Multimedia Attacks on Android devices using StageFright exploit

Sujal
Department of Computer Engineering
Delhi Technological University
New Delhi, India
e-mail: sujalahrodia_2k13@dtu.ac.in

Sushma Verma
Scientific Analysis Group (SAG)
Defence Research & Development Organization
New Delhi, India
e-mail: sushmaverma@sag.drdo.in

***Abstract*— Android is the most widely used operating system among smartphones today. The security of Android devices is of great concern due to the increased number of applications which is being used worldwide. Despite the effort by android application developers to provide a safe and secure environment to use these devices, the vulnerabilities still exist and are exploited by the attackers leading to security risks such as information leakage. The open-source nature of Android has helped in patching many vulnerabilities but at the same time, new vulnerabilities are discovered. In this paper, we have analyzed the vulnerability related to multimedia Stagefright library which allows an attacker to attack via heap overflow leaking important data to the attacker. We focus on the multiple attack vectors that are used to gain access to the system in order to inject malicious files which contains the shell codes needed to be executed by the victim's device. The main part of exploitation is the concept of heap spraying which involves shaping the heap in such a way that injected files are allocated right after the overflowed buffer. Some weaknesses of ASLR along with the methods of bypassing it are also explored. All this is analyzed in terms of atoms or chunks in mediaserver library which utilizes the metadata of media files to leak important information. Although the Android OS is being continuously fixed by patching and upgrading, many devices are still vulnerable to the exploit mentioned in the paper. Moreover, we present some preventive measures which may be helpful in building a protective environment for android devices.**

Keywords-component; Android vulnerabilities; security; stagefright; ASLR; mediaserver; multimedia

I. INTRODUCTION

Android is Linux kernel mobile platform which serves as an open source software stack for smartphones led by Google. Android OS is highly dependent on the device's processing capabilities in terms of its performance. Majority of the population uses Android in their smartphones, tablets, Tvs and even watches. This makes the security of Android an important aspect. Being an open source project, it is conductive to the use of third-party applications. The ultimate goal is to achieve a secure and usable mobile platform. A malicious software can cause immense harm in terms of getting root privilege and information leakage.
Recent research on the instability and crashes of mediaserver library has been reported. Initially, Android was launched with OPENCORE as its media engine and stagefright was added to the AOSP during the development phase of Android Éclair (2.0). The faster and efficient code led the market to shift towards Stagefright and eliminated OPENCORE from Android 2.2 onwards. Moreover, it is also used by Mac OS X, Firefox and Windows but not Linux as it uses gstreamer. Stagefright, a multimedia framework library has been analyzed for vulnerabilities. Since it is written in native C++ code, the use of various function pointers allows an attacker to cause memory corruptions. The library is highly vulnerable via multiple attack vectors as it can even work without any kind of interaction with the user, which makes it highly insecure. One such example is CVE-2015-3864 [1] which is used to exploit the device.
In this paper, we have analyzed the MPEG4 file format which is a set of multiple chunks used for exploiting the device. Our research is focused on finding out the weaknesses in Address Space Layout Address (ASLR) and possible ways of bypassing it. The MPEG4Extractor.cpp file was examined to find possible vulnerabilities which can cause memory corruptions. The most important part of the exploit is Heap shaping, which provides a way to manipulate the memory addresses and accordingly inject shellcodes at specific predictable locations. These shellcodes are executed by the device and thus bypasses the ASLR. There are various challenges and limitations for this vulnerability which are examined to provide more insight to the whole process. Ultimately exploitation in terms of memory mapping leads to information leakage through the 'duration' field of MetaData , details of which are mentioned in Section IV.
The rest of the paper is organized as follows: Section II discusses the security of Android OS and explains the vulnerabilities in libstagefright (Stagefright). Section III analyses the attack vectors, file information and exploitation techniques used in stagefright. Section IV describes the implementation of the exploit along with the flowchart Section V summarizes our suggestions for preventing memory corruptions and Section VI states the conclusion.

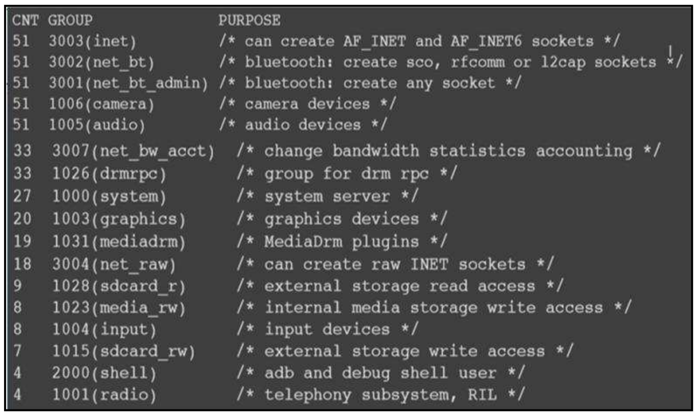
II. ANDROID VULNERABILTIES

The security of an android device is measured by: (1) information on the critical vulnerabilities found to affect particular versions of Android and (2) information on the distribution of Android versions over time[2]. These datasets provide a significant insight to the proportion of devices at risk from specific vulnerabilities. Certain vulnerabilities are considered to be critical as well as important for any security analysis of Android. These vulnerabilities affect all Android devices irrespective of their manufacturers.

