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

Di Shen a.k.a. Retme (@returnsme)

Keen Lab of Tencent

whoami

- Di Shen a.k.a. Retme (@returnsme)
- Member of Keen Lab
- Android Kernel vulnerability hunting and exploitation since 2014
- Aim: to work out universal rooting exploit for Android
- Trophy:
  - CVE-2016-6787 & CVE-2017-0403 (kernel/events/core.c)
  - CVE-2015-1805 (fs/pipe.c) ’s first working exploit
  - CVE-2015-4421,4422 (Huawei TrustZone)
  - KNOX Bypassing on Samsung Galaxy S7 (BHUSA 17’)
  - Exploiting Wireless Extension for all common Wi-Fi chipsets (BHEU 16’)
  - And more To Be Announced in the future
- Available on https://github.com/retme7/My-Slides

Agenda

- Rooting Android: Current situation
- Overview of exploiting UAF in kernel
  - Conventional approach
  - Afterwards: Gain root
- The Unconventional UAFs
  - Implementation of perf system
  - Exploiting CVE-2017-0403
  - Exploiting CVE-2016-6787
- Conclusion

Rooting Android: Current situation

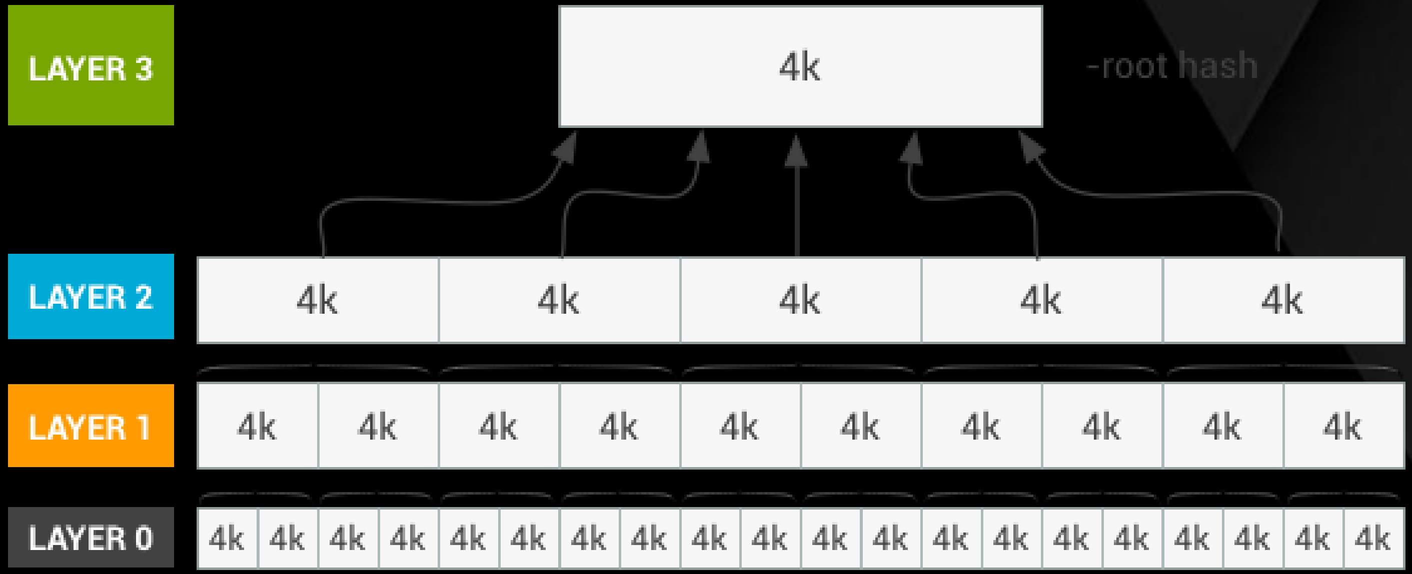
- Universal exploitable vulnerability is rare
  - Available attack surface:
    - Generic Linux syscalls
    - Android universal drivers like Binder, ION, Ashmem

Rooting Android: Current situation

- Enforced SELinux policy
  - Most of device drivers are inaccessible
  - Many syscalls are not reachable from untrusted Application
  - Sockets ioctl commands are partially restricted

Rooting Android: Current situation

- Verified Boot through dm-verity kernel feature
  - The gained root privilege is nonpersistent



Rooting Android: Future challenges

- Privileged Access Never (PAN)
- KASLR
- Pointer Authentication

Overview of exploiting UAF in kernel

- An easily exploitable UAF bug normally has following features:
  - Has a function pointer in freed object
  - Attacker has plenty of time to refill the freed object.

