Android's Performance Anxiety: An In-Depth Analysis of the Stagefright Bugs

Acknowledgement

First and foremost, I would like to thank my advisor, Professor James Waldo. This thesis would not have been possible without his feedback, encouragement, and anecdotes. Thank you to Professors Margo Seltzer and Steven Chong for agreeing to be readers for this thesis. Last but not least, I would like to thank Shannon Lytle, Zarmeena Dawood, and Jack Kleinman for their feedback and support throughout the writing process.

Contents

Chapter 1

Introduction

With 2.1 billion smartphone users in the world, the transmission of sensitive data through mobile devices has generated increased concern about mobile security [1]. An approximated 1.4 billion of these devices currently use the Android operating system [2]. Thus, security concerns surrounding the Android operating system are applicable to more than 60% of the market and have a huge impact on collective mobile security.

In recent years, major security vulnerabilities have exposed millions of Android devices to data breaches. In June of 2015, a collective group of bugs called Stagefright was discovered in Android versions 2.2 and newer. The bugs acted as backdoors for remote code execution and privilege escalation leading many to call it "the biggest Android security concern ever" [3]. This collective group of bugs exploited vulnerabilities in the Stagefright library, which processes media in the MediaServer process. Although patches for these bugs have been applied, many devices - particularly the older OS versions- have yet to receive them. The significance of these bugs is obvious, because millions of devices are still vulnerable. The bugs themselves offer great insight into the weaknesses of the MediaServer process for which new vulnerabilities are being discovered affecting Android version 5.1 and newer.

In this paper we test the existence of the Stagefright bugs across three versions of the Android OS: 4.4.4, 5.1.1, and 6.0. We analyze the code in each operating system to determine if the vulnerability still exists, and if patches besides a code vulnerability fix were able to circumvent the exploit. We also hypothesize what these patches could be if the OS is vulnerable but not exploitable. An in depth analysis of why these vulnerabilities exist is given in part 3. We also discuss vulnerabilities that have been discovered in 2016 regarding the MediaServer process, and how these vulnerabilities are related.

Chapter 2

Background

2.1 Definitions and Terms

When discussing the Stagefright library, we will refer to it as 'libStagefright'. The collective bugs are referred to as 'Stagefright'. We define a vulnerability as a weakness in the system which can be taken advantage of to produce unwanted behavior. An exploit is the method by which that vulnerability is explored. In a similar vein, an attack surface is the vulnerability in the code. An attack vector is the exploit that takes advantage of the vulnerability.

2.2 Android Security Architecture

Android is an open source operating system developed by Google, with source code available under the Android Open Source Project (AOSP) [4]. The source code is often adapted by Original Equipment Manufacturers (OEMs) to run on specific devices with modifications and proprietary software. The modifications to the source code may include changing native processes permissions.

In order to understand security vulnerabilities in the Android OS, we must first explore the Android security model. Android is built upon the Linux kernel, which provides drivers for hardware, process management, networking, and file system access. The Linux kernel for Android is slightly different from 'regular' Linux kernels because of features called "Androidisms" which allow the kernel to adapt to mobile systems. These changes include low memory killer, anonymous shared memory, and paranoid networking [5]. On top of the Linux kernel is the native userspace layer, which consists of several native daemons and libraries. On

top of the userspace layer is the Java Virtual Machines, called Dalvik in Android's current implementations. Dalvik allows multiple instances of virtual machines to be created simultaneously, which provides security, isolation, and memory management. At the highest level of the Android OS is the Application Layer, which includes pre-installed apps and 3rd party apps.

Figure 2.1: Android software stack [6].

At the most basic level, Android provides security through the Linux Kernel, application sandboxing, secure interprocess communication, application signing, and application defined and user-granted permissions. Android's native libraries and applications naturally have high privileges [7].

To understand the accesses that applications generally get when installed on your phone, let us explore the process of installing an application (intentionally or not). At the kernel level, each app is assigned a UID (user ID) and GID (group ID) at install time and updates. Android doesn't use Linux users and UIDs in

the usual way. There are "virtual users" that correspond to each app and when an app is installed, it gets a UID and an identical GID. The mapping from app to UID can be found in the /system/packages.list file, which lists the app name, UID and the location to the app's private data storage in the filesystem. Multiple unique apps signed with the same key can run under the same UID, but they are considered by the system to be part of the same app. Application data files are stored in $/data/data/app-name/$, and are read-writable only by that application process. When the Java VM, Dalvik, loads an application, it changes to the UID and GID for the application, so that the process is running in the correct security context. This ensures that applications are not able to get access to processes, memory, or data that they do not have the permission to obtain.

2.3 libStageFright & MediaServer

Since this paper focuses on vulnerabilities in the MediaServer process, it is important to understand the architecture of the MediaServer and the affected libraries.

2.3.1 libStageFright History

In Android version 1.0, the operating system's main multimedia framework was OpenCORE, which was also primarily in C++. In version 2.0, libStagefright was created and added to the AOSP and used optionally in devices. By Android version 2.2, most devices used Stagefright as it became regarded as the superior multimedia framework. It was set as the default engine in versions 2.3 and later. The Stagefright library was also forked and used by Firefox around the time of its creation as well [8].

Stagefright is Android's multimedia framework library, written primarily in C++ (and some Java). It supports the handling of video and audio files, but does not handle images. Its main functionality is to provide the playback facilities for video and audio files and extract metadata, such as thumbnail, dimensions, frame rate, etc.

2.3.2 MediaServer Architecture

The architecture of the media playback infrastructure is similar to the OS architecture. User apps are at the upper-most layer, followed with the application framework (the Audio manager, media player, etc.). This lays upon the Java Native interface, which includes the C++ implementation of the media player. The

native components are the clients that forward media requests from Java code to the underlying media placer service using interprocess communication (IPC) calls. The Media player service then selects the appropriate media player, which happens to be Stagefright in our experiments.

