# Android ION Hazard: the Curse of Customizable Memory Management System

Hang Zhang, Dongdong She, Zhiyun Qian
University of California, Riverside
hang@cs.ucr.edu, dshe002@ucr.edu, zhiyunq@cs.ucr.edu

## ABSTRACT

ION is a unified memory management interface for Android that is widely used on virtually all ARM based Android devices. ION attempts to achieve several ambitious goals that have not been simultaneously achieved before (not even on Linux). Different from managing regular memory in the system, ION is designed to share and manage memory with special constraints, *e.g.,* physically contiguous memory. Despite the great flexibility and performance benefits offered, such a critical subsystem, as we discover, unfortunately has flawed security assumptions and designs.

In this paper, we systematically analyze ION related vulnerabilities from conceptual root causes to detailed implementation decisions. Since ION is often customized heavily for different Android devices, the specific vulnerabilities often manifest themselves differently. By conducting a range of runtime testing as well as static analysis, we are able to uncover a large number of serious vulnerabilities on the latest Android devices (*e.g.,* Nexus 6P running Android 6.0 and 7.0 preview) such as denial-of-service and dumping memory from the system and arbitrary applications (*e.g.,* email content, passwords). Finally, we offer suggestions on how to redesign the ION subsystem to eliminate these flaws. We believe that the lessons learned can help guide the future design of similar memory management subsystems.

## 1. INTRODUCTION

Android operating system has gained tremendous popularity in the past few years thanks to the huge vendor support behind it. Unlike iOS that runs on only Apple-assembled hardware, Android is open source and encourages other vendors to build smartphones using it. This model works well as vendors do not need to build a new OS from scratch, and they can still heavily customize the phones to differentiate themselves on the market. The customization happens at all layers including hardware, OS, and applications. Major vendors such as Samsung, HTC, and Huawei all perform customizations to attract customers with features like better screens, audio, and even security [29]. While such customization itself is encouraged, it has been shown that the process of customization at the software layer often introduces security vulnerabilities [28, 29, 27].

In this study, we investigate an important OS subsystem, called ION, that is commonly customized to satisfy different requirements from the underlying hardware devices. ION [8] is a unified memory management interface widely used on ARM based Android platforms. First introduced by Google in Android 4.0, it was initially designed to replace previous fragmented interfaces originated from System-on-Chip (SoC) vendors [8]. Its main goal is to support the special requirements set by hardware devices such as the GPU and camera. For instance, some devices require physically contiguous memory to operate and some require certain cache coherency protocol for DMA to function correctly. To satisfy such requirements, on a given Android phone, ION is customized with a set of pre-configured memory heaps for the underlying hardware devices. Even though AOSP provides a set of pre-defined heap types and implementations of heap allocation and management, customization is commonplace for performance tuning and other purposes (as we will show in the paper). In addition, for hardware devices not covered by AOSP, vendors often need to define new heap types as well as provide their own implementations of heap allocation and management.

Unfortunately, the framework for supporting such customization is not well thought out regarding its security implications. For instance, we discover that the lack of fine-grained access control to individual memory heaps can easily cause denial-of-service of specific system services or the entire OS. Moreover, its buffer sharing capability exposes different types of kernel memory to user space without being screened carefully for security consequences. To demonstrate the seriousness of the identified vulnerabilities, attack demos and analysis can be found on our project website [1]. In this paper, we make three main contributions:

- We systematically analyze the security properties from the design and implementation of ION, and reveal two major security flaws that lead to many vulnerabilitis and corresponding exploits, which are already reported to and confirmed by Google.
- To detect specific vulnerability instances, we develop both a runtime testing procedure and a novel static taint analysis tool that help uncover vulnerabilities on newest flagship models like Nexus 6P and Samsung S7 running Android 6.0 and 7.0 preview (latest at the time of writing).
- By analyzing the root causes of the problem, we propose

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*CCS'16, October 24-28, 2016, Vienna, Austria*
© 2016 ACM. ISBN 978-1-4503-4139-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978320

| Types | Instance | Page source | Is contiguous |
|---|---|---|---|
| SYSTEM | Nexus 6P:system | alloc_pages() of buddy allocator | n |
| SYSTEM_CONTIG | Nexus 6P:kmalloc | alloc_pages() of buddy allocator or kmalloc() | y |
| CARVEOUT | Samsung S7:camera | preserved memory region, not reusable | y |
| CMA | Nexus 6P:qsecom | preserved memory region, reusable | y |
| SECURE_CMA | Nexus 6P:mm | preserved memory region, reusable | y |
| CHUNK | N/A | preserved memory region, not reusable | y |
| REMOVED | N/A | preserved memory region, not reusable | y |
| EXYNOS | Samsung S4:exynos_noncontig_heap | alloc_pages() of buddy allocator | n |
| EXYNOS_CONTIG | Samsung S4:exynos_contig_heap | preserved memory region, reusable | y |
| CPUDRAW | Huawei Mate8:cpudraw_heap | preserved memory region, not reusable | y |

[1] N/A means that we have not observed actual instances of the heap type
[2] Even the same type can have different implementations on different devices

Table 1: ION heap types and instances

alternative designs that preserve the ION functionality while eliminating its security flaws. We believe the lessons learned can shed light on future designs of customizable and extensible memory management system.

The remaining part of the paper will be organized as following: §2 will briefly introduce some ION-related background knowledge, §3 will give a thorough analysis of ION related vulnerabilities, §4 will detail our methodology to systematically identify the vulnerabilities on a wide range of Android devices, §5 will summarize the vulnerabilities we find on various devices and evaluate the effectiveness of our methodology, §6 will demonstrate our actual exploitations against ION related vulnerabilities on some representative devices. In §7, we discuss possible defense against the vulnerabilities. §8 discusses the related works and §9 will conclude the paper.

## 2. BACKGROUND

As briefly described, ION is designed to achieve two main goals. First, it aims to support hardware devices with diverse memory requirements. Prior to ION, different SoC vendors achieve this through proprietary and mutually incompatible interfaces such as PMEM for Qualcomm, NVMAP for Nvidia, and CMEM for TI [8]. System and application developers have to customize their code heavily for all such interfaces to ensure that the code can work across all different platforms. This problem is greatly alleviated since the introduction of ION that defines a common interface irrespective of SoC manufacturers. The underlying implementation in the form of a driver can be customized by SoC and smartphone vendors to guarantee that they return the correct type of memory asked by the user space.

As with most interfaces exposed to user space, the unified ION interface is exposed through the */dev/ion* file, which can be manipulated through `open()` and `ioctl()` system calls. The specific set of supported operations include "alloc" and "free". The user space code needs to specify a heap id from which the memory should be allocated. As shown in Figure 1, each ION heap has an assigned name, id, and more importantly, an associated heap type that is pre-defined for a particular Android device. Table 1 illustrates the set of AOSP-defined heap types, along with a selected subset of customized heap types we encounter in the studied Android devices. Even though incomplete, it illustrates the complexity of ION with heaps of different types and properties. Some

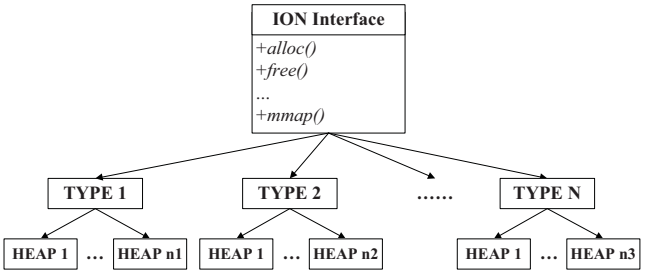


Figure 1: ION Architecture

heap types may appear to have similar properties: CMA and SECURE_CMA. However, they actually serve different purposes. CMA is accessible by third-party apps and system services. However, SECURE_CMA is usually intended to be used by trusted world (See TrustZone [4]), thus inaccessible from user space. Also, we omit another dimension, cache coherency, which is not the focus of this study.