Conventional approach to UAF exploitation

struct socket (freed)

ops->ioctl(…)

```
int inet_ioctl(struct socket *sock, un
{
	struct sock *sk = sock->sk;
	int err = 0;
```

struct socket (refilled)

ops->ioctl(…)

JOP gadgets

- Free the victim object
- Refill the object with malformed data by heap spraying or ret2dir
- Let the function pointer point to ROP/JOP gadgets in kernel
- Ask kernel reference this function pointer to achieve arbitrary kernel code execution

ioctl(sockfd,…) → kernel_sock_ioctl() → JOP gadgets

Afterwards, from code execution to root

Arbitrary kernel code execution → Overwrite process's addr_limit → Arbitrary kernel memory overwriting → Overwrite uid, security id, selinux_enforcing

However…

- Not every UAF bug in kernel is so that idealized
- More unconventional situation to deal with…
  - The victim object may don’t have a function pointer
  - The kernel may crash soon after UAF triggered
  - The attacker may cannot fully controlled the freed object

The unconventional UAFs I found

- All found in sys_perf_event_open()
  - Perf system is pretty buggy
  - Reachable by application last year
  - But now it's restricted by a feature called "perf_event_paranoid"

The unconventional UAFs I found

- CVE-2017-0403
  - Ever affected all devices shipped with 3.10 or earlier Linux kernel
  - More than 14 million users of KingRoot gain root privilege on their smart phones
- CVE-2016-6787
  - Ever affected all Qualcomm-based devices. (Only Qucalcomm enabled hardware perf event…)
  - A part of my exploit chain to bypass Samsung KNOX 2.6

sys_perf_event_open()

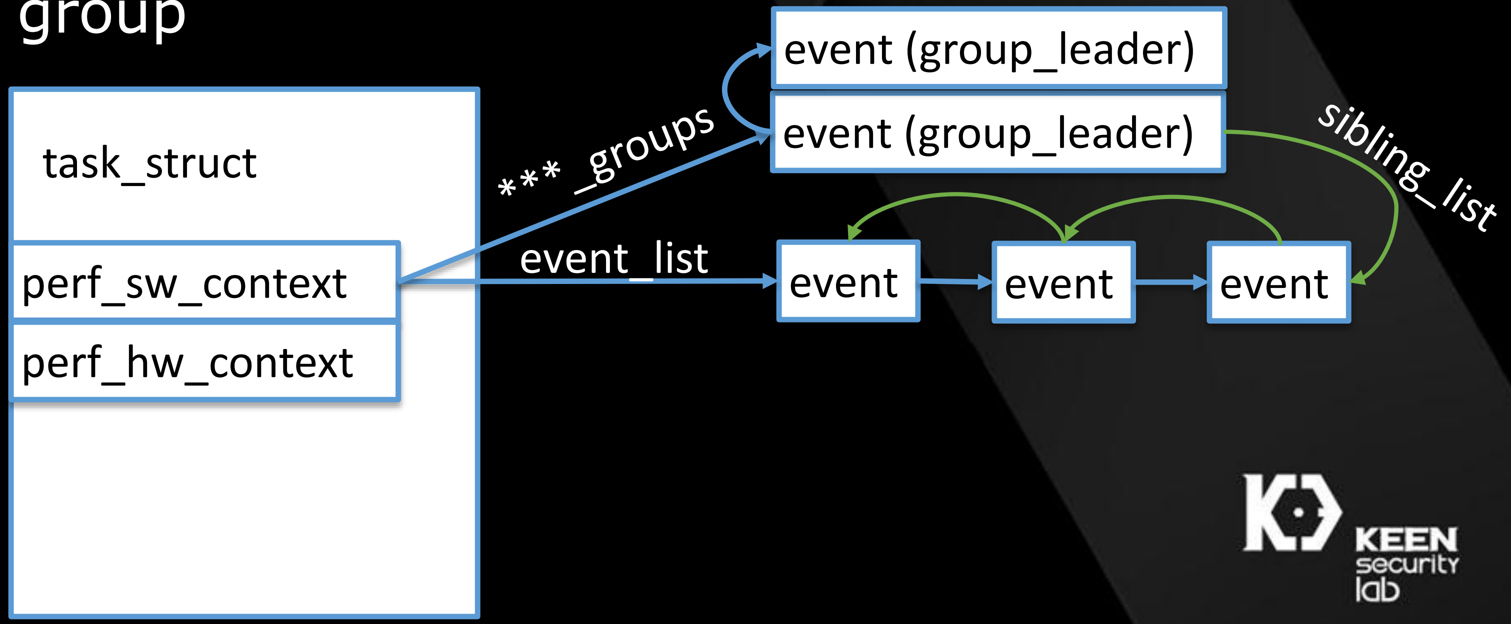
- Will create a perf_event
- Input: perf_event_attr
  - A description of what kind of performance event you need
- Input: group_fd (optional)
  - Specify the group leader of new perf_event
- Return the fd of perf_event to user space

```c
/**
 * sys_perf_event_open - open a performance event, associate it to a task/cpu
 *
 * @attr_uptr:  event_id type attributes for monitoring/sampling
 * @pid:        target pid
 * @cpu:        target cpu
 * @group_fd:       group leader event fd
 */
SYSCALL_DEFINE5(perf_event_open,
        struct perf_event_attr __user *, attr_uptr,
        pid_t, pid, int, cpu, int, group_fd, unsigned long, flags)
{
    struct perf_event *group_leader = NULL, *output_event = NULL;
    struct perf_event *event, *sibling;
    struct perf_event_attr attr;
    struct perf_event_context *ctx;
    struct file *event_file = NULL;
    struct fd group = {NULL, 0};
    struct task_struct *task = NULL;
    struct pmu *pmu;
    int event_fd;
    int move_group = 0;
    int err;
    int f_flags = O_RDWR;
```