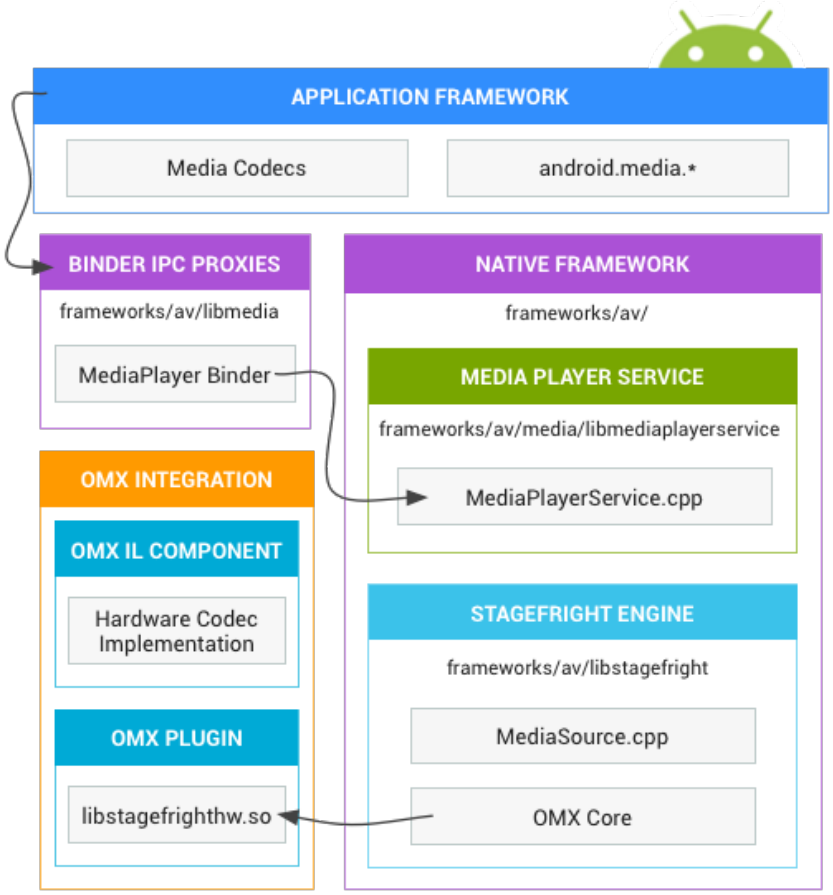

Figure 2.2: Media architecture [9].

2.3.3 MediaServer Priveleges

libStagefright processes media inside the MediaServer. As such, it has the same privileges as MediaServer. The MediaServer process runs in the background of all Android machines. It is a native service that is started at boot from */init.rc*. When a program started by init crashes, init will automatically restart it. As a result, the MediaServer process can be restarted infinitely if something causes it to crash over and over.

MediaServer belongs to the user class 'media' and group of audio, camera, in-

ternet, bluetooth, bluetooth admin, dam remote procedure call in all devices. It may belong to more groups depending on the manufacturer of the device. At a basic level, being a member of these groups allows the library to communicate with those processes and access the functionalities of these groups. Hence the MediaServer can connect to hosts on the internet; access and configure bluetooth; monitor, record, and playback audio; access the camera; etc. Depending on the manufacturer, some devices may have system access through MediaServer, which is basically one level below root, and are able to approve permissions for applications.

Chapter 3

Stagefright Explained

3.1 Overview

In June of 2015, Joshua Drake of Zimperium security firm publicly announced a group of bugs (seven in total) in the Android OS (versions 2.2 and newer) that acted as backdoors for remote code execution and privilege escalation. The collective bugs, which exploit the Android Stagefright library, were appropriately dubbed "Stagefright". The first bug was exploited by sending an MMS message containing a corrupt MP4 file that the phone would automatically retrieve and execute through the Android media handler. The MP4 file, containing malicious code, is automatically executed before the user has a chance to view the file due to the way the Android OS handles MMS messages.

Google's initial patch was that users could avoid MMS auto-retrieval by simply changing their settings, thus not executing the malicious code automatically. They also put out a statement that Android 4.0 and newer feature Address Space Layout Randomizing (ASLR) that prevents Stagefright attacks by changing the RAM any app uses. However, this technology was also circumvented, which we discuss in section 5. The venue of the exploits also varied, as the malicious code could be executed through the browser, apps, and more.

The real vulnerability was not the MMS auto-retrieval, but rather the weaknesses in the code which allowed malicious files to be parsed in the first place. libStagefright itself is a complex software library implemented in C++ that is used as a backend engine for playing various multimedia formats. Google quickly implemented patches for newer Android OS, however, the firmware updates for various Android devices were the responsibility of wireless carriers and OEMs. Thus, the patches took a longer time to be distributed to users. Some older and

cheaper devices were also never fully patched, if at all. The implication of the Stagefright bugs highlighted the organizational difficulties in propagating Android patches.

In the following sections, we begin by explaining the format of an MP4 file, which is relevant to the libStagefright (and to the Stagefright bugs) as the library is the primary dispatch for handling MP4 atoms. In section 4 we also discuss the choice for using MP4 files as our attack vector. Following this, we give detailed descriptions of each of the Stagefright vulnerabilities and compare their existence in the source code across the three OS versions. We begin by examining five of the seven vulnerabilities that are found in the MPEG4Extractor.cpp in libStagefright. We then examine the other two remaining vulnerabilities.

3.2 MP4 File Structure

An MP4 file is made up of discrete units, formerly called atoms, now referred to as 'boxes'. An atom has the following format:

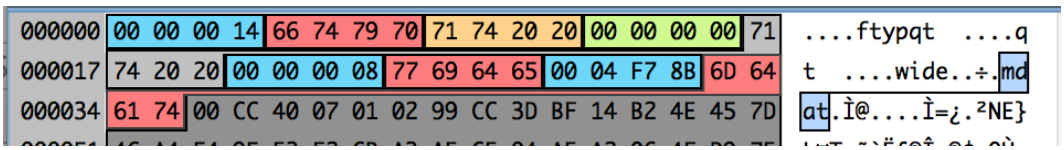

Figure 3.1: Atom format [10].

An atom is structured to have a 4 byte that designates the length of the atom (blue boxes), followed by the atom name (red boxes). The boxes after that are optional and are often defined by the atom itself. For any media file using atoms, the 'ftyp' atom is first because it holds information about what type of file it is [10].

The following is a list of the atoms relevant to the Stagefright bugs as well as a short description of their functions [11]:

- 'tx3g' - contains a wide range of metadata boxes that are text samples
- '3GPP' - parent to atoms 'albm', 'auth',
- 'covr' - album cover artwork
- 'stsc' - defines the sample-to-chunk mapping in the sample table of a media track

- ‘ctts’ - composition time to sample mapping for a sample table
- ‘stts’ - time-to-sample mapping for a sample table
- ‘stss’ - optional, specifies which samples within a sample table are sync samples
- ‘ESDS’ - contains initialization data that a decoder requires to decode the stream

3.3 Internal Bug 20923261

The vulnerabilities grouped under internal bug 20923261 are all located in the file MPEG4Extractor.cpp in libStagefright. Some of these vulnerabilities are not applicable to version 4.4.4 because the code had not yet been added to the Android source, which we noted in the descriptions. In each section, we first give the vulnerability summary as it exists in the National Vulnerability Database [12]. We then provide the vulnerable code alongside descriptions of how the vulnerability works, as well as the patches that have been applied to fix it. All code in this section is from the Android XRef project [13].