Generally the heaps fall into two categories: 1) Unreserved. The most representative one is the SYSTEM heap, whose memory provider is the low-level buddy allocator according to our analysis , the same as memory allocated through `malloc()`. 2) Reserved. This includes CARVEOUT and CMA heaps that involve memory set aside at boot time so as to combat memory fragmentation at runtime [23].

The second goal of ION is to allow efficient sharing of memory between user space, kernel space, and the hardware devices. This is achieved by sharing memory pages directly to avoid copying. Specifically, following the ION interface explained earlier, once memory is allocated successfully from a heap, a file descriptor is returned to user space which can be subsequently used to invoke `mmap()` to map the allocated pages into user space. This feature can be handy in many scenarios. For instance, in the case where both software and hardware rendering are needed for graphics processing, libraries such as OpenGL can manipulate the memory in user space easily and a GPU can also populate the same piece of physical memory with zero copying.

## 3. VULNERABILITY ANALYSIS

So far, we have explained the design philosophy of ION including 1) unified memory management interface for ease

of use and 2) memory sharing support between user and kernel space. Interestingly, each one introduces a new class of vulnerabilities. In this section, we will unveil the root causes of the security flaws.

## 3.1 Problems Introduced by Unified Interface

As mentioned in §2, ION uses a unified interface */dev/ion* for all types and instances of memory heaps it manages. Unlike the general memory allocated through `malloc()` in user space, ION heaps come with different sizes and purposes, which require a different security design than the one for general memory. In Android, an application can allocate "unlimited" amount of general memory through `malloc()`. Because it is general memory, applications may have legitimate reasons to allocate and use a large amount of memory (*e.g.,* 3D gaming apps). The only time when memory allocation fails is when the system is running out of memory.

Unfortunately, ION inherited the security design for general memory. There does not exist any limit on how much memory one can consume in ION heaps, causing potential DoS attacks. Even worse, due to the fact that third-party apps have legitimate reasons to allocate memory from at least one heap type (for graphic buffers), the unified */dev/ion* interface needs to have a relaxed permission that allows anyone in user space to have access to potentially *all* ION heaps. Indeed, on all 17 phones we studied, the file permission of */dev/ion* is always world-readable [1]. There exists no other security mechanisms (*e.g.,* access control) beyond the file permission and therefore any app can allocate any amount of memory from any ION heaps (but no more than the max available of a certain heap). Due to characteristics of different heap types, such a capability can lead to two different DoS attacks:

**For fixed size heaps.** Certain ION heap types such as CARVEOUT and CMA have a pre-determined size and region from which users can allocate memory from. These heaps are typically used for various system functionalities, *e.g.,* "audio" heap is used by "mediaserver" on Nexus 6P to perform audio playback. As mentioned in §1, during the Android device customization process, the available heap types and instances are tailored to satisfy the need of hardware devices. In the above example, the "audio" heap is designed to work with a specific audio chip. As long as a user exhausts all free space of a certain heap, related system functionalities will stop working due to the failure to get required memory from the specific heap, *e.g.,* sound/music playback on Nexus 6P will be disabled if "audio" heap is occupied by a malicious app. In some cases, critical service failure can even cause the whole system to crash as shown in §6.

**For unlimited size heaps.** Some heap types, such as SYSTEM, have no pre-reserved memory regions. According to our analysis, memory allocated from the SYSTEM heap is not correctly accounted for as part of the memory usage of the calling process. Thus, from SYSTEM heap a process can request as much memory as the current system can supply. When a user process drains too much memory from such heaps, the performance of the whole system will be affected. Besides, due to the existence of Android low memory killer [2], other innocent processes may get killed to release more memory in such a situation. More detailed analysis is given in §6.

[1]Readable on */dev/ion* in fact allows both memory allocation and memory mapping to user space [8].

One may argue that the above problems can be solved by integrating a general access control or quota limitation mechanism into current ION interface, however, this may not be an easy solution as will be discussed in §7.

## 3.2 Problems Introduced by Buffer Sharing

As mentioned earlier, the zero-copy buffer sharing among user space, kernel space, and hardware devices is one of the main goals of ION. While some specific heap types such as SECURE_CMA may deny any access from user space, there is no general restriction in the ION framework; therefore users can allocate memory from most heaps and map them into user space for read and write operations. This can lead to two different security problems:

**System crash due to hardware protection.** Some heaps' memory regions can be protected by hardware security mechanisms like TrustZone [4] so that any access from untrusted world will cause a protection exception which usually leads to a system crash or reboot. Unfortunately ION buffer sharing unexpectedly makes such protected heaps accessible to untrusted world apps, as will be shown in §6.

**Sensitive information leakage.** ION memory are drawn from various memory heaps, whose allocation functions correspond to low-level kernel functions such as `kmalloc()` and `dma_alloc_*()`. By default, many of them do not zero the newly allocated pages for performance reasons. The original assumption was that such pages will never get mapped to user space directly and hence safe to use in kernel. Unfortunately it is no longer correct with the introduction of ION that exports such heaps to user space for buffer sharing. We confirm that it is far from rare cases that buffers allocated from ION heaps contain dirty pages. A large number of Android devices, including the newest models like Nexus 6P, have one or more ION heaps failing to clear allocated buffers before handing them to user space, which eventually enables any third-party app to access sensitive information leaked from kernel, system services, and user applications.

### 3.2.1 Root Cause Analysis of Unzeroed Pages

After careful investigation, we summarize two main reasons for unzeroed pages:

**Customization.** ION by design has good extendability and supports customization, as described in §1. Different vendors can have their own choices about which heap types to use and how they will be implemented. They can also implement new heap types by themselves as shown in Table 1. Thus, even though all the default heap types have zeroed their allocated buffers in AOSP common branch kernels since 3.10, in practice, almost all Android devices are shipped with customized ION implementation which do have the dirty page problem.

**Kernel memory allocation functions with complicated behaviours regarding buffer zeroing.** Unlike the relatively simple and limited interfaces for user-space memory allocation like `malloc()`, there exist many different memory allocation interfaces in kernel space, which will be directly used by various ION heap types. `malloc()` typically involves the system call `brk()` and `mmap()` where virtual pages are returned first. Upon accessing such pages by the program, a page fault occurs which triggers the OS to locate a physical page and map the accessed virtual page to it. Due to the obvious security risks, the OS always zeroes

the physical page before mapping it to user space (unless it is a page recycled from the same process).

In contrast, other kernel memory allocation functions are diverse, largely undocumented, and not well understood. They generally fall into three categories:

1) *Guaranteed zeroing.* Interfaces like `kzalloc()` are guaranteed to zero the allocated memory, which do not pose any threats even when exposed to user space through ION.

2) *Expected to zero but actually may not.* Some functions will decide whether to zero the memory based on some function parameters like *GFP_ZERO* flag used in `alloc_pages()` of buddy allocator. However, there is no guarantee that the zero operation will be performed. In our analysis, we found that some functions, like `arm64_swiotlb_alloc_coherent()`, do accept a parameter deciding whether to zero the allocated buffer, but the function implementation simply does not honor such a parameter at all. A similar issue was reported previously in [9].

3) *Undecidable and undocumented zeroing behaviour.* There also exist other functions where it is not obvious whether the returned pages will be zeroed. `gen_pool_alloc_aligned()` is one such function that is usually used by CARVEOUT heaps.

The confusing behaviors of various kernel memory allocation functions makes it difficult for developers to decide whether they should zero the buffer after invoking any kernel memory allocation function. On one hand, failure to zero the buffer may cause information leakage, while on the other hand, repeated zeroing operations may affect the overall performance, especially on embedded platforms. As an example, some buffer zeroing logic is surprisingly commented out intentionally for several ION heap types according to the kernel source code for Huawei Mate 8, a popular device running Android 6.0. We suspect that developers are trying to avoid the extra performance penalty, but the end result is a severe security flaw as will be demonstrated in §6.