Key kernel objects in perf system

Tencent

- perf_event
  - A performance event which is registered by user
- perf_event_context
  - The container of all perf events created in one process
  - Each process has two contexts, one for software events, other one for hardware events
- Perf group and group leader
  - Multiple events can form a group
  - One event is the leader



KEEN security lab

move_group

- Happens when user try to create a hardware event in pure software group.

```
if (group_leader &&
    (is_software_event(event) != is_software_event(group_leader))) {
    if (is_software_event(event)) {
        /*
         * If event and group_leader are not both a software
         * event, and event is, then group leader is not.
         *
         * Allow the addition of software events to !software
         * groups, this is safe because software events never
         * fail to schedule.
         */
        pmu = group_leader->pmu;
    } else if (is_software_event(group_leader) &&
           (group_leader->group_flags & PERF_GROUP_SOFTWARE)) {
        /*
         * In case the group is a pure software group, and we
         * try to add a hardware event, move the whole group to
         * the hardware context.
         */
        move_group = 1;
    }
}
```

CVE-2016-6787

Remove the group_leader from origin software context and then install it to hardware context

```c
if (move_group) {
    struct perf_event_context *gctx = group_leader->ctx;

    mutex_lock(&gctx->mutex);
    perf_remove_from_context(group_leader, false);

    /*
     * Removing from the context ends up with disabled
     * event. What we want here is event in the initial
     * startup state, ready to be add into new context.
     */
    perf_event__state_init(group_leader);
    list_for_each_entry(sibling, &group_leader->sibling_list,
                        group_entry) {
        perf_remove_from_context(sibling, false);
        perf_event__state_init(sibling);
        put_ctx(gctx);
    }
    mutex_unlock(&gctx->mutex);
    put_ctx(gctx);
}
```

```c
if (move_group) {
    synchronize_rcu();
    perf_install_in_context(ctx, group_leader, group_leader->cpu);
    get_ctx(ctx);
    list_for_each_entry(sibling, &group_leader->sibling_list,
                        group_entry) {
        perf_install_in_context(ctx, sibling, sibling->cpu);
        get_ctx(ctx);
    }
}
```

Remove every event from software context, and then install it to new hardware context