Here, we emphasize our research on exploiting one of the most infamous vulnerabilities of Android – Stagefright. This research is primarily based on **exploit-38226** [3] which was implemented by Google and **Google Project Zero: Stagefrightened** [4].The exploit discussed in the paper was implemented by NorthBit [5]. There's been a lot of confusion about the remote exploitability of the issues, especially on modern devices [4]. Therefore, it is imperative to analyze the various aspects of this vulnerability.

A. *Stagefright*

Stagefright is the collective name for a group of software bugs that affect operating systems running on Android versions 2.2 ("Froyo") onwards. It allows an attacker to perform multiple operations on the victim device through privilege escalation and remote code execution [6].It was discovered by Joshua Drake, Zimperium, and was publicly disclosed on July 27th 2015. Several of its critical heap overflow vulnerabilities were also discussed. 'libstagefright' is an Android multimedia framework library written in C++ language which executes inside Media Server . It supports and handles the processing of all audio and video files (except images). This media server process runs in the background which is a native service at boot from /init.rc. So, the process restarts automatically when it crashes. The last part of the service definition in /init.rc shows the privileges that the service runs with, as shown in Fig.1 which groups the services according to their count.


```
CNT GROUP             PURPOSE
51  3003(inet)        /* can create AF_INET and AF_INET6 sockets */
51  3002(net_bt)      /* bluetooth: create sco, rfcomm or l2cap sockets */
51  3001(net_bt_admin) /* bluetooth: create any socket */
51  1006(camera)      /* camera devices */
51  1005(audio)       /* audio devices */
33  3007(net_bw_acct)  /* change bandwidth statistics accounting */
33  1026(drmrpc)       /* group for drm rpc */
27  1000(system)       /* system server */
20  1003(graphics)     /* graphics devices */
19  1031(mediadrm)     /* MediaDrm plugins */
18  3004(net_raw)      /* can create raw INET sockets */
9   1028(sdcard_r)     /* external storage read access */
8   1023(media_rw)     /* internal media storage write access */
8   1004(input)        /* input devices */
7   1015(sdcard_rw)    /* external storage write access */
4   2000(shell)        /* adb and debug shell user */
4   1001(radio)        /* telephony subsystem, RIL */
```
Fig. 1 Services of Mediaserver along with their privileges

As it is clearly seen, this service is highly privileged. Considering the level of access on all devices, one can monitor, record, and playback audio, access camera devices, connect to hosts on the internet, access and configure the bluetooth.

Before we explain the working of the exploit, it is important to understand the format of the media files. The libstagefright multimedia library works with a wide variety of file formats. We have used Mpeg4 file format for our research. It is a set of multiple Type-Value-Length chunks. Fig. 2 shows the structure of the file.

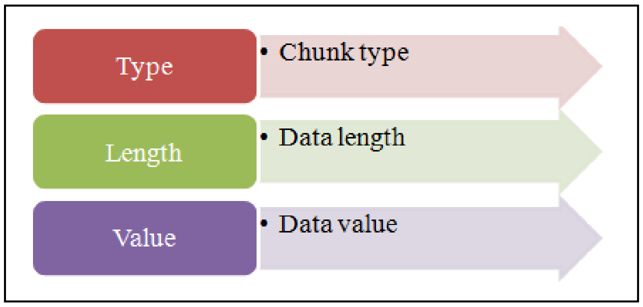


Fig. 2 MPEG4 Chunk Format

The pseudo code for MPEG4 chunks is given as [7]:
```
struct TLV
{
        unit32_t length;
        char atom[4];
        char data[length];
}
```
Here, the atom field (also named as FOURCC), is a sequence of 4 bytes used to uniquely identify the data formats and also describes the chunk type.

CVE-2015-3864 [1**]** helps the attacker to exploit the system by a linear-heap overflow which further authorizes the attacker to control the size of the allocation, the amount of overflow and more importantly the contents of the overflowed memory region. It has been discussed over many articles over the year. This specific vulnerability in libstagefright involves the parsing of MPEG4 files. It specifically involves the parsing of "tx3g" atom used for embedding subtitles (timed-text) in the media file [7]. All the timed-text chunks are collected and appended into a single buffer. By efficiently controlling the size, data and the object address, the attacker can easily shape the heap and exploit the device.'size' and 'chunk_size' are two unchecked variables, which can cause an Integer Flow. Although for Heap overflow to be achieved, we need to have at least one legitimate 'tx3g' chunk, as indicated by the code shown below.

MPEG4Extractor.cpp:

```
case FOURCC('t', 'x', '3', 'g'):
{
   uint32_t type;
   const void *data;
   size_t size = 0;
   if (!mLastTrack->meta->findData(
           // mLastTrack is NULL, SEGMENTATION FAULT
           kKeyTextFormatData, &type, &data, &size))
{
       size = 0;
   }
}
```
Although a patch was given for this specific vulnerability, but the chunk_size actually doesn't have type size_t. It is of type

uint64_t, even on 32- bit systems. The check seems to be sufficient, but the chunk_size can be bigger than SIZE_MAX, which causes the check to pass [4].

III. EXPLOITATION

The exploitation part consists of the techniques used for achieving the target of cracking the device. The most critical vulnerability is exploited in media parsing. Metadata of a file such as dimensions, thumbnails and frame rate are extracted by the process of parsing. For the device to be compromised, the victim's device doesn't even need to play the media, just parse it. The first step towards exploiting a device starts with the vectors leading to that device, known as attack vectors.