3.3.1 CVE-2015-3824

> CVE-2015-3824: The MPEG4Extractor::parseChunk function in MPEG4Extractor.cpp in libStagefright in Android before 5.1.1 LMY48I does not properly restrict size addition, which allows remote attackers to execute arbitrary code or cause a denial of service (integer overflow and memory corruption) via a crafted MPEG-4 tx3g atom, aka internal bug 20923261 [14].

CVE-2015-3824 is a vulnerability with the way tx3g atoms in the MPEG4Extractor::parseChunk function are handled. The source code as found in version 4.4.4 is as follows:

```
1717 case FOURCC('t', 'x', '3', 'g'):
1718 {
1719   uint32_t type;
1720   const void *data;
1721   size_t size = 0;
1722   if (!mLastTrack->meta->findData(
```

```
1723            kKeyTextFormatData, &type, &data, &size)) {
1724        size = 0;
1725    }
1726
1727    uint8_t *buffer = new uint8_t[size + chunk_size];
1728
1729    if (size > 0) {
1730        memcpy(buffer, data, size);
1731    }
1732
1733    if ((size_t)(mDataSource->readAt(*offset, buffer +
        ↪ size, chunk_size))
1734            < chunk_size) {
1735        delete[] buffer;
1736        buffer = NULL;
1737
1738        return ERROR_IO;
1739    }
1740
1741    mLastTrack->meta->setData(
1742            kKeyTextFormatData, 0, buffer, size +
                ↪ chunk_size);
1743
1744    delete[] buffer;
1745
1746    *offset += chunk_size;
1747    break;
1748 }
```

When handling the addition of *size* and *chunk_size* in line 1727, there is no check to see if the result is larger than 2^{32}, causing an integer overflow to occur. The result is then truncated to fit the 32 bit value. When allocating memory based on that result, it creates buffer allocations with less than the necessary memory. When *chunk_size* worth of data from the 'tx3g' atom is read into the buffer, which is allocated on the heap, it will overrun the buffer's boundary and start writing into adjacent memory. This is all executed when *memcpy* is called in line 1730. The *memcpy* method copies data into the destination (first parameter) from the object pointing to the source of the data (second parameter) by counting the number of bytes to copy (third parameter). The *memcpy* method also returns the address of the destination. Because the third parameter will be larger than the size of the data buffer, it results in a buffer over-read. If the MP4 file is crafted in such a

way that the data that is copied into the heap past the boundaries of the buffer is arbitrary code, the attacker may be able to exploit this vulnerability in addition to other existing vulnerabilities to execute the code. However, it is not enough to just write arbitrary code onto the heap, as Android versions 2.3 and newer have executable space protection, which we will discuss in detail in section 5.

The vulnerability still exists in the source code for version 5.1.1, as there is no check to see if $size + chunk_size$is greater than 2^{32}. However, we notice that the line in which the vulnerability occurs (1896) has been modified:

```
1886 case FOURCC('t', 'x', '3', 'g'):
1887 {
1888   uint32_t type;
1889   const void *data;
1890   size_t size = 0;
1891   if (!mLastTrack->meta->findData(
1892       kKeyTextFormatData, &type, &data, &size)) {
1893     size = 0;
1894   }
1895
1896   uint8_t *buffer = new (std::nothrow) uint8_t[size +
         ↪ chunk_size];
1897   if (buffer == NULL) {
1898     return ERROR_MALFORMED;
1899   }
```

In this version of the OS, there is an attempt to allocate memory using the *new* operator, and this time, *nothrow* is passed in as an argument. If the attempt to allocate memory fails, normally a *bad_alloc* exception is thrown. When *nothr ow* is passed in as an argument, upon failure, the new operator should return a null pointer instead of throwing an exception. The following if-statement in lines 1897 to 1899 then check to see if the buffer indeed failed to allocate and return an error.

In version 6.0, the code finally reflects the required patches. An if-statement checks to make sure the boundaries of the buffer will not be over read (lines 2010-2012).

```
2009 ...
2010 if ((chunk_size > SIZE_MAX) || (SIZE_MAX - chunk_size <=
       ↪  size)) {
2011  return ERROR_MALFORMED;
2012 }
2013
```

```
uint8_t *buffer = new (std::nothrow) uint8_t[size +
    ↪ chunk_size];
if (buffer == NULL) {
 return ERROR_MALFORMED;
}
...
```

3.3.2 CVE-2015-3826 and CVE-2015-3828

These two vulnerabilities are related issues and only exist in libStagefright in Android v. 5.0 and newer.

> CVE-2015-3826: The MPEG4Extractor::parse3GPPMetaData function in MPEG4Extractor.cpp in libstagefright in Android before 5.1.1 LMY48I does not enforce a minimum size for UTF-16 strings containing a Byte Order Mark (BOM), which allows remote attackers to cause a denial of service (integer underflow, buffer over-read, and mediaserver process crash) via crafted 3GPP metadata, aka internal bug 20923261, a related issue to CVE-2015-3828 [15].
>
> CVE-2015-3828: The MPEG4Extractor::parse3GPPMetaData function in MPEG4Extractor.cpp in libstagefright in Android before 5.1.1 LMY48I does not enforce a minimum size for UTF-16 strings containing a Byte Order Mark (BOM), which allows remote attackers to execute arbitrary code or cause a denial of service (integer underflow and memory corruption) via crafted 3GPP metadata, aka internal bug 20923261, a related issue to CVE-2015-3826 [16].