'move_group' leads to reducing context's refcont by one

CVE-2016-6787

- move_group ignored the concurrency issues
- UAF happens due to race condition
- Attacker trigger the move_group on same group leader simultaneously ,
- The ref count of group_leader->ctx may be reduced to zero
- task_struct->perf_event_ctxp[perf_sw_context] will be freed accidently

```
static void put_ctx(struct perf_event_context *ctx)
{
    if (atomic_dec_and_test(&ctx->refcount)) {
        if (ctx->parent_ctx)
            put_ctx(ctx->parent_ctx);
        if (ctx->task)
            put_task_struct(ctx->task);
        kfree_rcu(ctx, rcu_head);
    }
}
```

The object is freed

Free perf_event_context (PoC)

Main thread	Sub thread-1	Sub thread-2	
			ctx->refcount = 1
Create a software group_leader			
	Create a hardware perf_event	Create a hardware perf_event	ctx->refcount = 2
	move_group, put_ctx()	move_group, put_ctx()	ctx->refcount = 0
kfree_rcu(perf_event_context)			

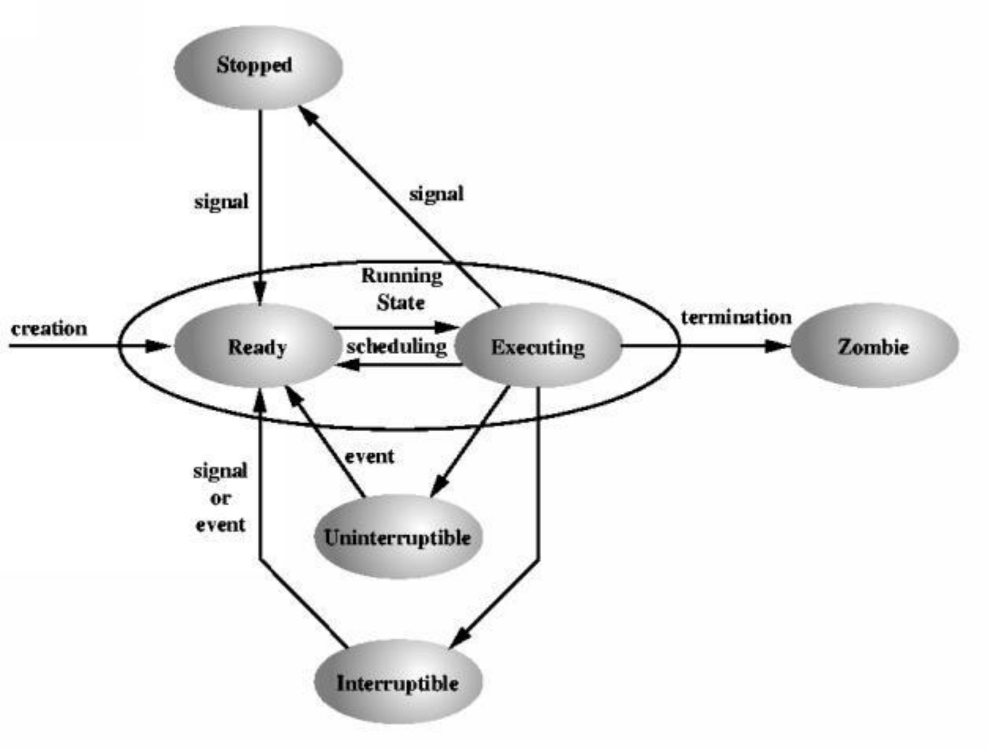
Kernel crashed instantly

- Kernel crashed soon after we freed the perf_event_context
- Thread scheduler need to dereference this object pointer consecutively
- We don‘t have plenty of time to refill the object ☹

```
<1>[67760.887267] Unable to handle kernel paging request at virtual address bfb8c160
<1>[67760.887314] pgd = ffffffc0475b1000
<1>[67760.887333] [bfb8c160] *pgd=0000000000000000
<0>[67760.887374] Internal error: Oops: 96000005 [#1] PREEMPT SMP
<4>[67760.887412] CPU: 1 PID: 10819 Comm: qoc_report Tainted: G        W    3.10.73-ga5cec12 #1
<4>[67760.887436] task: ffffffc03cbe8ac0 ti: ffffffc024438000 task.ti: ffffffc024438000
<4>[67760.887481] PC is at perf_event_context_sched_in.isra.72+0x30/0xbc
<4>[67760.887505] LR is at perf_event_context_sched_in.isra.72+0x20/0xbc
<4>[67760.887527] pc : [<ffffffc0002b3c44>] lr : [<ffffffc0002b3c34>] pstate: 60000185
<4>[67760.887545] sp : ffffffc02443bae0
<4>[67760.887562] x29: ffffffc02443bae0 x28: 0000000000000a60
<4>[67760.887603] x27: ffffffc0c1198e66 x26: ffffffc0c1198e6c
<4>[67760.887645] x25: ffffffc000e01000 x24: ffffffc024438000
<4>[67760.887687] x23: 0000000000000000 x22: ffffffc00e9a6b80
<4>[67760.887731] x21: ffffffc03cbe8ac0 x20: 00000000bfb8c000
<4>[67760.887773] x19: ffffffc0acbe5400 x18: ffffffc00e9f3d28
<4>[67760.887816] x17: 0000000000000000 x16: 0000000000000000
<4>[67760.887858] x15: 0000000000000000 x14: 0ffffffffffffffe
<4>[67760.887901] x13: 0000000000000030 x12: 0101010101010101
<4>[67760.887944] x11: 7f7f7f7f7f7f7f7f x10: feff676273687672
<4>[67760.887987] x9 : ffffffc02443bb90 x8 : ffffffc03cbe9020
<4>[67760.888029] x7 : 0000000000000180 x6 : ffffffc00160f850