A. *Attack vectors*

There exist multiple attack vectors which uses stagefright. An attack vector is a path or means by which an attacker can gain access to a computer or network server in order to deliver a payload or malicious file to the desired place. Attack vectors enable hackers to exploit various system vulnerabilities, including the human aspect [8].

- Mobile networks - (MMS)
- Physical – USB, SD card
- Physically adjacent – NFC, Vcard, Bluetooth
- Client side – Browser, Downloads, Emails

The Web browser acts as an important attack vector as we require Javascript, serving its strengths and limitations. Attack website, XSS (trusted website with malicious content), Hacked website, Ads, Free wifi, QR codes, Man-in-the-Middle etc are some methods to lure victims to land on the desired webpage. In addition to all of this, the victim needs to linger on the webpage for some time. Hence social engineering will affect the efficacy of the vulnerabilities.

B. *Redirection*

Run-time method mapping is supported by vtable which can be further redirected to the heap. The heap can be shaped in a way that the mDataSource object is allocated right after the overflowed buffer. By using the vulnerability discussed, an attacker can overwrite mDataSource's vtable and set the respective readAt entry which points to his own memory. This was implemented in exploit-38226 [3]. It gives full control of the vtable to an attacker by redirecting methods to any code addresses but it does require prediction of the forged table's address, which is possible as shown by Google Project Zero [4].

C. *Heap shaping*

Android uses jemalloc for its heap allocation. The implementation method of the jemalloc is important to understand the exploitation and bypassing of the ASLR. For all we need to know for now is that it allocates the objects of similar sizes in the same run, where 'run' is an array of buffers of the same size. They are called 'regions'. Fig. 3 shows the basic design of jemalloc [9].

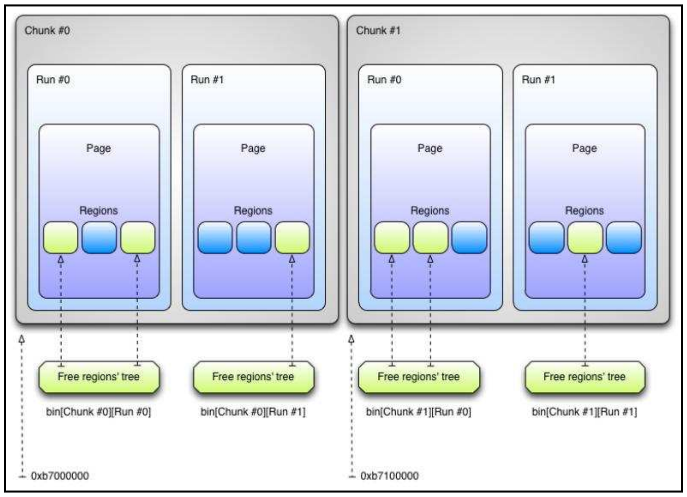


Fig. 3 jemalloc design [9]

A few heap manipulations are required to set up the environment and we can get them by spraying the heap accordingly. **Heap Spraying** is an attempt to insert the code into a predetermined location using the exploits of vulnerable browsers [10]. This takes place by using 'pssh' atoms which provides good primitive in handling their chunks. As soon as the 'pssh' chunk is encountered by the parser, it allocates a buffer and appends it to the list of buffers. The following code gives an insight to the 'pssh' chunk working.

MPEG4Extractor.cpp:

```
   // pssh.data is an allocated block of memory of
controlled size
   pssh.data = new (std::nothrow)
uint8_t[pssh.datalen];
   if (pssh.data == NULL) {
       return ERROR_MALFORMED;
   }
   ALOGV("allocated pssh @ %p", pssh.data);
   ssize_t requested = (ssize_t) pssh.datalen;
   // reading the controlled data
   if (mDataSource->readAt(data_offset + 24,
pssh.data, requested) < requested) {
       return ERROR_IO;
   }
   mPssh.push_back(pssh);
```

The size can be controlled and large values can also be passed. The only limitation is that the media file needs to include the same amount of data but that can also be managed. The order of allocations and deallocations can be controlled to design the heap objects in a predictable manner. This is achieved by the method of **Heap grooming** which is different from heap spraying because of its extensive functions.

Moreover in **exploit-38226** [3], the most useful allocations for this purpose were the handlers for '**avcC**' and '**hvcC**' chunk types. When handling these chunks, the parser allocates the memory block and stores it. When the parser encounters a second chunk of the same type, it replaces that allocation with a fresh one [4]. The code for 'avcC' and 'hvcC' are virtually identical. Here, we look at the code of 'avcC':

MPEG4Extractor.cpp:

```
case FOURCC('a', 'v', 'c', 'C'):
{
   *offset += chunk_size;
   sp<ABuffer> buffer = new
ABuffer(chunk_data_size);

   if (mDataSource->readAt(
               data_offset, buffer->data(),
chunk_data_size) < chunk_data_size) {
       return ERROR_IO;
   }
   /* implicitly copies buffer->data() to a buffer
of size chunk_data_size, and
   releases the previously stored data*/
   mLastTrack->meta->setData(
           kKeyAVCC, kTypeAVCC, buffer->data(),
chunk_data_size);
   break;
}
```

The parser passes the allocated buffer of controlled size to MetaData::setData which further copies the data into a new buffer and deletes the old entry. This method is inconsistent over different devices due to the difference in jemalloc configurations. Therefore, a generalized version of the method is to use the MPEG4-atoms ‘**titl**’, ’**pref**’, ‘**auth**’ and ‘**gnre**’. These atoms are used to extract the title, artist writer and gnre of the media file. Further, they are both processed and parsed by the MPEG-4Extractor.
The temporary buffer size is controlled with chunk_size whereas the actual copied buffer is controlled with the position of null byte (null-terminated string). This lets the attacker to allocate the temporary object in a different run and hence gives him greater flexibility in exploitation. Moreover, the redundant entries are replaced in the **MetaData** and the aforementioned atoms are used to control the heap. While overwriting mDataSource, the location is predicted using the ‘**stbl**’ atom which reallocates mDataSource. The new data source is then used for parsing all sub-chunks contained within the ‘stbl’ chunk.