# 4. METHODOLOGY

In this section, we will present our methodology to systematically test and discover the vulnerabilities uncovered in §3.

## 4.1 Problems Introduced by Unified Interface

The goal is to test whether a third-party app can indeed occupy memory from different heaps entirely to cause DoS attacks. Further, we want to understand what specific system functionalities can be targeted using which heaps. To this end, we design a simple runtime testing procedure as follows: given an Android device, we first enumerate all available ION heaps (declared through the Device Tree file [5]), identify their type and size information; we then try to allocate buffers from them with an ION memory probing app we develop. Once we find a heap able to provide memory to our app, we will further try to exhaust all available memory resources remained in the heap. This can be automatically done by our probe program, which will try to allocate a buffer as large as possible in each iteration of a loop and terminate the loop if no more memory can be allocated. The largest available buffer size in each iteration is decided by an efficient binary-search style probing. For unlimited sized SYSTEM heaps, we will also try to allocate as much memory as we can, until exceptions occur, such as our process getting killed by low memory killer.

As soon as an ION heap is exhausted, we will monitor system behaviors to see whether there will be any anomalies. Usually the heap name will give a good indication about what system behaviors to watch, for example, the name "audio" suggests that the heap should be used for audio data processing, then we will focus on the issues such as whether the system can still play sound normally. For SYSTEM heaps, we mainly focus on questions like whether the system performance will be affected or whether there are other processes get killed by low memory killer. If kernel and platform source code for the target device is available, we will also try to take a reference of it to figure out how the dedicated ION heaps will be used, which can help us find the potential DoS problems more efficiently and precisely.

## 4.2 Problems Introduced by Buffer Sharing

If a heap not only allows our app to allocate memory from it but also enables it to access the allocated buffers, then we will first attempt to access the buffers. In some cases, a simple memory read operation can already cause a system crash as described in §3. If the buffer can be accessed without causing exceptions, we will then determine whether the buffers from the current heap may contain dirty pages. This can be done in two ways:

**Blackbox testing.** We can simply exhaust the free space of a certain heap and read the allocated buffers to see whether they contain any non-zero bytes. To avoid the cases where the heap may not be populated by other services yet, *e.g.,* camera has not been used yet and therefore no data has been stored in the heap, we could write to the heap first and later on read from it again from another app to see if the data remain. The challenge with such a blackbox testing approach is that the behaviors can be dependent on the system state and the parameters we pass through the ION interface, which may not be easy to determine; this can lead to inaccurate assessments. In addition, blackbox testing also requires access to actual devices.

**Program analysis.** Alternatively, if the kernel source code for an Android device is available, which usually is the case due to open source licensing requirements, we can in fact accurately determine this via static analysis on the source code. As discussed before in §3.2, since the behaviors of kernel memory allocation functions are complex and in many cases not well documented, program analysis can automate the process and greatly reduce the manual effort.

### 4.2.1 Static Taint Analysis on Buffer Zeroing

To fulfill this task, we design and implement a novel *static taint analysis* tool to analyze the zeroing behaviors of memory allocation functions. Our design is based on three key observations:

(1) Most, if not all, memory allocation functions will take a parameter indicating the size of the requested memory. We consider such "size" parameters as *taint source*.

(2) Usually the zeroing operations will be performed through some common utility functions such as `memset()`, which will be considered the *taint sink*.

(3) When buffer zeroing operations occur in memory allocation functions, the amount of memory zeroed should be dependent on the "size" taint source. In other words, the taint source should propagate to the sink to indicate a true buffer zero operation.

The lack of an information flow from the source ("size") to

```
1 allocate_1(...,size,...){
2   order=log_2(size);
3   some allocation operations;
4   memset(addr,0,2^order);
5 }
```

Figure 2: Data dependency

```
1 allocate_2(...,size,...){
2   some allocation operations;
3   for(i=0;i<size;i+=4096)
4       memset(addr+i,0,4096);
5 }
```

Figure 3: Control dependency

sink (functions such as `memset()`) in an ION heap allocation function indicates that the function does not zero the buffer before returning it to user space. It is worth noting that the analyzed memory allocation functions may include zeroing operations for some internal or temporary data structures other than the allocated buffer, in which case may cause confusion. Our intuition is that such data structures will not be dependent on the "size" taint source and therefore can be eliminated automatically.

**Design Considerations.** Even though the formulated problem is clearly defined, there are still several complications that need to be carefully considered. First, taint propagation typically has two forms: data dependency (explicit flow) and control dependency (implicit flow). We need to decide whether to track data dependency alone or both. Most static taint analysis tools focus on only data dependency [22, 12, 17, 26]. However, in the case of memory zeroing operations, the decision may not be so straightforward. We illustrate two real world examples (simplified) we encounter in Figure 2 and Figure 3. `allocate_1()` round up the requested size to the nearest power of 2 before allocating a buffer and `memset()`ing it. The "order" variable is data dependent on the "size" taint source; therefore, it is sufficient to consider data dependency only in this case. In contrast, `allocate_2()` decides to invoke `memset()` to zero the allocated buffer page by page. No data dependency exists from "size" to the parameter of `memset()`. Instead, a tainted control dependency exists from "size" to `memset()` as the loop condition is dependent on "size". In this case, we will need to follow all function calls after meeting a tainted control dependency so as to not miss any sink functions (*e.g.,* `memset()`). However, such strategies can incur false positives, as we will show in §5.3. We acknowledge that it is an inherently difficult problem to propagate taint through control dependencies, as is recognized in prior work [13]. As an alternative solution, manual intervention can be used to determine the propagation rules upon each tainted control dependency. We give a complete walkthrough of the methodology in §5.3.

Second, it is possible that the ION memory allocation function may internally invoke different low-level kernel memory allocation functions (*e.g.,* fall back to a different function if a previous one fails). Therefore, even if there exists an information flow from source to sink (for certain program paths), it does not rule out the possibility that another program path does not zero the buffer. To address this issue, our tool will output the unique call chains associated with the taint paths and guide the developer to look for other low-level memory allocation functions; our assumption here is that there must exist a different memory allocation function for each type of low-level memory allocator. Depending on the system state, or the result of an earlier memory allocator, ION may choose to invoke a different memory allocator (again, in the form of a separate function). With this assumption, the tool can output the

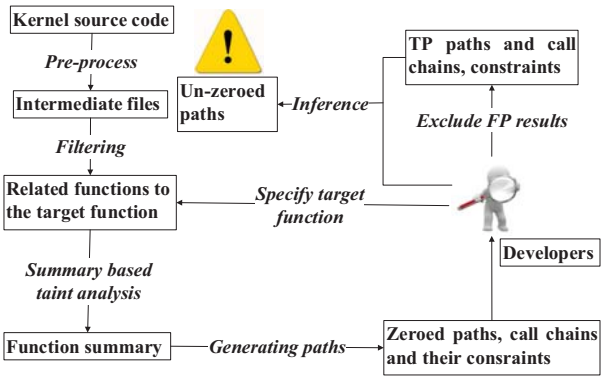


Figure 4: Static taint analysis tool workflow

callees for each memory allocation function in the tainted call chain. If developers recognize any callee that also appear to be a memory allocation function (and takes in a tainted argument), they can query the taint analysis result to see if the callee has encountered any zero operation down the line. If so, the tool simply repeats the same procedure to look for additional candidate callees. Otherwise, we conclude that there does exist a program path that performs memory allocation without buffer zeroing. In §5.3, we use the SYSTEM heap on Nexus 6P as a case study to explain the methodology in detail.