<4>[67760.888072] x5 : ffffffc0c1197db8 x4 : ffffffc00110a70e
<4>[67760.888114] x3 : 0000000000000000 x2 : 0000007f848ab098
<4>[67760.888157] x1 : ffffffc0019d93b8 x0 : 00000000bfb8c000
<4>[67760.888198]
<0>[67760.888219] Process qoc_report (pid: 10819, stack limit = 0xffffffc024438058)
<4>[67760.888237] Call trace:
<4>[67760.888267] [<ffffffc0002b3c44>] perf_event_context_sched_in.isra.72+0x30/0xbc
<4>[67760.888295] [<ffffffc0002b3d04>] __perf_event_task_sched_in+0x34/0x144
<4>[67760.888328] [<ffffffc00024925c>] finish_task_switch+0x104/0x120
<4>[67760.888363] [<ffffffc000c83a04>] __schedule+0x928/0xb38
<4>[67760.888390] [<ffffffc000c83c78>] schedule+0x64/0x70
<4>[67760.888416] [<ffffffc000c82870>] do_nanosleep+0x7c/0x100
<4>[67760.888448] [<ffffffc0002418b8>] hrtimer_nanosleep+0x94/0x108
<4>[67760.888473] [<ffffffc0002419ac>] SyS_nanosleep+0x80/0x98
<0>[67760.888502] Code: f0004ea1 f9459021 f8605820 8b000294 (f940b280)
<4>[67760.888527] ---[ end trace 278ee17304bbc989 ]---
```

Solution: freeze thread after free

- Keep thread scheduler away from me
- Switch the status of attacker's thread from running to (un)interruptible
- The thread will be frozen and kernel won't crash as soon as perf_event_context freed



How to freeze a thread from user land?

- Sleep() ? Not working
- Use futex_wait_queue_me()

switch to interruptible

freezable_schedule()

```
static void futex_wait_queue_me(struct futex_hash_bucket *hb, struct futex_q *q,
                struct hrtimer_sleeper *timeout)
{
    /*
     * The task state is guaranteed to be set before another task can
     * wake it. set_current_state() is implemented using set_mb() and
     * queue_me() calls spin_unlock() upon completion, both serializing
     * access to the hash list and forcing another memory barrier.
     */
    set_current_state(TASK_INTERRUPTIBLE);
    queue_me(q, hb);

    /* Arm the timer */
    if (timeout) {
        hrtimer_start_expires(&timeout->timer, HRTIMER_MODE_ABS);
        if (!hrtimer_active(&timeout->timer))
            timeout->task = NULL;
    }

    /*
     * If we have been removed from the hash list, then another task
     * has tried to wake us, and we can skip the call to schedule().
     */
    if (likely(!plist_node_empty(&q->list))) {
        /*
         * If the timer has already expired, current will already be
         * flagged for rescheduling. Only call schedule if there
         * is no timeout, or if it has yet to expire.
         */
        if (!timeout || timeout->task)
            freezable_schedule();
    }
    __set_current_state(TASK_RUNNING);
} « end futex_wait_queue_me »
```