MPEG4Extractor.cpp:

```
if(chunk_type==FOURCC ( 's','t','b','l'))
{
      ALOGV("sampleTable chunk is %"PRIu64
" bytes long." ,  chunk_size );
  if( mDataSource -> flags ()
     & (DataSource :: kWantsPrefetching
      |  DataSource :: kIsCachingDataSource ))
        {
        sp < MPEG4DataSource >  cachedSource = new
                  MPEG4DataSource ( mDataSource );
   if(cachedSource -> setCachedRange (* offset ,
                            chunk_size )    ==  OK )
        {
                   mDataSource  =  cachedSource;
        }
}
    mLastTrack -> sampleTable  =   new   SampleTable
(mDataSource );
}
```

The ‘pssh’atoms are used to spray the heap such that new heap runs with predictable order. The ‘titl’ and ‘gnre’ atoms are used as placeholders where ‘titl’ atoms are allocated first and then ‘gnre’ atoms are allocated. The ‘gnre’ atoms are then deallocated followed by the allocation of an MPEG4DataSource with the help of ‘stbl’ atom [7].
The ‘titl’ block is freed once it is deallocated and the ‘tx3g’ atom will take its place. Fig. 4 illustrates the working of this process.

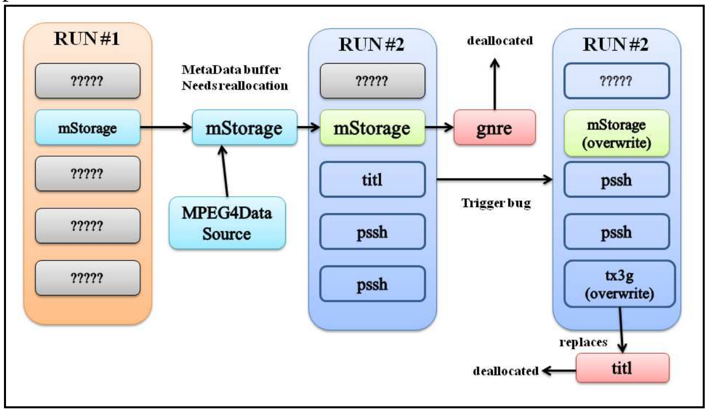
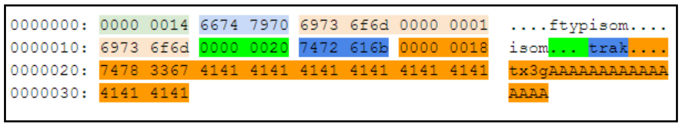


Fig. 4 Process of heap grooming to overflow mDataSource

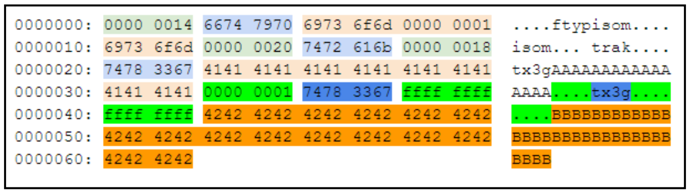
The heap needs to be carefully shaped to ensure a free space directly before the MPEG4DataSource object. The layout of memory in the system is represented in the following text with an example. Here an ‘ftyp’ chunk is used for the file to be detected as an MPEG4 file by libstagefright and parsed accordingly. The basic structure of the chunk used in the depiction is a 4-byte chunk size (green) and a 4-byte tag (blue) followed by the chunk data (yellow). Since we need at least one track to exploit the vulnerable code, a ‘trak’ chunk is used to initialize the mLastTrack and further act as a container to other chunks.



```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0020 7472 616b 0000 0018  isom... trak....
0000020: 7478 3367 4141 4141 4141 4141 4141 4141  tx3gAAAAAAAAAAAA
0000030: 4141 4141                                AAAA
```

Fig. 5 (a) memory layout

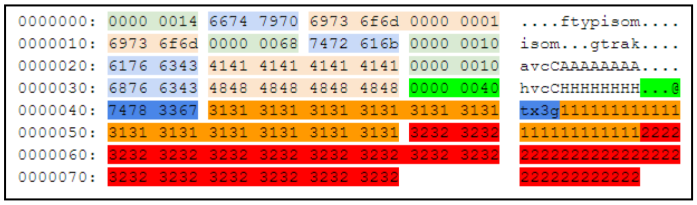
The ‘tx3g’ chunk passed initially which is represented by the value ”A” doesn’t cause an overflow. In order to trigger an overflow, a chunk_size with a large value needs to be passed. Therefore, another ‘tx3g’ chunk is used, represented with the value “B” which seems to be large enough for causing an overflow. Similarly, chunk lengths and size of chunk data can be manipulated to achieve the same.



```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0020 7472 616b 0000 0018  isom... trak....
0000020: 7478 3367 4141 4141 4141 4141 4141 4141  tx3gAAAAAAAAAAAA
0000030: 4141 4141 0000 0001 7478 3367 ffff ffff  AAAA....tx3g....
0000040: ffff ffff 4242 4242 4242 4242 4242 4242  ....BBBBBBBBBBBB
0000050: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
0000060: 4242 4242                                BBBB
```