Third, theoretically buffer zeroing can also occur during memory release functions, *e.g.,* `free()`. In practice, we find that ION heaps always have relatively simple logic in memory release functions and they almost never zero buffers in them. This could be due to the fact that memory allocation functions can opportunistically skip zeroing operations if the pages are from the same process, *e.g.,* memory allocated through `malloc()`. Also, if the released memory is not be reused afterwards, the zeroing is simply wasted. Therefore, in our analysis, we focus on analyzing memory allocation functions, which are orders of magnitude more complex and may or may not contain zeroing operations.

**Implementation.** We implement the static taint analysis based on STAC [7], an open source static taint analysis tool. The workflow is described in Figure 4. Given the kernel source code of a specific Android device, we first perform pre-processing using GCC to produce .i files with expanded macros and include files. Then we perform filtering to exclude the functions that are never reachable from the ION functions. Finally, we implement a flow-sensitive taint analysis engine that takes in the entry function, *i.e.,* ION memory allocation functions, as well as the source and sink. When the taint engine finishes the computation, we output the taint paths where the source "size" can successfully propagate to sinks (*e.g.,* `memset()`). Finally, we group the taint paths into call chains for developers to inspect and confirm. Based on the true positives, developers can then follow the procedure described above to locate unzeroed paths.

# 5. EVALUATION

In this section, we will first give the experiment setting and a summary of discovered vulnerability instances. Then we will evaluate the effectiveness of the methodology to discover two classes of vulnerabilities, including how successful

| Devices | Platform | Kernel | A1 | | A2 | | A3 | | A4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | effect | heap | effect | heap | effect | heap | effect | heap |
| Galaxy S7* | exynos5 | 3.18.14 | D1 | T3:N7,N16 T4:N15,N11 T5:N17,N14, N12,N13 | yes | T1:N18 | no | no | L1 | T3:N7,N16 |
| Galaxy S6* | exynos5 | 3.10.61 | D1,D4,D3 | T3:N5,N3 | yes | T1:N18 | no | no | L1 | T3:N3,N5 |
| Meizu Pro 5 | exynos5 | 3.10.61 | D1,D4 | T3:N5,N3 | yes | T1:N18 | no | no | L1 | T3:N3,N5 |
| Nexus 6P* | msm8994 | 3.10.73 | D1,D2 | T4:N2,N4,N1 | yes | T1:N19 T2:N20 | yes | T4:N6 | L2,L4 | T4:N1,N4,N2 |
| LG V10 | msm8992 | 3.10.49 | D1,D2 | T4:N2,N4,N1 | yes | T1:N19 T2:N20 | no | no | L2,L4 | T4:N1,N4,N2 |
| HTC A9 | msm8952 | 3.10.73 | D1,D2 | T4:N2,N4,N1 | yes | T1:N19 T2:N20 | yes | T4:N6 | L2,L4 | T4:N1,N4,N2 |
| Oppo R7sm* | msm8916 | 3.10.49 | D2 | T4:N2,N4,N1 | yes | T1:N19 T2:N20 | yes | T4:N6 | L2,L4 | T4:N1,N4,N2 |
| Nexus 5X* | msm8992 | 3.10.73 | D1,D2 | T4:N2,N4,N1 | yes | T1:N19 T2:N20 | no | no | L2,L4 | T4:N1,N4,N2 |
| Xiaomi 4C* | msm8992 | 3.10.49 | D2 | T4:N2,N4,N1 | yes | T1:N19 T2:N20 | no | no | L2,L4 | T4:N1,N4,N2 |
| vivo Y927 | msm8916 | 3.10.28 | D2 | T4:N2,N4,N1 | yes | T1:N19 T2:N20 | no | no | L2,L4 | T4:N1,N4,N2 |
| Nexus 5* | msm8974 | 3.4.0 | D2 | T4:N2,N4 T3:N1,N6 | yes | T1:N21 T2:N20 | no | no | L2 | T3:N1 |
| LG D950* | msm8974 | 3.4.0 | D2 | T4:N2,N4 T3:N1,N6 | yes | T1:N19 T2:N20 | no | no | L2 | T3:N1 |
| HTC D816 | msm8226 | 3.4.0 | D2 | T4:N2,N4 T3:N1,N6 | yes | T1:N19 T2:N20 | no | no | L2 | T3:N1 |
| Oneplus One | msm8974 | 3.4.0 | D2 | T4:N2,N4 T3:N1,N6 | yes | T1:N19 T2:N20 | no | no | L2 | T3:N1 |
| Galaxy note 3 | msm8974 | 3.4.0 | D2,D4 | T4:N2,N4 T3:N1,N6,N7 | yes | T1:N19 T2:N20 | no | no | L1,L2 | T3:N1,N7 |
| Huawei P9 | hi3650 | 3.4.90 | D5 | T3:N8 | yes | T1:N22 T2:N10 | no | no | L4 | T4:N9 T2:N10 |
| Huawei Mate8* | hi3650 | 3.4.86 | D5 | T3:N8 | yes | T1:N22 T2:N10 | no | no | L4 | T4:N9 T2:N10 |

* Devices with detailed experimentation and constructed exploits
- "effect" column shows only attack effects known to us (which can be incomplete)
[**A**] **Vulnerabilities Classification:** **A1**:DoS for limited size heaps **A2**:DoS for unlimited size heaps **A3**:System crash due to protection exception **A4**:Information leakage
[**D**] **DoS Attack:** **D1**:fingerprint **D2**:audio **D3**:video **D4**:camera **D5**:system crash
[**L**] **Information Leakage:** **L1**:camera **L2**:audio **L3**:video **L4**:general user apps
[**T**] **Heap Type:** **T1**:SYSTEM **T2**:SYSTEM_CONTIG **T3**:CARVEOUT **T4**:CMA **T5**:SECURE_CMA
[**N**] **Heap Name:** **N1**:audio **N2**:adsp **N3**:video **N4**:qsecom **N5**:secdma **N6**:pil **N7**:camera **N8**:carveout-heap **N9**:ion-dma-heap **N10**:system-contig-heap **N11**:video_nfw **N12**:video_fw **N13**:video_scaler **N14**:video_frame **N15**:crypto **N16**:gpu_crc **N17**:gpu_buffer **N18**:ion_noncontig_heap **N19**:system **N20**:kmalloc **N21**:vmalloc **N22**:system_heap

Table 2: Vulnerability summary

is the static taint analysis tool in practice. Finally, we will use case studies to highlights important findings.

**Scope.** We have analyzed 17 Android devices in total, which are listed in Table 2. They cover a wide range of devices such as Nexus, Samsung, to HTC. All of them are verified through runtime testing and source code analysis. Out of 17 devices, we have experimented in detail using 8 devices, for which we have constructed exploits to confirm the existence of vulnerabilities.

## 5.1 Summary of ION related Vulnerabilities

After applying the methodology described in §4, we report our findings in Table 2. Note that we group the tested devices based on their hardware platforms (SoC vendor and model) and kernel versions. This is because hardware devices are the most common reason for customization of ION. Generally speaking, devices sharing the same hardware platform will have similar configurations regarding ION heap types and instances. In addition, different phone vendors and kernel versions may also have an impact.

In our study, we focus on three main general hardware platforms: MSM (Qualcomm), Exynos (Samsung), and Hisi (Huawei). Each platform can also include different models, *e.g.,* Snapdragon 810 and 820 correspond to two different Qualcomm SoCs, along with numerous Android and kernel versions. As we can see in the Table, there exist a variety of vulnerable ION heaps (up to 22 instances across all devices).