Main thread | Sub thread-1 | Sub thread-2

Phase 1

Create a software group_leader

Create a hardware perf_event

Create a hardware perf_event

move_group, put_ctx()

move_group, put_ctx()

kfree_rcu(perf_event_context)

futex_wait_queue_me()

Phase 2

Spraying the heap by using ‘ret2dir’ trick, fill a malformed perf_event_context{} in every 1024 bytes

Phase 3

Use futex_wake() wake up main thread

Phase 4

schedule()

finish_task_switch()

perf_event_context_sched_in()

ctx->pmu->pmu_disable()

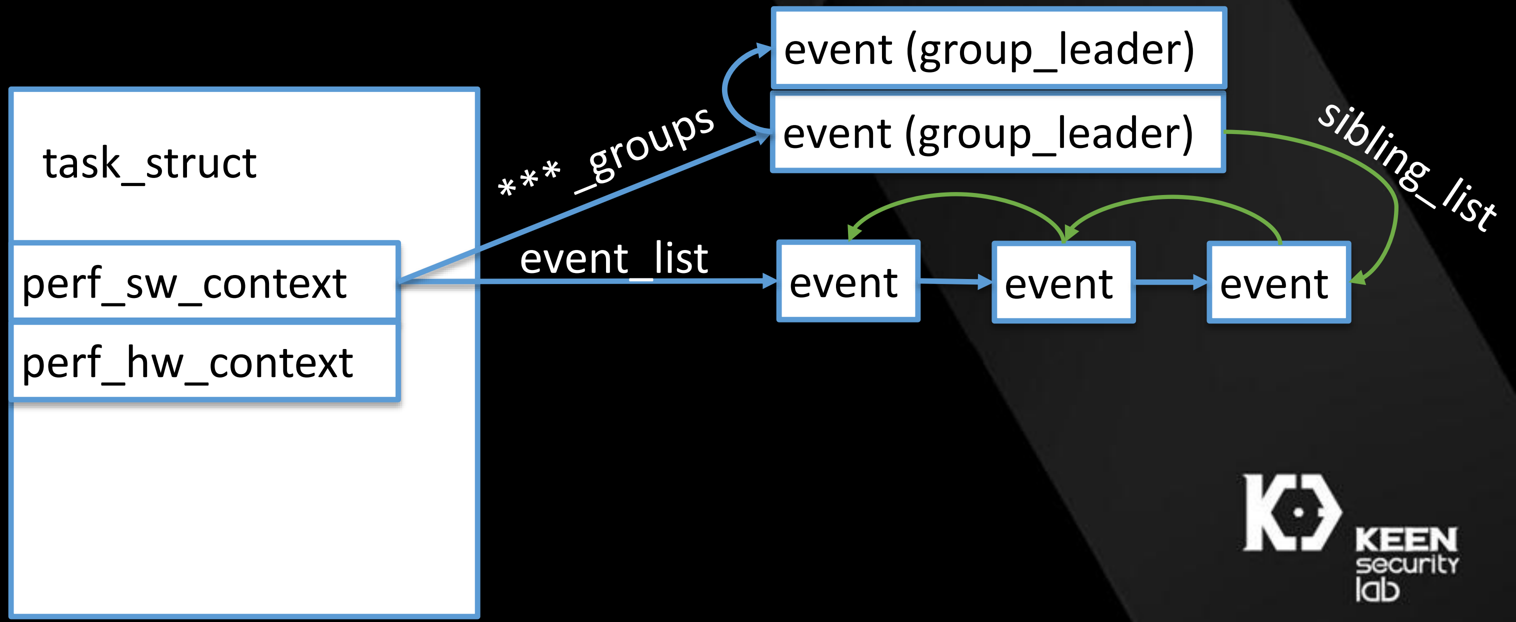
Code flow controlled

A brief summary of CVE-2016-6787

- Easy to win the race, and trigger the bug
- **Hard to refill the freed object (no time)**
- Easy to control the code flow (corrupted object has function pointer)
- Proposed an approach to freezing thread to gain more time to refill object

Review: relationship between perf event, group and group leader

- Group leader has a sibling_list
- sibling_list is a list of perf events which belongs this group