Both vulnerabilities arise from the parse3GPPMetaData function, which is called when dealing with the following atoms:

```
case FOURCC('t', 'i', 't', 'l'):
case FOURCC('p', 'e', 'r', 'f'):
case FOURCC('a', 'u', 't', 'h'):
case FOURCC('g', 'n', 'r', 'e'):
case FOURCC('a', 'l', 'b', 'm'):
case FOURCC('y', 'r', 'r', 'c'):
{
    *offset += chunk_size;

```

```
1954        status_t err = parse3GPPMetaData(data_offset,
                ↪ chunk_data_size, depth);
1955
1956        if (err != OK) {
1957            return err;
1958        }
1959
1960        break;
```

The vulnerable code in parse3GPPMetaData is below, where the variable *chunk_data_size* is passed in as parameter *size*:

```
2468 if (metadataKey > 0) {
2469     bool isUTF8 = true; // Common case
2470     char16_t *framedata = NULL;
2471     int len16 = 0; // Number of UTF-16 characters
2472
2473     // smallest possible valid UTF-16 string w BOM: 0xfe
             ↪  0xff 0x00 0x00
2474     if (size - 6 >= 4) {
2475         len16 = ((size - 6) / 2) - 1; // don't include 0
                 ↪ x0000 terminator
2476         framedata = (char16_t *)(buffer + 6);
2477         if (0xfffe == *framedata) {
2478             // endianness marker (BOM) doesn't match
                     ↪ host endianness
2479             for (int i = 0; i < len16; i++) {
2480                 framedata[i] = bswap_16(framedata[i]);
2481             }
2482             // BOM is now swapped to 0xfeff, we will
                     ↪ execute next block too
2483         }
2484
2485         if (0xfeff == *framedata) {
2486             // Remove the BOM
2487             framedata++;
2488             len16--;
2489             isUTF8 = false;
2490         }
2491         // else normal non-zero-length UTF-8 string
2492         // we can't handle UTF-16 without BOM as there
                 ↪ is no other
```

```
2493            // indication of encoding.
2494        }
2495
2496        if (isUTF8) {
2497            mFileMetaData->setCString(metadataKey, (const
                    ↪ char *)buffer + 6);
2498        } else {
2499            // Convert from UTF-16 string to UTF-8 string.
2500            String8 tmpUTF8str(framedata, len16);
2501            mFileMetaData->setCString(metadataKey,
                    ↪ tmpUTF8str.string());
2502        }
2503    }
2504
2505    delete[] buffer;
2506    buffer = NULL;
2507
2508    return OK;
```

The integer underflow occurs when the *len*16 variable is created by subtracting 6 from *size* in line 2475. Because there is no check to make sure *size* is not less than 6, if the *size* variable is less than 6, subtracting 6 from it results in less than the minimum allowable integer value (2^{16}). This integer underflow causes the value to wrap around and create a very large value. *len*16 is then used for a linear byteswap operating for decoding the framedata (code lines 2479 to 2481). This vulnerability allows memory to be read past the end of the heap memory, and in lines 2497and 2501, the *setCString*() method is called to copy the *metadataKey* into memory. The *setCString*() method eventually calls *memcpy*(). As described in section 3.3.1, this will lead to a buffer over-read, and may lead to execution of arbitrary code if exploited with other vulnerabilities.

3.3.3 CVE-2015-3827 and CVE-2015-3829

CVE-2015-3827: The MPEG4Extractor::parseChunk function in MPEG4Extractor.cpp in libStagefright in Android before 5.1.1 LMY48I does not validate the relationship between chunk sizes and skip sizes, which allows remote attackers to execute arbitrary code or cause a denial of service (integer underflow and memory corruption) via crafted MPEG-4 'covr' atoms, aka internal bug 20923261 [17].

CVE-2015-3829: Off-by-one error in the MPEG4Extractor::parseChu-

nk function in MPEG4Extractor.cpp in libStagefright in Android before 5.1.1 LMY48I allows remote attackers to execute arbitrary code or cause a denial of service (integer overflow and memory corruption) via crafted MPEG-4 covr atoms with a size equal to $SIZE\_MAX$, aka internal bug 20923261 [18].

CVE-2015-3827 and CVE-2015-3829 address vulnerabilities in how the MPEG4Extractor::parseChunk method handles "covr" atoms. The vulnerable code in version 4.4.4 is as follows:

```
1750 case FOURCC('c', 'o', 'v', 'r'):
1751 {
1752     if (mFileMetaData != NULL) {
1753         ALOGV("chunk_data_size = %lld and data_offset =
                 ↪ %lld",
1754                 chunk_data_size, data_offset);
1755         sp<ABuffer> buffer = new ABuffer(chunk_data_size
                 ↪  + 1);
1756         if (mDataSource->readAt(
1757             data_offset, buffer->data(), chunk_data_size
                     ↪ ) != (ssize_t)chunk_data_size) {
1758             return ERROR_IO;
1759         }
1760         const int kSkipBytesOfDataBox = 16;
1761         mFileMetaData->setData(
1762             kKeyAlbumArt, MetaData::TYPE_NONE,
1763             buffer->data() + kSkipBytesOfDataBox,
                     ↪ chunk_data_size - kSkipBytesOfDataBox)
                     ↪ ;
1764     }
1765
1766     *offset += chunk_size;
1767     break;
1768 }
```

CVE-2015-3827 is a vulnerability caused by the operation in line 1763, when the size of *chunk_data_size* is not checked before subtracting *kSkipBytesOfDataBox* from it. This has the potential to cause an integer underflow if *chunk_data_size* is less than *kSkipBytesOfDataBox*, and results in a very large value being passed into MetaData::setData:

```
190 bool MetaData::setData(
```

```
191         uint32_t key, uint32_t type, const void *data,
              ↪ size_t size) {
192     bool overwrote_existing = true;
193
194     ssize_t i = mItems.indexOfKey(key);
195     if (i < 0) {
196         typed_data item;
197         i = mItems.add(key, item);
198
199         overwrote_existing = false;
200     }
201
202     typed_data &item = mItems.editValueAt(i);
203
204     item.setData(type, data, size);
205
206     return overwrote_existing;
207 }
```

MetaData::setData then calls item.setData with the type, data, and size in line 204, which does the following:

```
258 void MetaData::typed_data::setData(
259         uint32_t type, const void *data, size_t size) {
260     clear();
261
262     mType = type;
263     allocateStorage(size);
264     memcpy(storage(), data, size);
265 }
```

When the size variable is very large, the *typed_data* :: *setData* method allocates more storage than necessary and copies the data into the allocated storage. Once again, *memcpy* method copies data into storage when the size variable is very large, allowing for memory corruption or in extreme cases, execution of arbitrary code.

This vulnerability is easily patched, but the fix is not present until version 6.0.0, as seen below:

```
2042 case FOURCC('c', 'o', 'v', 'r'):
```

...

```
2059            if (chunk_data_size <= kSkipBytesOfDataBox) {
2060                return ERROR_MALFORMED;
2061            }
```

CVE-2015-3829 concerns the same code that handles 'covr' atoms presented earlier in section 3.3.3. In this vulnerability, an integer overflow occurs if *chunk_data_size* is $SIZE\_MAX$ in line 1755, because the buffer is allocated to be of size $chunk\_data\_size + 1$. When the integer overflow occurs, the value gets wrapped around to be 0, causing a buffer of size 0 to be allocated. The code also does not include any checks to make sure the buffer is allocated correctly, so the following memory operations in which the data is copied into the undersized buffer causes a buffer overflow.