In the Table, we breakdown the vulnerabilities into 4 categories, along with their corresponding attack effects and vulnerable heaps (types and instances). For instance, regarding A1: DoS against heaps of limited size, all experimented devices are vulnerable in one form or another. On Huawei devices specifically, A1 attack can even cause the whole system to crash directly. Regarding A4: information leakage, all studied Android devices have unzeroed memory that can be leaked from different heap types. The most surprising result is that 9 out of 17 devices have the information leakage vulnerabilities that allow a malicious app to obtain dirty pages used by other apps, which can contain sensitive

| Heap Type | TP* | FP* | FN* | Un-zeroed paths? | Analysis/Actual result | Involved allocation function |
|---|---|---|---|---|---|---|
| SYSTEM | 1158(8) | 5 | 0 | n | zeroed/zeroed | alloc_pages()** |
| SYSTEM_CONTIG | 288(6) | 0 | 0 | n | zeroed/zeroed | alloc_pages()** |
| CMA | 4(4) | 2 | 0 | y | un-zeroed/un-zeroed | dma_alloc_attrs() |
| CARVEOUT | 0(0) | 0 | 0 | y | un-zeroed/un-zeroed | gen_pool_alloc_aligned() |

* TP, FP, and FN refer to the amount of call chains. The number in brackets indicate the number of paths.
** The parameters passed to alloc_pages() are different for the two heap types.

Table 3: Static taint analysis result on Nexus 6P

information such as passwords, credit card numbers, or even secret keys.

Although the number of Android devices we analyze is limited, they do cover most representative manufactures, hardware platforms and software versions, thus we can infer that most Android devices to date are affected by ION-related vulnerabilities. Specifically, Nexus 6P, Samsung Galaxy S7, and Huawei Mate 8 represent the latest devices from each manufacturer, all of which have both DoS and information leakage vulnerabilities.

## 5.2 Runtime Testing for DoS Vulnerabilities

The runtime testing procedure described in §4.1 is overall effective for most devices; however, when applying this methodology, we did encounter some special cases in which the normal routine fails to give useful results even though we can successfully allocate arbitrary memory buffers from a certain heap. We describe them below.

**Failure to identify any DoS vulnerabilities.** For certain heaps with limited size on some devices, even after we occupy all of its free space, no issues can be observed. After looking into these cases, we conclude two main reasons for this: 1) Some heaps will be rarely, if not never, used by their host devices. For instance, we cannot observe any utilization of "kmalloc" heap, whose type is SYSTEM_CONTIG, on Nexus 6P in our experiments. Besides, there exists other heaps that may be used in only the early stage of system booting, such as "pil" heaps on some devices that are used to load certain firmware images during the boot process.

**Vulnerabilities depending on proper timing.** In some cases, we can successfully perform DoS attacks against certain system functionalities by exhausting specific heaps, but not at arbitrary points in time. In the case of Samsung S6's fingerprint authentication service, a CARVEOUT heap named "secdma" is used to fulfill its task. If a malicious app occupies the entire heap ahead of time, then fingerprint service will stop functioning. The challenge is that the service itself typically occupies the heap when the screen is locked and releases it only when the screen is unlocked. Generally speaking, our testing methodology may not always be able to catch the correct timing; however, such vulnerabilities do exist and are simply harder to trigger. Manual investigations are performed to catch these cases as reported later in §4.1.

## 5.3 Static Taint Analysis for Dirty Pages

Next, we evaluate the effectiveness of the static taint analysis described in §4.2.1, using Nexus 6P as a case study. The source code we analyze is from kernel version 3.10.73. In total, it takes about 9.5 hours for the tool to analyze the memory allocation functions for 4 ION heap types — SYSTEM, SYSTEM_CONTIG, CMA, and CARVEOUT — on a server with Intel Xeon E5-2640 V2 CPU and 64GB physical memory. The analyzed LOC is over 10,000. We omit the remaining heap types that fall in two categories: 1) the ones that never have any instances, *e.g.,* REMOVED type; 2) the ones that deny any `mmap()` requests from user space, *e.g.,* SECURE_CMA, indicating that it is impossible to access the memory although they may contain dirty pages.

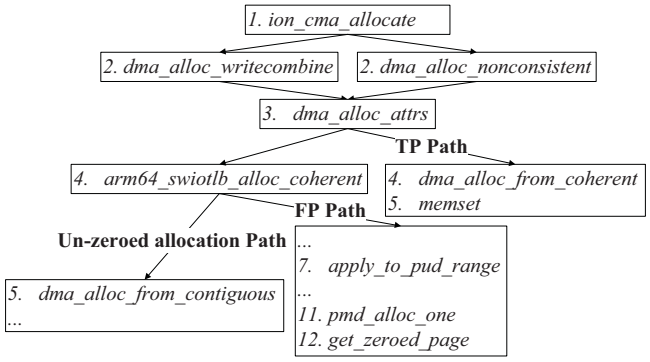


Figure 5: Call chains of interest for CMA heap type

We summarize the result of the tool on Nexus 6P's kernel source code in Table 3. We output the number of taint paths as well as the corresponding unique call chains (the number in brackets indicates the number of unique call chains). We confirm the true positives, false positives, and false negatives by manually analyzing the source code. The false positives are relatively easy to deal with, as a developer or researcher can quickly inspect the output path (or even the function call chain) to confirm them (we will explain the FP cases of CMA heap later). For false negatives, it is more problematic as we may not even realize this and incorrectly report that there are no zeroing operations while in fact there are. One potential source of false negatives is the incomplete set of sink functions considered, which include the common functions such as `memset()` and `bzero()` so far. However in our evaluation, we found that `memset()` and its wrappers are the only used sinks.

**CMA heap type.** We first discuss the false positive taint paths in CMA heap. We group such paths into call chains as shown in Figure 5 (labelled "FP path"). When we look at the results, the taint analysis in fact correctly outputs the taint result according to the source and sink definition. Unfortunately, in some cases, the allocation of auxiliary data structures, such as page table entries, is also dependent on the size of requested buffer. Specifically, the auxiliary data here is the page table entries that are created and subsequently zeroed. In other words, the source ("size" parameter) indeed propagated to the sink ("zeroing"

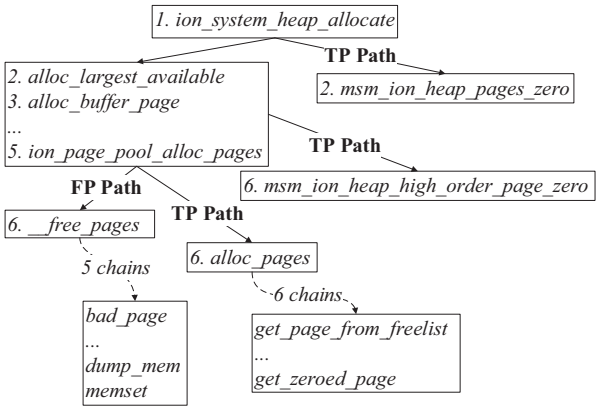


Figure 6: Call chains of interest for SYSTEM heap type

operation); the zeroing operation is simply not applied to the returned buffer. Fortunately, by simply looking at the function names involved in the FP call chain in Figure 5, it is easy to conclude that this call chain is for allocating pages to hold page table entries given the term "pmd" (page middle directory).

According to our evaluation, there do exist true positive paths that result in zeroed buffers. The cases are also shown in Figure 5. Now the question is whether there still exist unzeroed paths other than the zeroed path. As stated previously in §4.2.1, we walk backwards along the true positive call chains and look for branches that may invoke other memory allocation functions that do not belong to any taint call chain. As shown in Figure 5, starting from `dma_alloc_from_coherent()`, we walk backwards to its caller `dma_alloc_attrs()` and enumerate all of its callees. By looking at the names of the callees, we locate `arm64_swiotlb_alloc_coherent()`, which also takes in a tainted parameter "size", and appears to be a memory allocation function. We then cross reference the function name with our taint analysis results. In this case, we found no taint call chains that involve `arm64_swiotlb_alloc_coherent()`, which indicates the possibility of a path where the allocated memory is unzeroed. Upon reading the code, we realize that the code tries to allocate memory through `dma_alloc_from_coherent()` first, and fall back to `arm64_swiotlb_alloc_coherent()` only when an error is returned earlier. During our runtime testing on Nexus 6P, `dma_alloc_from_coherent()` appears to be failing all the time and therefore we are able to successfully obtain dirty pages in CMA heap.

