size = sizeof(txn);
    bwr.read_consumed = 0;
    bwr.read_buffer = (uintptr_t)&txn;

    if (ioctl(binder_fd, BINDER_WRITE_READ, &bwr) < 0) {
        perror("Binder write_read ioctl failed");
    }

    munmap(mapped, 128 * 1024);
    close(binder_fd);
    return EXIT_SUCCESS;
}
```

Listing 1: Process A - Binder Transaction Setup

Process B Code:

```c
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <sys/wait.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <linux/android/binder.h>
#include <linux/android/binderfs.h>

#define BINDER_DEVICE "/dev/binder"
#define BINDER_WRITE_READ _IOWR('b', 1, struct binder_write_read)

int main() {
    int binder_fd = open(BINDER_DEVICE, O_RDWR);
    if (binder_fd < 0) {
        perror("Failed to open binder device");
        return EXIT_FAILURE;
    }

    struct {
        uint32_t cmd;
        struct binder_transaction_data txn;
    } __attribute__((packed)) write_data = {0};

    write_data.cmd = BC_TRANSACTION;
    write_data.txn.target.handle = 0;
    write_data.txn.cookie = 0;
    write_data.txn.code = 123;
    write_data.txn.flags = TF_ONE_WAY;
    write_data.txn.sender_euid = getuid();
    write_data.txn.data_size = 16+8+0x10-0x7;
    write_data.txn.offsets_size = 8;


    uint64_t data[10] = {};
    data[0] = 0x09;
    data[1] = 0x73622a8500;
    write_data.txn.data.ptr.buffer = (uintptr_t)&data;
    write_data.txn.data.ptr.offsets = (uintptr_t)&data;

    struct binder_write_read bwr;
    bwr.write_size = sizeof(write_data);
    bwr.write_consumed = 0;
    bwr.write_buffer = (uintptr_t)&write_data;
    bwr.read_size = 0;
    bwr.read_consumed = 0;
    bwr.read_buffer = 0;

    if (ioctl(binder_fd, BINDER_WRITE_READ, &bwr) < 0) {
        perror("Binder write_read ioctl failed");
        close(binder_fd);
        return EXIT_FAILURE;
    }

    close(binder_fd);
    printf("Binder transaction sent successfully!\n");
    return EXIT_SUCCESS;
}
```

Listing 2: Process B - Sending Binder Transaction