CVE-2017-0403 (PoC)

- Create a perf event as ‘A’
- Create another perf event as ‘B’, specify ‘A’ as its group leader
- Free ‘A’ , the group leader
- Free ‘B’, a sibling of group ←---- UAF happens here

Root cause

- Now group leader ‘A’ is freed
- Kernel doesn’t empty its sibling list
- Leads to leaving a dangling pointer in sibling’s event->group_entry

```c
static void perf_group_detach(struct perf_event *event)
{
	struct perf_event *sibling, *tmp;
	struct list_head *list = NULL;

	/*
	 * We can have double detach due to exit/hot-unplug + close.
	 */
	if (!(event->attach_state & PERF_ATTACH_GROUP))
		return;

	event->attach_state &= ~PERF_ATTACH_GROUP;

	/*
	 * If this is a sibling, remove it from its group.
	 */
	if (event->group_leader != event) {
		list_del_init(&event->group_entry);
		event->group_leader->nr_siblings--;
		goto out;
	}

	if (!list_empty(&event->group_entry))
		list = &event->group_entry;

	/*
	 * If this was a group event with sibling events then
	 * upgrade the siblings to singleton events by adding them
	 * to whatever list we are on.
	 */
	list_for_each_entry_safe(sibling, tmp, &event->sibling_list, group_entry) {
		if (list)
			list_move_tail(&sibling->group_entry, list);
		sibling->group_leader = sibling;
```

Root cause

- Later on the sibling ‘B’ is freed
- list_del_event()
- list_del_init(&event->group_entry);
- overwrite a pointer to the freed group leader.

```
static void perf_group_detach(struct perf_event *event)
{
        struct perf_event *sibling, *tmp;
        struct list_head *list = NULL;

        /*
         * We can have double detach due to exit/hot-unpl
         */
        if (!(event->attach_state & PERF_ATTACH_GROUP))
                return;

        event->attach_state &= ~PERF_ATTACH_GROUP;

        /*
         * If this is a sibling, remove it from its group
         */
        if (event->group_leader != event) {
                list_del_init(&event->group_entry);
                event->group_leader->nr_siblings--;
                goto out;
        }
```

KEEN security lab

- SLUB poison information
- 0xfffffc00fc2b1a0 is overwritten to (group_leader+ 0x20)

```
<3>[  238.517738] ==========================================================================
<3>[  238.517780] BUG kmalloc-1024 (Tainted: G        W    ): Poison overwritten
<3>[  238.517799] --------------------------------------------------------------------------
<3>[  238.517799]
<4>[  238.517827] Disabling lock debugging due to kernel taint
<3>[  238.517856] INFO: 0xffffffc00fc2b1a0-0xffffffc00fc2b1af. First byte 0xa0 instead of 0x6
<3>[  238.517915] INFO: Allocated in perf_event_alloc+0x68/0x3e4 age=34 cpu=2 pid=17548
<3>[  238.517953]       alloc_debug_processing+0xc8/0x16c
<3>[  238.517990]       __slab_alloc.isra.20.constprop.27+0x27c/0x2dc
<3>[  238.518023]       kmem_cache_alloc_trace+0x74/0x1c8
<3>[  238.518061]       perf_event_alloc+0x64/0x3e4
<3>[  238.518102]       SyS_perf_event_open+0x584/0x9b8
<3>[  238.518137]       cpu_switch_to+0x48/0x4c
<3>[  238.518178] INFO: Freed in free_event_rcu+0x28/0x34 age=29 cpu=2 pid=16
<3>[  238.518206]       free_debug_processing+0x204/0x2ac
<3>[  238.518231]       __slab_free+0x1b8/0x2cc
<3>[  238.518257]       kfree+0x218/0x220
<3>[  238.518283]       free_event_rcu+0x24/0x34
<3>[  238.518314]       rcu_process_callbacks+0x474/0x85c
<3>[  238.518342]       __do_softirq+0x154/0x288
<3>[  238.518368]       run_ksoftirqd+0x2c/0x54
<3>[  238.518397]       smpboot_thread_fn+0x280/0x288
<3>[  238.518422]       kthread+0xb0/0xc4
<3>[  238.518449]       ret_from_fork+0xc/0x50
<3>[  238.518476] INFO: Slab 0xffffffbc009d9900 objects=23 used=22 fp=0xffffffc00fc2d800 flag
<3>[  238.518500] INFO: Object 0xffffffc00fc2b180 @offset=12672 fp=0xffffffc00fc2e300
<3>[  238.518500]
<3>[  238.518536] Bytes b4 ffffffc00fc2b170: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a
<3>[  238.518563] Object ffffffc00fc2b180: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518589] Object ffffffc00fc2b190: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518614] Object ffffffc00fc2b1a0: a0 b1 c2 0f c0 ff ff ff a0 b1 c2 0f c0 ff ff ff  .
<3>[  238.518641] Object ffffffc00fc2b1b0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518667] Object ffffffc00fc2b1c0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518693] Object ffffffc00fc2b1d0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518719] Object ffffffc00fc2b1e0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518745] Object ffffffc00fc2b1f0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518771] Object ffffffc00fc2b200: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518797] Object ffffffc00fc2b210: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518824] Object ffffffc00fc2b220: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518850] Object ffffffc00fc2b230: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518876] Object ffffffc00fc2b240: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518903] Object ffffffc00fc2b250: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
<3>[  238.518929] Object ffffffc00fc2b260: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  k
```