**SYSTEM heap type.** SYSTEM heap is complex and involves many paths leading to zero operation, a simplified call graph is shown in Figure 6. All 5 FP call chains are introduced by the function `__free_pages()`, from which an error branch is eventually triggered which invokes `memset()`. When we look at the issue closely, it all started from the function `alloc_largest_available()`, whose key logic is depicted in Figure 7. Upon close inspection, there exists a control dependency (implicit flow) that caused the problem. The function essentially runs in a loop to identify the closest round-up of the request memory size in the power of 2, and use the round-up value to allocate memory. Here the constant array $order[i]$ simply pre-defines all possible round-up values. Note that the $size$ parameter (taint source) is compared against the round-up value $orders[i]$, which results in a control dependency. The variable $order[i]$ itself is not tainted as it is a read-only constant. When it is passed over to `alloc_buffer_page()`, we lose track of the taint. In reality, $order[i]$ is semantically derived from the taint source $size$ (round-up of size), however, traditional taint propagation rules are unable to catch this case. Therefore, we have to record all function invocations after the control dependency and report whenever a sink (*e.g.,* `memset()`) is encountered, regardless of whether its parameters are tainted (we had to implicitly assume that all parameters of the sink is tainted).

```
1  struct page_info *alloc_largest_available(...,size,...){
2      for (i = 0; i < num_orders; i++) {
3          if (size < order_to_size(orders[i]))
4              continue;
5          page = alloc_buffer_page(...,orders[i],...);
6      }
7  }
```

Figure 7: Key logic for alloc_largest_available()

Of course, in practice, such a coarse-grained control taint propagation rule is likely going to introduce false positives. It is inherently a challenge to deal with control taint propagation, as widely acknowledged in previous studies [13]. As an alternative solution, a developer can resolve the control taint manually. In this case, since we know $orders[i]$ is essentially derived from the taint source $size$, one can simply taint $orders[i]$ directly to avoid the false positives.

Aside from the false positives, we wish to point out an interesting observation. Figure 6 appears to suggest that there are multiple paths to zero the allocated buffer and there exists only one path that actually allocates memory. Specifically, both `msm_ion_heap_pages_zero()` and `msm_ion_heap_high_order_page_zero()` are simply zeroing a buffer without allocating any memory. Only `alloc_pages()` is allocating memory (as well as zeroing the buffer afterwards). Therefore, one may think whether the buffer is zeroed for more than one time. However, upon a closer look, we realize that two of the sinks are not really performing the zeroing operations. Both `msm_ion_heap_high_order_page_zero()` and `alloc_pages()` depend on a flag "GFP_ZERO". Only if it is set will they zero the buffer. In this particular case, the flag is not set for either function to avoid repeated zeroing that can waste CPU cycles. It is interesting to see how complex the memory allocation can be and how hard the developers need to try to ensure security as well as performance. This once again shows the benefit of a program analysis tool to help developers make correct implementation decisions.

# 6. CASE STUDY

In this section, we will demonstrate our exploitation of ION related vulnerabilities on a few latest and representative Android devices including flagship models from mainstream manufactures with newest Android system and kernel. It is worth noting that although the vulnerabilities usually manifest themselves differently on various devices due to customization, the underlying cause stems from the same design and implementation of ION as we outlined earlier.

## 6.1 DoS against Heaps of Fixed Size

**Disable fingerprint authentication service on (multiple devices).** On Nexus 6P, if one occupies enough free

space of "qsecom" heap, which is of CMA type, the device's fingerprint authentication functionality will be effectively disabled. When a user tries to unlock the device with his/her finger, the system will show an error message "fingerprint hardware is unavailable". Similar attacks can also be performed on all Android devices using MSM platform, including Samsung Galaxy S7 and S6.

**Disable audio service (multiple devices).** On many MSM platform Android devices there exists an "audio" heap, with either CARVEOUT type or CMA type. If we exhaust this heap's free memory, the system will be unable to produce any sound, including ringtones. Affected devices, such as Nexus 6P and OPPO R7s, will be unable to notify users or play any music under this attack. The sound playback is the responsibility of a system service named "mediaserver", which heavily depends on the "audio" heap as its buffer provider on MSM-based Android devices.

**System crash on Huawei Mate 8.** Huawei Mate 8 is shipped with a CARVEOUT heap named "carveout-heap". We can request memory buffers from it and when we claim and keep a big enough buffer, the device will crash directly. This CARVEOUT heap has only a fixed size, and is used by the critical system framebuffer service that is responsible for the screen display and refresh. When we occupy too much resource in the heap, the critical system service will fail to work and eventually cause the whole system to crash.

## 6.2 DoS against Heaps of Unlimited Size

**System level DoS (multiple devices).** On virtually all devices, there is a SYSTEM heap usually named "system". If we allocate a large enough memory buffer (usually around or more than 1GB) from there, the system will freeze and many running background services will be killed at the same time, including music playback service and push notification service.

As mentioned previously in §3.1, SYSTEM heaps will request new pages from the basic buddy allocator when needed, so the available memory for them is equal to that of the whole system. Surprisingly, there is one important difference between allocating memory from the "system" heap and simply using functions like `malloc()`: the memory from the latter will be considered "owned" by the calling process, whereas the buffers from the former is actually allocated and "owned" by the ION driver; what the requesting process gets is only a handle to the buffer. Since Android is equipped with a low memory killer [2] which is responsible for releasing memory by killing processes when it detects that the system is currently low on memory, when a process allocates too much memory via interfaces like malloc(), it will gain a high priority in the "killing list" since the killer thinks that it owns too much, the result is that it will be killed soon and the system will thus recover quickly. However, when allocating a large amount of memory from the "system" heap, the killer will not consider our process as a main memory holder; instead, it will try to kill other innocent processes such as the push notification service.

## 6.3 DoS via Protected Memory Access

**System crash (multiple MSM platform devices).** There is a CMA heap named "pil" on Nexus 6P and many other Android devices that use MSM platforms, from which a program can request memory buffers. By trying to access the allocated buffer, the whole Android operating system will crash and the device will reboot immediately. The name "pil" is short for "peripheral loader"; it is used to load peripheral devices' firmware images when kernel boots. Since it is important to guarantee firmware images' integrity, the memory region of this heap will be protected by TrustZone, which is an SoC security extension providing Trusted Execution Environment (TEE) for sensitive operations with physically isolated memory and CPU mode [4]. Thus, any access to this protected memory region from normal world will cause the system to raise a protection exception, which usually leads to a system reboot.

The problem here is that a user can allocate buffers from "pil" heap even though it is never supposed to be exposed to user space. Unfortunately, as we highlight, the unified interface of ION grants a program access to virtually all heaps. In addition, the buffer sharing capability allows a program to further access the allocated buffer. Either reset the protection before the user can access the allocated buffers or simply deny users' memory requests for "pil" heap can solve the problem; however, neither is done on the devices.

## 6.4 Information Leakage

According to our analysis, there exist three different types of information leakage vulnerabilities, classified based on ION heap types. They are CARVEOUT, CMA and SYSTEM_CONTIG respectively, all of which do not zero the buffers before returning them to user space. This series of vulnerabilities can cause sensitive information leakage from both system and user applications, enabling an attacker to easily breach user privacy such as getting access to email content, bank accounts, and passwords.

**Camera data leakage on Samsung Galaxy S7.** There exists a CARVEOUT heap named "camera" in Samsung's newest flagship phone model S7. We can obtain image data captured by the phone's camera without any permission with the following steps:

(1) Open the system "camera" application, or third-party camera applications which also need to use the system camera hardware. Then capture some images with the application, note that it is unnecessary to actually take a picture — simply seeing the preview on the screen is enough.