This vulnerability also has a simple patch, which is not reflected until version 6.0.0, which is to check to make sure the *chunk_data_size*is not greater than or equal to $SIZE\_MAX - 1$, as shown below.

```
2042 case FOURCC('c', 'o', 'v', 'r'):
```

...

```
2050            if (chunk_data_size < 0 || static_cast<uint64_t
                    ↪ >(chunk_data_size) >= SIZE_MAX - 1) {
2051                return ERROR_MALFORMED;
2052            }
2053            sp<ABuffer> buffer = new ABuffer(chunk_data_size
                    ↪  + 1);
```

3.4 Internal Bug 20139950

Internal Bug 20139950 refers to a couple of bugs in various files in libStagefright. The ones that are grouped into the Stagefright bugs are discussed here:

3.4.1 CVE-2015-1538

> CVE-2015-1538: Integer overflow in the SampleTable::setSampleToChunkParams function in SampleTable.cpp in libstagefright in Android before 5.1.1 LMY48I allows remote attackers to execute arbitrary code via crafted atoms in MP4 data that trigger an unchecked multiplication, aka internal bug 20139950, a related issue to CVE-2015-4496 [19].

This vulnerability is an integer overflow vulnerability that can be exploited through four different atoms: ‘stsc’, ‘ctts’, ‘stts’, and ‘stss’. The line of code (as it exists in version 4.4.4) which causes this vulnerability is in the setSampleToChunkParams function:

```
233 mSampleToChunkEntries = new SampleToChunkEntry[
    ↪ mNumSampleToChunkOffsets];
```

Because there is no explicit check to validate that mNumSampleToChunkOffets will not cause an integer overflow, the vulnerable line could cause a smaller buffer than necessary to be allocated for any of the four atoms affected, allowing for data to be written past the buffer's boundaries.

The necessary fix for this is reflected in version 6.0 with the following patch:

```
238 if (SIZE_MAX / sizeof(SampleToChunkEntry) <= (size_t)
    ↪ mNumSampleToChunkOffsets)
239     return ERROR_OUT_OF_RANGE;
```

3.4.2 CVE-2015-1539

> CVE-2015-1539: Multiple integer underflows in the ESDS::parseESDescriptor function in ESDS.cpp in libStagefright in Android before 5.1.1 LMY48I allow remote attackers to execute arbitrary code via crafted ESDS atoms, aka internal bug 20139950, a related issue to CVE-2015-4493 [20].

The last of the Stagefright vulnerabilities concerns multiple opportunities for integer underflows to occur in the ESDS::parseESDescriptor function in ESDS.cpp. The following code reflects those vulnerabilities in version 4.4.4:

```
138 if (streamDependenceFlag) {
139         offset += 2;
140         size -= 2;
141     }
142 
143     if (URL_Flag) {
144         if (offset >= size) {
145             return ERROR_MALFORMED;
146         }
147         unsigned URLlength = mData[offset];
148         offset += URLlength + 1;
```

```
149         size -= URLlength + 1;
150     }
151
152     if (OCRstreamFlag) {
153         offset += 2;
154         size -= 2;
```

The value of variable *size* is not checked before various operations are carried out with it. In the affected code, subtraction operations in lines 140, 149, and 154 may cause an integer underflow to occur, causing the value to wrap around and creating a very large number. This would cause an out of bounds read operation later on when *size* is passed into another function that reads data into memory.

Chapter 4

Experiments

4.1 Testbed

All experiments were carried out through Android machines installed through a virtual machine. The VMs ran on Oracle VirtualBox, version 5.0.14.

4.1.1 Android Versions

The decision to use Android versions 4.4.4, 5.1.1 and 6.0.0 arose from the relevance of these versions in the market. Android version 4.4 is currently the most prevalent version in the market, being currently active in 35% of devices. Additionally, because Stagefright was discovered in versions 2.2 and newer, we decided that the results would be similar for versions between 2.2 and 4.4, and thus was not worth considering each version separately. In previous tests of the Stagefright vulnerabilities discovered that the vulnerabilities were not exploitable in version 5.1. This was an interesting result, considering many of the vulnerabilities were present in the source code. Thus, we found it relevant to test version 5.1 to better understand the vulnerabilities. Lastly, in order to understand the vulnerabilities in the most current OS, Android 6.0.0 (Marshmallow) was also chosen for the experiments.

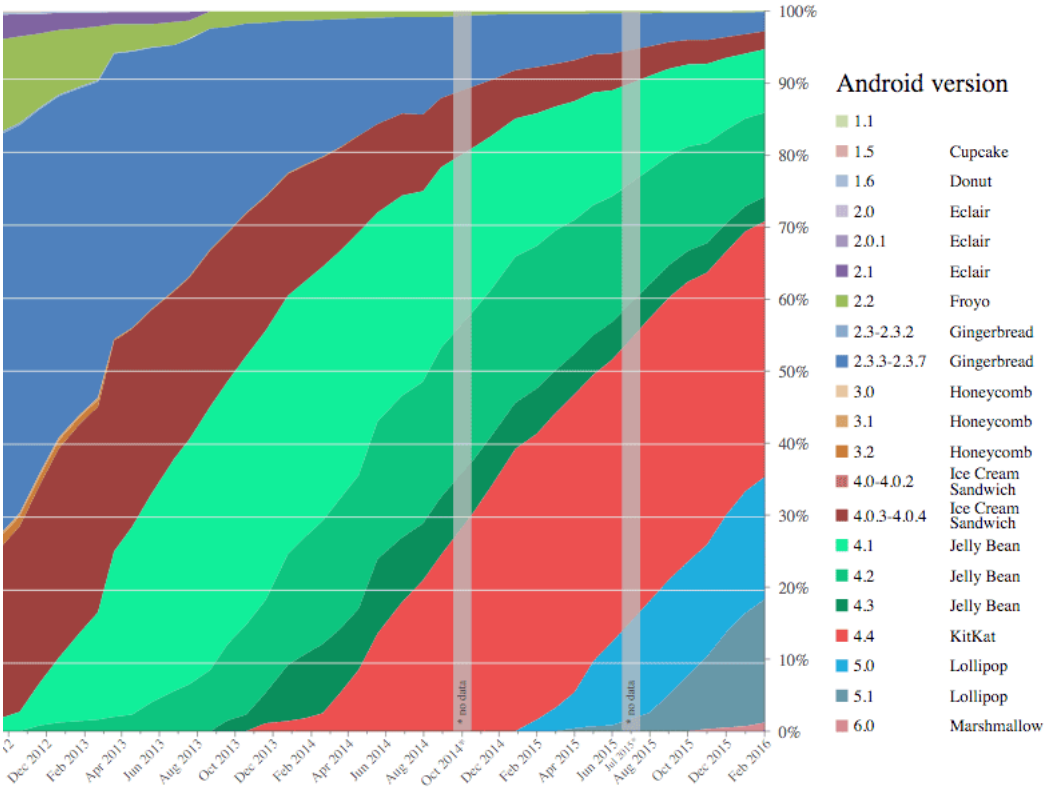

Figure 4.1: Android OS versions [21].