The unconventional scenario

- The only thing I can do is overwriting the freed object as following

```
*(size_t*)(freed_object + 0x20) = (freed_object + 0x20)
```

Pipe subsystem in Linux

- readv() & writev(): read/write multiple buffers through pipe
- Use an array of struct iovec{iov_base,iov_len} to describe user buffers
- When no contents available from the write end, readv() may block in kernel
- Then an array of struct iovec{} may stay in kernel's heap

Compromise pipe system

- Call readv()
- rw_copy_check_uvector() confirm every iov_base must points to userland space.
- An array of struct iovec{} now is in heap. Nothing comes from the write end of pipe, so readv() block.

iov_base	iov_len	iov_base	iov_len	iov_base	iov_len

- If you can somehow overwrite the iovec{}, modify the iov_base to a kernel address. Emmm…

kernel_addr	iov_len	kernel_addr	iov_len	kernel_addr	iov_len

Compromise pipe system

Tencent

- Now write something to another end of pipe
- pipe_iov_copy_to_user() won't check the iov_base again.
- Buffers you wrote to pipe will be copied to **the kernel address**

```
static int
pipe_iov_copy_to_user(struct iovec *iov, const void *from, unsigned long len,
                      int atomic)
{
        unsigned long copy;

        while (len > 0) {
                while (!iov->iov_len)
                        iov++;
                copy = min_t(unsigned long, len, iov->iov_len);

                if (atomic) {
                        if (__copy_to_user_inatomic(iov->iov_base, from, copy
                                return -EFAULT;
                } else {
                        if (copy_to_user(iov->iov_base, from, copy))
                                return -EFAULT;
                }
                from += copy;
                len -= copy;
                iov->iov_base += copy;
                iov->iov_len -= copy;
        }
        return 0;
}
```

Solution: convert UAF to arbitrary R/W

Tencent

1. ↓ the 1st freed object, address is A ↓the 2nd freed object, address is B = A + 0x400

Freed	Data	Freed	Data	Freed	....	Freed	Data	Freed	Data	....

2. Use iovec to spray the heap

base	len	base	len	base	....	base	len	base	len	....

3. Trigger UAF, write two 8-bytes value "A+0x20" to address = A+0x20

base	len	A + 0x20	A+0x20	base	....	base	len	base	len	....

4. Write a buffer to pipe , the buffer will be copied to (A + 0x20)

base	len	KADDR	8	KADDR	....	KADDR	8	KADDR	8	....

5. Write a buffer to pipe again , it will be copied to **KADDR**

KADDR can be any address value, we achieved arbitrary kernel memory overwriting

KEEN security lab

A brief summary of CVE-2017-0403

- Attacker lost the file descriptor of freed object
- Cannot achieve code execution via refilling object's function pointer
- Only be able to write the address value of freed object twice to freed object
- Proposed a new approach: compromising pipe system

Conclusion

- Most UAF bugs looks not exploitable, but there may be another way
- No idea? Put it down for a while, but do not let it go…
- Be familiar with kernel’s source code, kernel’s own feature may help your exploitation (e.g. pipe for CVE-2017-0403)

Tencent

KEEN security lab