(2) Close the camera application and then dump the whole "camera" heap by having any app allocating buffers from it and read their content, which contains the image data captured by the system camera, including the previews. We confirm this by byte-to-byte comparison between the picture (taken by the camera) and the memory content of the ION buffers.

As shown in Table 1, a CARVEOUT heap will manage a fixed physically contiguous memory region which is reserved by kernel at boot time for special purposes (*e.g.,* to satisfy the requirement of certain hardware devices). On S7, "camera" is such a heap that serves as a data buffer used by the system camera service. The camera device must have the requirement of physically continuous memory in order to perform DMA (and possibly other) operations. When users are running a camera app, the camera device should be populating the image data into the buffer allocated from the "camera" heap, which will then be released when users exit the app. Thus, an attacker can now re-allocate buffers from the "camera" heap to obtain the dirty buffer.

In our research, this kind of problems for CARVEOUT heaps widely exist on multiple Android devices, enabling

attackers to steal sensitive data from various system services. For example, the audio data can be leaked via "audio" heap on Nexus 6P and many other MSM-based Android devices in the same way.

**Live memory dump of running apps (multiple devices).** "Qsecom" heap on Nexus 6P is of CMA type, which is used mainly by TrustZone related services on MSM platforms. Different from CARVEOUT heaps that are used exclusively by certain system services, with the CMA heap we can obtain various kinds of sensitive information, including but not limited to Gmail contents, Chase bank transactions and wi-fi passwords, from "live memory" of running apps. This is achieved by the following three steps:

(1) Drain free system memory by allocating as much memory as we can from either normal user space interfaces like malloc() or ION SYSTEM heaps as shown in §1. Note that the buffer allocation should not cause observable slowdown to the system to avoid alerting the users.

(2) Run any victim app (*e.g.,* Gmail) normally which will naturally produce sensitive data in memory.

(3) Dump the content of "qsecom" heap, which will contain sensitive information as mentioned above.

The root cause is as follows: while both CARVEOUT heaps and CMA heaps manage some pre-reserved contiguous memory regions dedicated for certain system services, a major difference is that CARVEOUT memory regions are set aside at system boot time and invisible to kernel memory manager after the system boots; thus no other processes can reuse these regions — even when they are free — using normal memory allocation interfaces (*e.g.,* `malloc()`) other than ION. Effectively, the CARVEOUT memory is stolen from the system, which guarantees the availability of memory buffers to the corresponding system service and hardware device (*e.g.,* camera) In contrast, CMA heaps expose their reserved memory regions to kernel memory manager (*e.g.,* Linux buddy allocator) and thus allow other processes to utilize these regions through standard interfaces such as `malloc()`, when there is no sufficient memory that can be found elsewhere (attack step 1 ensures this condition). However, to allow system services to function, the memory allocated from CMA heap can be reclaimed on demand as soon as the system service asks for them (which is how attack step 3 can successfully dump live memory of other running apps). Prior to the memory being reclaimed, the data generated on the CMA heap will be copied elsewhere and page tables will be updated to reflect the change. As we can see, the design of CMA heap allows a better utilization of memory resources at the cost of a potentially longer latency when memory is allocated from CMA heap [23].

The CMA attack is considered extremely dangerous since it effectively allows a malicious app to dump the live memory of any other apps. Equipped with the capability of knowing which apps run in the foreground (through attacks such as [14]), it can effectively be tailored to reliably extract any app-specific information. One may even be able to reconstruct the app GUIs by taking multiple snapshots of the memory dumps; This very attack was recently achieved by forensically dumping the entire physical memory [24]. Finally, even though we have not attempted, there is no reason to believe that it is impossible to extract crypto keys used by apps and the system.

**System-wide information leakage on Mate 8** The "system-contig" heap on Huawei Mate 8 is assigned the type "SYSTEM_CONTIG". With this heap we can get various sensitive information similar to what we can get with CMA heap; they include Gmail content, Chrome browsing history and html data of previously loaded web pages. Besides data leaked from user applications, it is also possible to learn information from the kernel with such a heap type. The attack process is sketched as below:

(1) Open arbitrary apps and operate normally, then exit (which means that memory will be released by the apps).

(2) Allocate as much memory as possible from "system-contig" heap and record their content, which will include significant sensitive information.

Since SYSTEM_CONTIG heap draws pages directly from the basic system memory manager (*i.e.,* buddy allocator), whoever returns pages to the system without zeroing them can leak data to the attacker. This includes any application or kernel data such as passwords, credit cards, and secret keys. This vulnerability is very much similar to the one in CMA heap except that the opportunity arises only when the memory is freed by other applications.

# 7. DEFENSE DISCUSSION

The fundamental problems with ION stem from its two design goals, *unified interface* and *buffer sharing across layers*, as we highlight throughout the paper. Since they are not really simple implementation errors, they require a more systematic investigation. One may consider the vulnerability of unzeroed buffers simply a glaring error; we argue that it actually is a much more complex problem than what it appears. As shown in §3.2.1 and §5, the unzeroed buffers are introduced due to several complex reasons:

1) Kernel memory allocation functions have complex behaviors. Many of them never need to be exposed to user space prior to ION's cross-layer buffer sharing capability. It is hard to make the correct assumption on whether a particular function will zero the returned buffer or not.

2) Customization of ION can lead to a drastically different implementation from the common branch in AOSP. In our analysis of Nexus 6P, we find that its ION implementation follows the one customized by Qualcomm (due to the fact that Nexus 6P uses the Qualcomm SoC). Interestingly, we note that the Qualcomm kernel source was forked prior to AOSP common branch fixing the vulnerability of unzeroed buffers in CMA heap. Unfortunately, once the Qualcomm source tree is forked, it no longer merges the patches applied to AOSP common branch. This is demonstrated by the fact that even the Android 7.0 preview on Nexus 6P still has unzeroed buffer vulnerabilities.

Therefore, we believe the static taint analysis tool can be effective in assisting developers with the insight into the buffer zeroing behaviors and help them navigate the complex kernel functions.

However, even when the buffer zeroing problem is resolved. The DoS vulnerabilities introduced by ION's *unified interface* design still remain. Fundamentally, the unified interface hurts security as it supports only coarse-grained access control (through file permission of */dev/ion*) — a user has access to either all heaps or none. A fine-grained access control is necessary to solve the problem. For instance, a third-party app should not be able to access the heap used by fingerprint service; we have not seen a case where such heaps are accessed by any other process. Due to the fact that ION is a complex system and the entire software stack

is involved (from applications, system services, and kernel drivers), it is important that the changes minimize side effects such as backward compatibility and performance hit. In addition, we should avoid adding new security mechanisms to the kernel and instead try to piggyback on existing mechanisms offered by Android and Linux if possible.

One straightforward solution can be adding the fine-grained access control to the kernel to govern how much memory each user can allocate (based on uid/gid). This solution maintains backward-compatibility to existing applications and system services (and can be implemented efficiently). However, the downside is that the kernel needs to maintain an access control list for all (uid, ion heap) pairs that does not fit in any existing Android or Linux security mechanism. Even SeAndroid/SeLinux cannot express such a security policy, as it will need to be able to interpret the argument `struct ion_allocation_data *` of `ioctl()` to extract the heap information. Furthermore, the additional access control list needs to be changed every time when customization occurs as heap types and instances may change.