4.1.2 System Configurations

The Android versions installed to run in VirtualBox were configured to their respective recommended settings.

- Version 4.4.4: The system was configured to have 1 processor, with 1000 MB of base memory.
- Version 5.1.1: The system was configured to have 2 processors, with 2048 MB of base memory.
- Version 6.0.0: The system was configured to have 4 processors, with 2048 MB of base memory.

4.2 Stagefright Exploits

When considering the set of bugs that Stagefright is comprised of, there are two main trajectories we have to consider: the attack surface and the attack vector. Our attack surface consisted of the vulnerabilities discussed in section 3.

4.2.1 Attack Vector

The attack vector is varied and any reasonable way to execute the vulnerable code is fair game. In our experiments, we decided to execute our exploits through MP4 files. While libStagefright handles multiple media formats and many of the vulnerabilities were exploitable by multiple media formats, they were all exploitable by MP4 files. Thus, we chose the attack vector that was applicable to all of the vulnerabilities. Because the Stagefright vulnerabilities are related to media parsing (collecting information about thumbnail, video length, etc. from MP4 metadata atoms), the video file just needed to have some way of being parsed. Parsing can occur if the media file is embedded in a web page, downloaded by the cellphone, or sent over a messaging service. The attack vector may also vary for device by manufacturer depending on what process calls automatic retrieval of the media files.

For our experiments, the media was presented through a Dropbox link, which allowed the video to be embedded in the webpage, causing libStagefright to render it when the page loaded. We also tested the vulnerabilities by downloading the MP4 files onto each virtual device and rendering them by trying to play the video file. Both methods produced the same results.

The exploits for Stagefright consisted of a separate MP4 file for each exploit (including separate files for each atom the vulnerable code in CVE-2015-1538 processes). The proof of concept MP4 files for Stagefright have been released by Zimperium [22], and they were used as a baseline for our experiments. However, we also modified and ran python scripts [23] to create our own proof of concept files that exploited each Stagefright bug. Both methods produced the same results, presumably because they both constructed MP4 files in the same method and directly attempted to exploit the vulnerabilities.

4.3 Results

Our results are as follows:

Vulnerability ID	Android v. 4.4	Android v. 5.1	Android v. 6.0
1538 - 'stsc'	E	V, NE	N/A
1538 - 'ctts'	E	V, NE	N/A
1538 - 'stts'	E	V, NE	N/A
1538 - 'stss'	E	V, NE	N/A
1539	N/A	V, NE	N/A
3827	E	V, NE	N/A
3826	N/A	E	N/A
3828	N/A	V, NE	N/A
3824	E	V, NE	N/A
3829	E	V, NE	N/A

Table 4.1: Table of results: Orange cells indicate successful exploits. Blue cells indicate existing vulnerabilities that we failed to exploit. N/A indicates that the vulnerability did not exist in the codebase.

Of the vulnerabilities that existed in version 4.4.4, we were able to exploit all of them. Only one of the 10 exploits which were present in version 5.1.1 was successfully exploited. And since none of the vulnerabilities were present in version 6.0.0, none of the exploits produced any malicious result.

4.4 adb logs

In order to fully understand what happened when each MP4 file rendered, we looked at the logs of the device through adb. This allowed us to observe a couple of things: If the expected behavior occurred, and the media server crashed, we were able to take note of that. Generally, when a process starts, it is assigned a new PID. When the media server process crashes, it is automatically restarted by *init* and assigned a new PID. Thus, we we observe a change in the PID of the mediaServer, we can assume something caused the process to crash.

The following sections present the most interesting lines of the adb logs that shed light into what happens when the exploits were successfully carried out and a mediaServer crash occurred.

4.4.1 adb Log for CVE-2015-3824

```
F/libc    ( 2779): Fatal signal 11 (SIGSEGV) at 0
  ↪ xdeadbaad (code=1), thread 2787 (Binder_1)
```

```
...
I/DEBUG   ( 1057): signal 11 (SIGSEGV), code 1 (
   ↪ SEGV_MAPERR), fault addr deadbaad
I/DEBUG   ( 1057): Abort message: 'invalid address or
   ↪ address of corrupt block 0x42fcb908 passed to
   ↪ dlfree'
...
```

From the first line of the log, we can see that the traceback of the mediaServer crash was due to a 'SIGSEV', which means that a segmentation fault occurred. This happens when a user tried to access invalid memory. Segmentation faults are common in languages like C++ where there is access to low level memory. This also makes sense because our proof of concept file for CVE-2015-3824 attempted to cause an integer overflow, creating a buffer that would be too small for the data that was read in. This data was then read past the boundary of the buffer, causing a segmentation fault, as we are attempting to access memory we should not be able to. '$SEGV\_MAPERR$' indicates the space in memory which was being passed was not mapped into the address space of the application [24].

The next two lines of the log prove our point, given the error code 'deadbaad' is used by the Android when native heap corruption is detected [24]. Additionally, the abort message makes it clear that whatever address being passed into 'dlfree' is invalid or corrupt. However, this proves that a denial of service is possible with this exploit.

4.4.2 adb Log for CVE-2015-3826

```
F/libc    ( 3118): Fatal signal 4 (SIGILL), code 2,
   ↪ fault addr 0xb74b023d in tid 3133 (generic)
...
I/DEBUG   ( 1081): signal 4 (SIGILL), code 2 (ILL_ILLOPN
   ↪ ), fault addr 0xb74b023d
E/DEBUG   ( 1081): AM write failure (32 / Broken pipe)
I/DEBUG   ( 1081): Abort message: 'frameworks/av/media/
   ↪ libstagefright/foundation/ABitReader.cpp:34
...
```

This one gave us interesting results, as the signal 'SIGILL' indicates that some executable file is corrupted, which usually occurs when a program writes past the bounds of its array or buffer, corrupting other data on the stack. This also proves that denial of service is possible with this exploit, and that because we were able to write into memory that is executable, execution of arbitrary code is also

possible [24].