Fig. 5 (b) memory layout

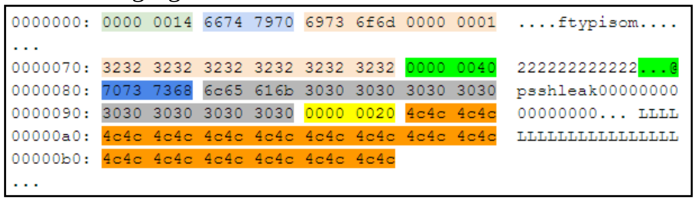
Further, ‘avcC’ and ‘hvcC’ chunks trigger additional temporary memory allocations which may interfere with the grooming of heap. Therefore, they are allocated before the initialization of ‘tx3g’ chunk. The size is the same which is passed during the memcpy function. Here, the data represented by value ‘2’ will be allocated outside the 32 byte allocation and will result in an overflow.



```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0068 7472 616b 0000 0010  isom...gtrak....
0000020: 6176 6343 4141 4141 4141 4141 0000 0010  avcCAAAAAAAA....
0000030: 6876 6343 4848 4848 4848 4848 0000 0040  hvcCHHHHHHHH...@
0000040: 7478 3367 3131 3131 3131 3131 3131 3131  tx3g111111111111
0000050: 3131 3131 3131 3131 3131 3131 3232 3232  1111111111112222
0000060: 3232 3232 3232 3232 3232 3232 3232 3232  2222222222222222
0000070: 3232 3232 3232 3232 3232 3232            222222222222
```

Fig. 5 (c) memory layout

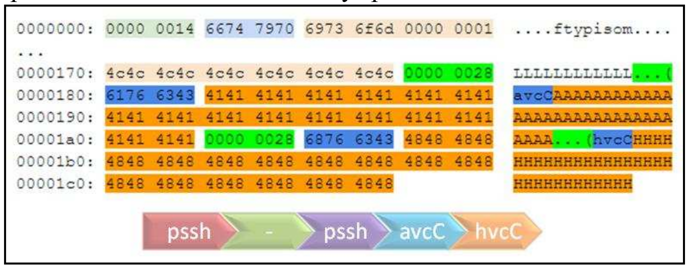
After this, the heap is ready to be shaped accordingly. The first step involves the defragmentation for target allocation size by allocating some ‘pssh’ chunks according to the same target size. These chunks have their own internal structure but we are only concerned with the size and data of the chunk. The size is represented in yellow and the data is represented in orange in the following figure.



```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
...
0000070: 3232 3232 3232 3232 3232 3232 0000 0040  222222222222...@
0000080: 7073 7368 6c65 616b 3030 3030 3030 3030  psshleak00000000
0000090: 3030 3030 3030 3030 0000 0020 4c4c 4c4c  00000000... LLLL
00000a0: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c  LLLLLLLLLLLLLLLL
00000b0: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c
...
```

Fig. 5 (d) memory layout

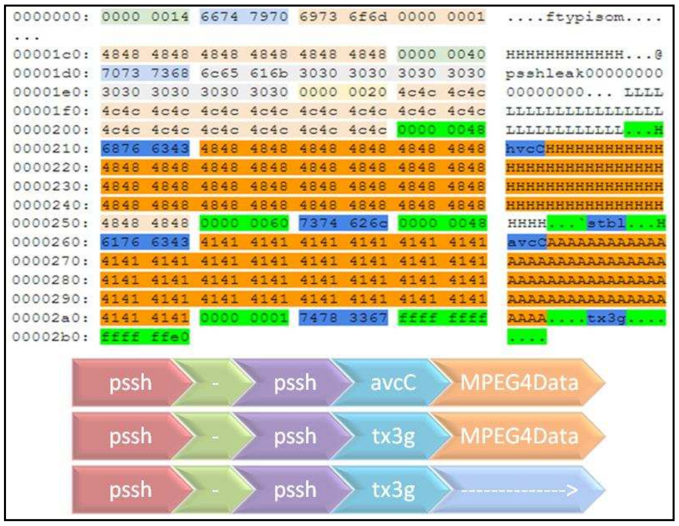
This step is followed by the allocation of ‘avcC’ and ‘hvcC’ chunks of the target size which are assumed to be contiguous. Since there is a temporary allocation which takes place during the parsing of these chunks, so it is possible that there might be some blank spaces between these chunks and the respective ‘pssh’ chunk. This can be resolved by allocating another ‘pssh’ chunk in the void memory space.



```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
...
0000170: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 0000 0028  LLLLLLLLLLLL...(
0000180: 6176 6343 4141 4141 4141 4141 4141 4141  avcCAAAAAAAAAAAA
0000190: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00001a0: 4141 4141 0000 0028 6876 6343 4848 4848  AAAA...(hvcCHHHH
00001b0: 4848 4848 4848 4848 4848 4848 4848 4848  HHHHHHHHHHHHHHHH
00001c0: 4848 4848 4848 4848 4848 4848            HHHHHHHHHHHH
```