Alternatively, one can place the access control enforcement at the user space. This requires revoking direct access to */dev/ion* from third-party apps and allowing access to only system services. The idea is that apps will need to go through system services to allocate memory from ION heaps, in which case the system services can enforce the access control policy. For instance, if the policy says that a regular app can allocate memory from SYSTEM heap for graphics processing of up to 20MB, it will have to send the request to "mediaserver" (or whichever process that is responsible for managing graphics buffers). The "meidaserver" can then check if the requested memory is indeed in SYSTEM heap (not any other heap). Further, it will keep track of how much memory has been used by the user. This solution requires grouping all system service uids (*e.g.,* media) into an "ion" group so that the file permission of */dev/ion* can be changed to 660 (`rw-rw----`) where the user owner is system and group owner is ion. Now an app needs to go through system services for memory allocation in ION heaps. Since most apps only need to request memory for graphics buffers, the changes should involve only the app-side libraries that are responsible for allocating graphics buffers (that previously interacts with */dev/ion* directly) and a single system service (*e.g.,* "mediaserver"). Other heaps are automatically inaccessible to third-party apps. This does introduce the overhead of an additional IPC round trip for each ION memory allocation. Note that even though the memory allocation goes through system services, the returned memory pages still need to be zeroed. With this extra layer of indirection, the zero operation can in fact be performed by the system service itself before sending the file descriptor over to the app that requested the memory.

Besides the challenge in maintaining backward-compatibility, another downside of the alternative design is that once a process capable of accessing */dev/ion* is compromised, it can still launch the DoS attacks against other services. That is because of the access control not fine-grained enough to differentiate different system services. To truly achieve fine-grained access control, each system service needs to run with a different uid which the kernel can use to enforce the access control properly (as is done in the first solution).

In summary, we show two potential solutions that have different tradeoffs in the following aspects: backward-compatibility, performance, and avoid introducing new security mechanisms in the kernel.

# 8. RELATED WORK

**Android customization and related security issues.** Android customization is known to introduce new security vulnerabilities across layers. At the application layer, preloaded apps have been shown to require more permissions than needed [27]. At the framework layer, customized system services have been shown to have missing permission checks [25]. At the system layer, devices files are shown to have weak permissions that allow third-party apps to directly manipulate device drivers and perform privileged operations [29]. In addition, devices drivers also introduce vulnerabilities that can directly cause root exploits [28]. A recent study has shown that by analyzing configuration differences across customized Android ROMs, many security flaws can be reveled in all these layers [10]. Our study is a systematic analysis of an overlooked system component, Android ION, that is customizable by SoC and smartphone vendors.

**Android DoS vulnerabilities.** DoS attacks (*e.g.,* soft reboot) against the Android system services have been demonstrated using different techniques, *e.g.,* by issuing targeted and repeated requests to the system services [19], or forking an unlimited number of processes exploiting a weak local socket permission of the Zygote process [11]. In addition, a number of other vulnerabilities such as Null pointer and integer overflow have been reported recently [6, 3]. All of the DoS attacks can cause only the entire Android framework or system to reboot. Our DoS attacks exploit a new class of vulnerabilities that exist due to the lack of access control and memory usage limit in various ION heaps. Any services or apps that require memory from ION can be targeted. Due to the fact that some heaps are mostly used by one or two services, the DoS impact can be controlled to affect only those services; this has not been reported before.

**Unzeroed/Dirty memory.** Dirty memory can leak critical data to a malicious application, and there are different underlying causes. For instance, recently it has been shown that newly allocated GPU memory pages are not zeroed, and may contain data rendered by other applications [20]. In Linux, memory obtained by `malloc()` will automatically be zeroed by the underlying OS if a physical page has been previously used by a different process [15]. However, kernel memory allocation functions like `kmalloc()` do not get zeroed as they are intended for kernel-space use only; this has obvious performance benefits. Unfortunately, the introduction of ION and its user-space and kernel-space buffer sharing capability effectively breaks the assumption.

**Static analysis tools on Android**. Static taint analysis is one of the most popular techniques used to analyze and vet Android apps [22, 21, 12, 16, 17, 26]. For instance, Chex [22] can statically analyze the byte code of Android apps for component hijacking vulnerabilities. AAPL [21] compares the the produced information flows for apps in similar categories (*e.g.,* news) to identify suspicious apps with excessive information flows compared to others. Besides taint analysis, many other static analysis tools are built to discover vulnerabilities. Woodpecker [18] analyzes apps to look for capability leaks (*e.g.,* through Intent) that allow confused-deputy attacks. At the system layer, Kratos [25] analyzes

Android framework and look for inconsistent security policy enforcements. Even though static taint analysis has been used widely to analyze apps, it is rarely used to analyze the Android/Linux kernel. Our work applies static taint analysis in a novel setting to identify unzeroed memory pages allocated and returned to user space through ION.

# 9. CONCLUSION

In this paper, we report multiple vulnerabilities of the ION memory management system that can lead to either DoS or sensitive information leakage on virtually all Android devices to date. We build a novel static taint analysis tool to uncover the unzeroed ION heap vulnerabilities systematically. To demonstrate the seriousness of the vulnerabilities, we build exploits against several latest Android devices running latest Android operating systems, including Nexus 6P, Samsung Galaxy S7, and Huawei Mate 8 that run Android 6.0 and even 7.0 preview. In addition, we analyze and digest the root causes of the vulnerabilities in depth. Finally, we outline the defense strategies that have different tradeoffs which can shed light on future design of such a large and complex memory management system.

# 10. REFERENCES


[1] https://sites.google.com/a/androidionhackdemo.net/androidionhackdemo/.

[2] Android Low Memory Killer. https://android.googlesource.com/kernel/common.git/+/android-3.4/drivers/staging/android/lowmemorykiller.c.

[3] Android MediaServer Bug Traps Phones in Endless Reboots. http://blog.trendmicro.com/trendlabs-security-intelligence/android-mediaserver-bug-traps-phones-in-endless-reboots/.

[4] Arm TrustZone Technology. http://www.arm.com/products/processors/technologies/trustzone/.

[5] Device Tree. https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt.

[6] Integer Overflow leading to Heap Corruption while Unflattening GraphicBuffer. http://seclists.org/fulldisclosure/2015/Mar/63.

[7] STAC - Static Taint Analysis for C. http://code.google.com/p/tanalysis/.

[8] The Android ION memory allocator. https://lwn.net/Articles/480055/, 2012.

[9] Patch: sparc32: dma_alloc_coherent must honour GFP_ZERO. https://patchwork.ozlabs.org/patch/386217/, 2014.

[10] Y. Aafer, X. Zhang, and W. Du. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *USENIX SECURITY*, 2016.

[11] A. ARMANDO, A. MERLO, M. MIGLIARDI, , and L. VERDERAME. Would You Mind Forking this Process? A Denial of Service Attack on Android (and Some Countermeasures). In *Information S&P Research*, 2016.

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *PLDI*, 2014.

[13] L. Cavallaro, P. Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *DIMVA*, 2008.

[14] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. of USENIX Security*, 2014.

[15] N. Douglas. User Mode Memory Page Allocation: A Silver Bullet For Memory Allocation? Technical report, 2011.

[16] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *TRUST*, 2012.

[17] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[18] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012.

[19] H. Huang, S. Zhu, K. Chen, and P. Liu. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In *CCS*, 2015.

[20] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *Oakland*, 2014.

[21] K. Lu, Z. Li, V. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting. In *NDSS*, 2015.

[22] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *CCS*, 2012.

[23] M. Nazarewicz. A Deep Dive into CMA. https://lwn.net/Articles/486301/.

[24] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. GUITAR: Piecing Together Android App GUIs from Memory Images. In *CCS*, 2015.

[25] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *NDSS*, 2016.

[26] F. Wei, S. Roy, X. Ou, et al. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *CCS*, 2014.

[27] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The Impact of Vendor Customizations on Android Security. In *CCS*, 2013.

[28] H. Zhang, D. She, and Z. Qian. Android Root and Its Providers: A Double-Edged Sword. In *CCS*, 2015.

[29] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Oakland*, 2014.