4.4.3 adb Log for CVE-2015-3828

```
E/GenericSource( 3138): Failed to init from data source!
```

This is one of the two vulnerabilities we were not able to exploit across any of the OS versions.

4.4.4 adb Log for CVE-2015-3827

```
F/libc    ( 2912): Fatal signal 11 (SIGSEGV) at 0
   ↪ x00000000 (code=1), thread 2912 (mediaserver)
...
I/DEBUG   ( 1057): pid: 2912, tid: 2912, name:
   ↪ mediaserver  >>> /system/bin/mediaserver <<<
I/DEBUG   ( 1057): signal 11 (SIGSEGV), code 1 (
   ↪ SEGV_MAPERR), fault addr 00000000
```

This was similar to the result of CVE-2015-3824, and caused a segmentation fault.

4.4.5 adb Log for CVE-2015-3829

```
F/libc    ( 2808): invalid address or address of corrupt
   ↪  block 0x41bef518 passed to dlfree
F/libc    ( 2808): Fatal signal 11 (SIGSEGV) at 0
   ↪ xdeadbaad (code=1), thread 2808 (mediaserver)
```

Once again, we were able to achieve a segmentation fail by corrupting memory.

4.4.6 adb Log for CVE-2015-1538 - ‘stsc’

```
F/libc    ( 1062): heap corruption detected by dlmalloc
F/libc    ( 1062): Fatal signal 6 (SIGABRT) at 0
   ↪ x00000426 (code=-6), thread 1062 (mediaserver)
```

4.4.7 adb Log for CVE-2015-1538 - ‘ctts’

```
F/libc    ( 2816): Fatal signal 11 (SIGSEGV) at 0
   ↪ x41cdf000 (code=2), thread 2816 (mediaserver)
...
I/DEBUG   ( 1057): pid: 2816, tid: 2816, name:
   ↪ mediaserver  >>> /system/bin/mediaserver <<<
```

```
I/DEBUG   ( 1057): signal 11 (SIGSEGV), code 2 (
   ↪ SEGV_ACCERR), fault addr 41cdf000
```

This one is slightly different than the segmentation fault we've already covered because it is a different code, '*SEGV_ACCERR*', which indicates that there were invalid permissions for a mapped object [25].

4.4.8 adb Log for CVE-2015-1538 - 'stts'

```
F/libc    ( 2849): Fatal signal 11 (SIGSEGV) at 0
   ↪ x41cdf000 (code=2), thread 2858 (Binder_1)
...
I/DEBUG   ( 1057): pid: 2849, tid: 2858, name: Binder_1
   ↪  >>> /system/bin/mediaserver <<<
I/DEBUG   ( 1057): signal 11 (SIGSEGV), code 2 (
   ↪ SEGV_ACCERR), fault addr 41cdf000
```

This result was identical to CVE-2015-1538 - 'ctts'.

4.4.9 adb Log for CVE-2015-1538 - 'stss'

```
F/libc    ( 2881): Fatal signal 11 (SIGSEGV) at 0
   ↪ x41ddf000 (code=2), thread 2889 (Binder_1)
...
I/DEBUG   ( 1057): pid: 2881, tid: 2889, name: Binder_1
   ↪  >>> /system/bin/mediaserver <<<
I/DEBUG   ( 1057): signal 11 (SIGSEGV), code 2 (
   ↪ SEGV_ACCERR), fault addr 41ddf000
```

This result was identical to the previous two results.

4.4.10 adb Log for CVE-2015-1539

```
E/GenericSource( 3138): Failed to init from data source!
```

Again, this is one of the two vulnerabilities we were not able to exploit across any of the OS versions.

Chapter 5

Analysis

5.1 Why exploits are avoided

There may be several reasons as to why the vulnerabilities that exist in the code of version 5.1.1 were not able to be exploited. As mentioned before, one of the biggest features that prevents arbitrary code execution in many of these vulnerabilities is the existence of executable space protection. Therefore, the data or arbitrary code that is written into the heap or stack does not pose a danger of being executed. Another possibility is that when processing MP4 files, there is more emphasis on other atoms, such that the lack of one of those atoms indicates to the mediaServer that the file is not in the correct format. This is a likely possibility, as many of the exploits' logs for version 5.5.1 demonstrated that the MP4 file never parsed any of its atoms.

5.1.1 ASLR

One feature that prevents many of the buffer overflows and boundary over-reads is Address Space Layout Randomization. ASLR is a security technique that prevents buffer overflows by randomizing the address at which data is written. As such even if a program has access to a devices memory, it cannot reliably access a specific function in memory. In response to the news of the Stagefright bugs in August of 2015, Google's press release stated that it was unrealistic that so many devices were vulnerable to the bugs because of ASLR. However, very recently a security firm has demonstrated a way to circumvent this measure. In a 32 bit linux system, ASLR is implemented so that every chunk of memory that is written is shifted between 0 and 255 pages. The amount that is shifted is called the ASLR slide,

which is generated during a process startup and does not change. Thus, when a module is written into memory, the next chunk is written at a base address that is shifted down by the ASLR slide. Because this slide is the same for all modules, in order to predict the base address of a malicious function in memory, we only need two pieces of information: the base address of a single module and the ASLR slide. The ASLR slide information is available for specific devices and versions [26].

5.1.2 Why this happens in a C++ Library

The issues that these vulnerabilities demonstrate, namely buffer overflow, integer overflow, integer underflow, and other heap related issues, all relate back to the fact that these operations are done in C++. C++ was designed for systems and applications programming as an extension of C. Unlike object oriented languages, C does not require implementation of garbage collection, which can be used to efficiently manage memory. Because it does not implement garbage collection, it is easier to implement overflows in structures.

Buffer Overflow

When writing data to a buffer, the boundary of the buffer is overrun and the data is written into adjacent memory. This is a common issue in C and C++, because there is no built in protection against accessing any part of memory and no check to make sure the data being written into the designated buffer does not exceed the boundaries. The lack of these two features allows a program to write outside of the designated buffer into memory, which may cause memory corruption. It also allows the program to access what it writes, even if it is in memory that it should not have access to. This is the biggest vulnerability that Stagefright takes advantage of, allowing the malicious MP4 file to write executable code into the memory of the device and then access that code to execute it.

Integer Overflow

Overflows occur when an operation on two integers exceeds the maximum value that the data type can hold. In C++, when the operation on an unsigned integer causes an overflow, it takes the module power of 2. This means that the integers will wrap around on overflow. This may sometimes cause the integer to become negative, causing undefined behavior. With signed integer, it always causes undefined behavior.