Fig. 5 (e) memory layout

The ‘hvcC’ chunk can then be freed to trigger the allocation of MPEG4DataSource. This leaves us with only one more step. The ‘avcC’ chunk needs to be released to trigger the ‘tx3g’ overflow inside the ‘stbl’ atom.



```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
...
00001c0: 4848 4848 4848 4848 4848 4848 0000 0040  HHHHHHHHHHHH...@
00001d0: 7073 7368 6c65 616b 3030 3030 3030 3030  psshleak00000000
00001e0: 3030 3030 3030 3030 0000 0020 4c4c 4c4c  00000000... LLLL
00001f0: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c  LLLLLLLLLLLLLLLL
0000200: 4c4c 4c4c 4c4c 4c4c 4c4c 4c4c 0000 0048  LLLLLLLLLLLL...H
0000210: 6876 6343 4848 4848 4848 4848 4848 4848  hvcCHHHHHHHHHHHH
0000220: 4848 4848 4848 4848 4848 4848 4848 4848  HHHHHHHHHHHHHHHH
0000230: 4848 4848 4848 4848 4848 4848 4848 4848  HHHHHHHHHHHHHHHH
0000240: 4848 4848 4848 4848 4848 4848 4848 4848  HHHHHHHHHHHHHHHH
0000250: 4848 4848 0000 0060 7374 626c 0000 0048  HHHH...`stbl...H
0000260: 6176 6343 4141 4141 4141 4141 4141 4141  avcCAAAAAAAAAAAA
0000270: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0000280: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0000290: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00002a0: 4141 4141 0000 0001 7478 3367 ffff ffff  AAAA....tx3g....
00002b0: ffff ffe0                                ....
```

Fig. 5 (f) memory layout

As we have seen the possible way of shaping the heap of a system, it cannot be achieved unless the exact locations are not identified. It is important to analyze the ASLR to achieve the discussed exploitation technique.

D. *ASLR at Risk*

In order to break the ASLR, we need some information about the device. Different devices have different configurations and therefore have a high probability of having different memory offsets or the preferred base addresses. But it is plausible to get an arbitrary pointer which leaks the information back to the web browser. This is achieved using the discussed vulnerability in order to break the ASLR [7]. But there are various limitations which hampers the ability to read the memory. Certain assumptions are also made to explain the working of the exploit. The various limitations are as follows:

- **JavaScript Capabilities** – Since the attack is via the web browser, the various properties of JavaScript affects the exploitation of a device. Arbitrary memory addresses can be sent back to the browsers which are further used by JavaScript to overwrite the pointer pointing to the original metadata.
- **Returning of MetaData** – The metadata is returned to the browser only when `mInitCheck` inside the ‘moov’ chunk is set to OK, which is set only when this chunk is parsed. Therefore, the presence of moov chunk guarantees the returning of metadata.
- **Returning MetaData after Overflow** - The same media file can’t be reused to execute multiple overflows because the returning of ERROR_IO from the MPEG4Extractor::`parseChunk` can’t be avoided once the ‘tx3g’ bug is triggered.

- **Bypassing process termination** - While streaming videos using HTTP, the mediaserver is terminated because of the high value of 'size'. But the FileSource::`readAt` method helps in bypassing the termination of process by not using any macros**.**
- **Leaking Information** – The metadata is sent back to the browser after being parsed by the mediaserver and is stored inside `MetaData` objects. The data is further stored in `mItems` fields which are actually the dictionary of FOURCC keys to `MetaData::typed_data` values. KeyedVector is defined as [7]:

```
KeyedVector < uint32_t , typed_data >
  mItems;
```

and `typed_data` is defined as [7]:

```
struct  typed_data
{
     uint32_t  mType;
     size_t  mSize;
  union
  {
     void   * ext_data;
     float  reservoir;
  } u;
}
```

Here we can see that ext_data and reservoir share the same address. So if (mSize >4), ext_data will point to the data's memory location. Otherwise, reservoir will contain the data.

The **KeyedVector** objects store the data in their mStorage field which is an array of keys and typed_data elements. If the contents of mStorage array are overwritten, the metadata pointers can also be overwritten to point to arbitrary memory locations. **Hence, leaking the information to the browser.**

Any size greater than 4 bytes is considered to be a pointer. But as we can control the size, the usage of unused and needless metadata pointers is avoided. Since the mSize field is supposed to be greater than 4 bytes, the memory leak can only be achieved through 'duration' field, which is 8 bytes long and hence a pointer as well. videoWidth and videoHeight are also 4 bytes long and cannot be used to leak memory. Even if we attempt to increase the sizes of these fields, the process will be terminated.

The **KeyedVector** stores the data sorted by the key value. Therefore, we need to overwrite the array of 16 bytes structure with sorted elements of type: key_value_pair <key, value>. Further, the grooming of heap is carried out in a similar fashion as explained earlier using the same heap overflow vulnerability i.e. **CVE-2015-3864** [1] and the data of duration field is ultimately converted from microseconds to milliseconds, which causes some data loss.