Integer Underflow

This is similar to integer overflow, however, underflow occurs when the product of subtracting one value from another causes the value to be less than the minimum allowable integer value. This can happen in signed and unsigned integers. Oftentimes this causes the value to be wrapped around and creates a very large value as a result, as we saw in a few of the Stagefright bugs.

5.1.3 Other vulnerabilities in MediaServer

Unfortunately, as much as we now know about the Stagefright bugs, there are still many new vulnerabilities that have been identified in the mediaServer process that are applicable to Android versions 5.1.1 and newer. Many of the descriptions of these vulnerabilities - for which no known exploit exists (yet) - are similar to the descriptions of the Stagefright bugs. Below is a short list of four of the newest vulnerabilities in the mediaServer.

- CVE-2015-6636: mediaserver in Android 5.x before 5.1.1 LMY49F and 6.0 before 2016-01-01 allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted media file, aka internal bugs 25070493 and 24686670 [27].
- CVE-2015-6616: mediaserver in Android before 5.1.1 LMY48Z and 6.0 before 2015-12-01 allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted media file, aka internal bugs 24630158 and 23882800, a different vulnerability than CVE-2015-8505, CVE-2015-8506, and CVE-2015-8507 [28].
- CVE-2015-8505: mediaserver in Android before 5.1.1 LMY48Z allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted media file, aka internal bug 17769851, a different vulnerability than CVE-2015-6616, CVE-2015-8506, and CVE-2015-8507 [29].
- CVE-2015-8506: mediaserver in Android before 5.1.1 LMY48Z and 6.0 before 2015-12-01 allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted media file, aka internal bug 24441553, a different vulnerability than CVE-2015-6616, CVE-2015-8505, and CVE-2015-8507 [30].

It is clear from the vague descriptions that not much is known about these vulnerabilities specifically, however, the process by which the Stagefright bugs were exploited informs us about a general approach to these bugs as well. This conclusion is not a huge leap or overstatement considering these vulnerabilities exist in the same process as Stagefright, and can be exploited in the same way, via a crafted media file.

5.2 Conclusion

The quick turnaround of vulnerabilities and so called patches makes mobile security a field in which very little documentation occurs. The preference in the industry is for fast code production and patches - which is understandable in the competitive technology landscape. However, when bugs - especially those that are very prevalent in a system - are not well documented, it allows for the same mistakes to be made repeatedly.

For example, vulnerabilities CVE-2015-1539, CVE-2015-3826, and CVE-2015-3828 did not exist in the Android source code until version 5.0. However, the vulnerabilities are very similar to the ones that did already exist in version 4.4.4, as well as some that had already been patched. In fact, a quick look into the patches that Google has released in the last three years shows us that the Stagefright bugs are not the first time (or last) in which integer overflows and underflows and memory corruptions have been a security issue. But because these bugs are not well documented and patches only extend to cover the vulnerability in question, it allows for the same errors to occur.

Our contribution through this thesis was to present a thorough explanation and analysis of the Stagefright bugs. Our hope is that this will better inform researchers in the vulnerabilities of Android's mediaServer libraries and allow for quicker vulnerability patches, and even a proactive approach to security flaws.

Bibliography

[1] Number of smartphone users worldwide from 2014 to 2019 (in millions), 2015. Available at: http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[2] John Callaham. Google says there are now 1.4 billion active Android devices worldwide, September 2015. Available at: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide.

[3] Kris Carlon. New Stagefright security exploit puts a billion Android devices at risk, December 2015. Available at: https://www.androidpit.com/what-is-stagefright-on-android-am-i-affected-and-what-can-i-do.

[4] Android Open Source Project. Available at: https://source.android.com/.

[5] Android Kernel Features, 2016. Available at: https://goo.gl/kqCEuh.

[6] Android Interfaces and Architecture, 2016. Available at: https://source.android.com/devices/index.html.

[7] System and kernel security, 2016. Available at: https://source.android.com/security/overview/kernel-security.html.

[8] Joshua Drake. Stagefright: Scary Code in the Heart of Android, August 2015. Available at: https://www.youtube.com/watch?v=71YP65UANP0.

[9] Media, 2016. Available at: https://source.android.com/devices/media/index.html.

[10] Atoms, Boxes, Parents, Children and hex (oh my). Available at: http://atomicparsley.sourceforge.net/mpeg-4files.html.

[11] Video File Format Specification Version 10. Available at: https://www.adobe.com/content/dam/Adobe/en/...pdf.

[12] National Vulnerability Database. Available at: https://nvd.nist.gov/.

[13] Android Xref. Available at: http://androidxref.com/.

[14] Vulnerability Summary for CVE-2015-3824, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnid=cve-2015-3824.

[15] Vulnerability Summary for CVE-2015-3826, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnid=cve-2015-3826.

[16] Vulnerability Summary for CVE-2015-3828, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnid=cve-2015-3828.

[17] Vulnerability Summary for CVE-2015-3827, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnid=cve-2015-3827.

[18] Vulnerability Summary for CVE-2015-3829, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnid=cve-2015-3829.

[19] Vulnerability Summary for CVE-2015-1538, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnid=cve-2015-1538.

[20] Vulnerability Summary for CVE-2015-1539, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnid=cve-2015-1539.

[21] Android Version History, 2016. Available at: https://goo.gl/DnF3WQ.

[22] Stagefright: Vulnerability Details, Stagefright Detector tool released, August 2015. Available at: https://blog.zimperium.com/stagefright-vulnerability-details-stagefright-detector-tool-released/.

[23] The Latest on Stagefright: CVE-2015-1538 Exploit is Now Available for Testing Purposes, August 2015. Available at: https://blog.zimperium.com/the-latest-on-stagefright-cve-2015-1538-exploit-is-now-available-for-testing-purposes/.

[24] Invalid address passed to dlfree, August 2014. Available at: https://stackoverflow.com/questions/25069186/invalid-address-passed-to-dlfree.

[25] SEGV MAPPER, August 2014. Available at: https://stackoverflow.com/questions/1000002/what-is-segv-maperr.

[26] Hanan Be?er. Metaphor, March 2016. Available at: https://www.exploit-db.com/docs/39527.pdf.

[27] Vulnerability Summary for CVE-2015-6636, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6636.

[28] Vulnerability Summary for CVE-2015-6616, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6616.

[29] Vulnerability Summary for CVE-2015-8505, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-8505.

[30] Vulnerability Summary for CVE-2015-8506, September 2015. Available at: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-8506.