The browser filters the negative and infinite values as their highest bits are set. In that case, these values are ignored and the 'duration' field is set to zero. Since the PRId64 formats the cross platform printing of a 64-bit signed integer, the largest possible value which is valid in this entire process is calculated as:

(2^31 – 1) *1000 + 499 = 2,147,483,647,499

= 0x000001F3; FFFFFE08 (1)

The values higher than this value will overflow to the sign bit and the browser will filter them as negative or infinite values for 'duration' field. This will deter any leakage of information as well. Therefore, 8-bytes can only be leaked if the higher 23 bits are filled with zero, which gives an attacker access to about 32-35 bits of data, depending on the value.

E. ASLR Weakness

Address space layout randomization (ASLR) is a memory-protection process for operating systems (OS) which guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory [11].The memory offset varies from 0 – 255 pages and the amount of paged shifted, is called the **ASLR Slide**. It is generated at the initiation of a process and persists throughout its lifetime.

The mediaserver is executed hundreds of time to record the possible address ranges of the module [7]. And it is seen that each and every module is first loaded to the base address and then shifted down by ASLR slide. This states that ASLR slide is same for all the modules (except some devices which load the modules dynamically). Considering the aforementioned statements, a single base address is essential to shape the memory layout of the modules.

All the required gadgets reside in 'libc.so', and their gadget offsets changes between different devices in respect to the different versions of libc.so. So, building a lookup table involving the device-version and gadget offsets eliminates the processing of expensive operations. This leaves only base address of libc.so as the only requirement for remote code execution at runtime.

The ASLR slide is defined as:

```
      final_base = preferred_base - aslr_slide
```

therefore,

```
      aslr_slide = preferred_base - final_base
```

As we know, the **Executable and Linkable Format** (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, core dumps and shared libraries [12]. It is not bound to any particular processor in addition to its flexibility and extensibility. Fig. 6 depicts the format of an ELF header. Therefore, it is widely adapted by various operating systems. The ELF header needs to be page-aligned and placed at the beginning of the executable region.

Once we go down from the preferred base, we eventually land on the ELF header. But the ELF header can’t be leaked because the 8-bytes value is greater than the limitations of this specific method. In response to this, other field (p_memsez and p_flags of the third header table at offset 0x88 [7]) is searched which is unique to each module and **the data is leaked from there.**

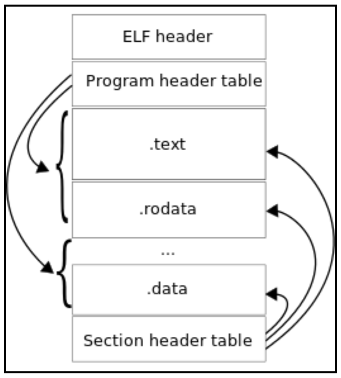


Fig. 6 ELF Header format

Since this method is useful in leaking only few bits of information through these fields, the victim needs to download and parse up to 256 media files just to locate the ELF header. Moreover, the code execution file might be large in size due to heap grooming.

IV. IMPLEMENTATION

The techniques discussed above forms different modules to exploit the device in real time. They are segregated into three modules:

- ➢ Crash
  - ✓ Creation of a generic media file
  - ✓ Crashing of mediaserver for state recovery
  - ✓ Checking the existence of bugs while building the lookup tables
- ➢ Remote Code Execution
  - ✓ Creation of a device-specific media file which executes shellcode in mediaserver
  - ✓ Lookup table provides the offsets and preferred base address
  - ✓ ASLR slide is received as a parameter according to which the offsets are converted into absolute addresses
- ➢ Memory Leak
  - ✓ Creation of a device-specific media file to leak memory from mediaserver
  - ✓ Receiving of address to leak is used as a parameter(this might be unmapped, which results in a crash)
  - ✓ Information retrieval through <duration> field of <video> tag
  - ✓ Requirement: web browser supporting XmlHTTPRequest with blob response type.

The following flowchart depicts the entire flow for exploitation:

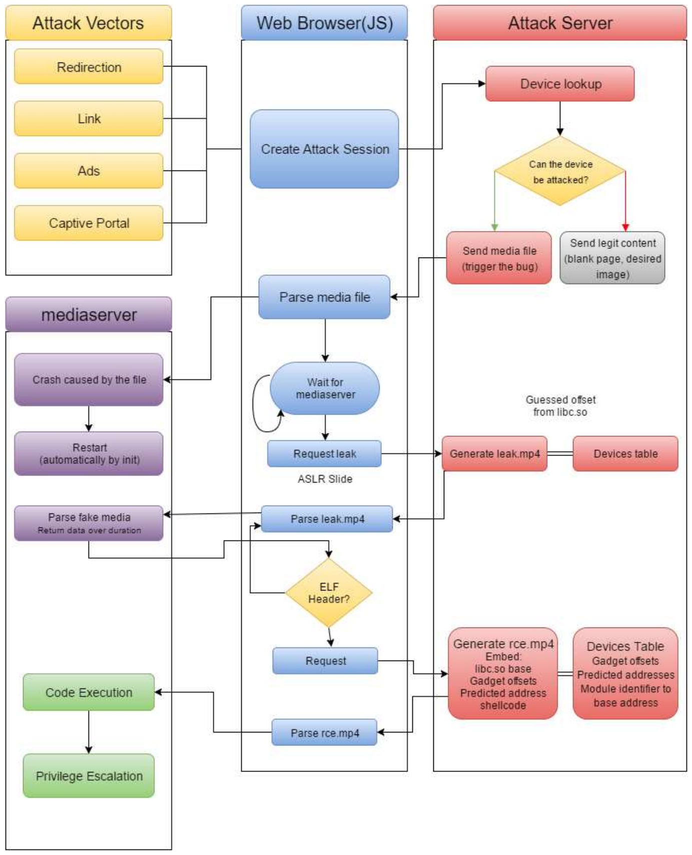

Attack Vectors
Redirection
Link
Ads
Captive Portal
Web Browser(JS)
Create Attack Session
Attack Server
Device lookup
Can the device be attacked?
Send media file (trigger the bug)
Send legit content (blank page, desired image)
Parse media file
mediaserver
Crash caused by the file
Wait for mediaserver
Restart (automatically by init)
Guessed offset from libc.so
Request leak
ASLR Slide
Generate leak.mp4
Devices table
Parse fake media Return data over duration
Parse leak.mp4
ELF Header?
Request
Generate rce.mp4 Embed: libc.so base Gadget offsets Predicted address shellcode
Devices Table Gadget offsets Predicted addresses Module identifier to base address
Code Execution
Parse rce.mp4
Privilege Escalation

Fig. 7 Flowchart of working of the exploit

V. PREVENTION

The effectiveness of exploitation discussed in the paper can be further increased by adopting some measures such as:

- The efficiency of heap spraying can be increased by using the ‘stbl’ atom recursively. The wrapping of spray data with the ‘stbl’ atom can double the efficiency, as illustrated in detail in NCC Group Paper [13]. The size of the remote code execution exploit can also be reduced significantly using this method.
- The number of leaks required can be significantly reduced by leaking different information from the ELF header. Random addresses in rich memory regions can be chosen instead of ELF headers. The exploitation time can be reduced by using this method depending on the amount of leaks required, device, identifiers and some other factors.

Keeping all of this in mind, the security of Android devices has become a point of great concern in today’s world. Although the Android OS is being continuously fixed by patching and upgrading, many devices are still vulnerable to the exploit mentioned in the paper. The Android Ecosystem takes time to alleviate the vulnerabilities due to its eclectic nature. It is necessary to explore some preventive measures which may help in building a protective environment for android devices. We present two methods which will help in achieving the same.

Hardening the memory mapping will drastically reduce the efficacy of heap spraying and making it harder for the attacker to gain controlled access at the known address locations. In order to achieve that, Android can use a similar technique used by Chrome known as PartitionAlloc() to improve the weak randomization of memory mapping.

Hardening of 'libc' implementation will also be fruitful in ameliorating the exploitation. The implementation of pointers to mangle the functions like setjmp and longjmp will be beneficial as far as the security of device is concerned. It will prevent an attacker from using these functions as a stack pivot as it protects the corruption of jmp_buf structures.

Although these techniques doesn't guarantee the non-exploitability of memory corruptions in Android devices but they will definitely increase the cost of attackers in exploiting the devices and be helpful in making the device more secure from stagefright attacks in the future

VI. CONCLUSION

Since Android OS dominates the smartphone market and is widely used across the globe. It is worth noting that only 19.4% of the Android devices run versions 5.1. Metaphor, the exploit [7] works in Android versions 2.2 – 4.0 and 5.0, which estimates about 275 million devices still vulnerable. libstagefright continues to be a very vulnerable library which affects numerous devices and has good potential attack vectors such as MMS, Instant Messaging and Web Browsing. We have presented vulnerability in the memory mapping technique in this library which allows an attacker to break a device. CVE-2015-3864 [1] is one such vulnerability which plays a major role in leaking the information through an 8 byte 'duration' field. A point worth noting is that we can leak 8 bytes integer data back to the web browser with an accuracy of +500 or –500. It is device specific for obvious reasons, and exploitation is slightly different in terms memory addresses but the basic idea is same throughout. As mentioned in the research by NorthBit, stagefright vulnerability has been tested for Nexus5 with stock ROM, HTC One, LG G3 and Samsung S5 [7]. Since the privileges of mediaserver differ among the vendors, an attacker can easily exploit a device with the suitable modifications. The patches are available from Google for versions 5.1 and higher but the exploit still exposes some of the devices running those versions. This shows that it is necessary to patch the existing vulnerabilities to protect existing devices and provide more secure environment for the future devices.

REFERENCES

[1] CVE-2015-3864, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3864

[2] Daniel R. Thomas, Alastair R.Beresford, Andrew Rice, "Security Metrics for the Android Ecosystem", ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) 2015,
DOI: http://dx.doi.org/10.1145/2808117.2808118.

[3] exploit-38226, https://www.exploit-db.com/exploits/38226/

[4] Google Project Zero: Stagefrightened, http://googleprojectzero.blogspot.in/2015/09/stagefrightened.html

[5] NorthBit, http://north-bit.com/

[6] Stagefright(Bug), https://en.wikipedia.org/wiki/Stagefright_(bug)

[7] Metaphor: A (real) real-life Stagefright exploit, Researched and implemented by NorthBit. Written by Hanan Be'er.

[8] Attack vectors, http://searchsecurity.techtarget.com/definition/attack-vector

[9] Patroklos Argyroudis and Chariton Karamitas, Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap, Census Black Hat USA 2012

[10] Heap Spraying, http://www.pctools.com/security-news/heap-spraying/

[11] ASLR, http://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR

[12] ELF header, https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[13] NCC Group Paper, "A few notes on usefully exploiting libstagefright on Android 5.x ", prepared by Aaron Adams.